1998 International Computer Symposium
Workshop on Artificial Intelligence
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

A DEPENDENCY-BASED CONSTRAINT RELAXATION SCHEME FOR OVER-CONSTRAINED PROBLEMS

Chia-Lin Hsieh[†]*, and Jackie Archibald*

School of Computing and Information Systems,
University of Sunderland, Sunderland, U.K.
Email: {cs0mhs,cs0jar}@isis.sunderland.ac.uk

ABSTRACT

In this paper, we propose an incremental constraint solver in order to realise a constraint relaxation system for over-constrained problems. There are two requirements to be considered. Firstly, the constraint relaxation is based on constraint propagation techniques to maintain local consistency. Secondly, in the case of inconsistency, it proposes conflict candidates to retract only those constraints which are really involved in the contradiction. Thereby, no unnecessary modifications of the original CSP are executed. What is crucial is that the relaxation operation is incremental, in that it maintains a dependency graph among variables and constraints and updates only subset of the graph which is the real responsibility for the failure. Moreover, this system borrows intelligent backtracking techniques in determining, upon the inconsistency, a pertinent choice point which can restore the satisfiability. Therefore, these considerable terms would support constraint relaxations and experiments have shown arbitrarily speedups compared with previous approaches.

1. INTRODUCTION

Constraints are nowadays widely used for solving problems arising in various fields (e.g. Artificial Intelligence, Operations Research,...). As noticed by G. Verfaillie and T. Scheix [1], many problems in those fields can be viewed naturally in the form of constraint satisfaction problems (CSPs). A classic CSP involves a set of variables each with an associated domain and a set of constraints. The problem is to find an assignment of values to variables which satisfies all constraints.

Many real world problems are often *over-constrained*[1] (e.g. timetabling, scheduling). To provide an acceptable solution to the user, it is essential to obtain a constraint relaxation[2] system which may relax some constraints to obtain a partial solution satisfying as many constraints if possible. However, the users of those constraint relaxation systems normally do not have enough information *a priori* to specify the constraints to be relaxed. In this paper, we suggest a constraint relaxation approach which relies upon the main considerations:

- When a contradiction occurs in an over-constrained problem, a constraint to be relaxed should be automatically identified by considering from the conflict set the real effects on the variables of the problem. We call this part the *identification*.
- When relaxing a constraint, a complete re-execution must be avoided as much as possible. For that, information on the constraint past effects should be kept in order to ensure a real *incrementality* despite the necessary modifications of the systems. That is what we call: the *bookkeeping*.

From our point of view, the current requirements may be achieved by maintaining useful information during search in order to realise more efficient relaxations. Another motivation is the necessity of *efficient failure analysis* to cope with the detection of inconsistent situations (where no traditional solution is possible) and identify the minimal conflict set. Our approach realises the philosophy just to change those parts which are really influenced directly by the modification; so an entire re-computation can often be suppressed.

In this paper, we present an incremental algorithm for the CSPs to handle constraint relaxations for over-constrained problems. Our approach consists in formulating the constraint system over finite domain by continuously maintaining a *specific dependency graph* on the current constraint system. When contradiction is verified, the backtracking is performed to relax some constraints. Our approach relies on the notion of *configuration*. The configuration is the splitting of the constraints of the problem into two sets: *active* and *inactive*. A non-contradictory configuration is then determined.

The problem solving approach consists in two phases: an *enumeration phase* and an *evaluation phase*. The former phase not only performs a pre-processing to exploit all dependent constraints but also keeps track of changes for future use. The latter one performs failure analysis and incremental revision. The key idea is to benefit from the past computation, i.e. to provide an incremental search algorithm to keep as much and precisely as possible of the

[†] Also Department of Industrial Management, Tamsui Oxford University College, Tamsui, Taiwan.
[1] The associated constraint system possesses no solution.
[2] Here we mean by relaxation: the relaxation of past and last effects of a constraint.

computation already done by redoing only the sub-constraint. Only in this way we can hope to get a reasonably efficient constraint relaxation system. Preliminary experimental results, on both real-life and general classes of problems, are very encouraging, since they show that this way of achieving constraint relaxation is much more efficient than re-computing from scratch.

The following section recapitulates related constraint approaches. Section 3 illustrates the basic specification and formulation of the constraint system. We also state its basic properties, and fix the terminology used throughout the paper. Section 4 puts forward the proposed scheme for failure analysis and intelligent constraint updating. The actual procedure is depicted subsequently. Section 5 gives a performance evaluation of the algorithm on various problems and Section 6 concludes the present paper by summarising its results.

2. RELATED WORKS

In this section, we first review some works about efficient constraint relaxations in dynamic CSP problems. A few algorithms which allow efficient modifications of constraints to achieve dynamic arc-consistency have been evaluated [2][3]. During the constraint relaxation, with the help of *justifications*[3], it can incrementally add to the current domain values that belong to the new consistent domain. In the case that the considered constraints are qualities and inequalities over finite domains, a specialised incremental re-propagation methods is proposed in [5].

The logic programming community has been interested for a few years in intelligent backtracking for constraint relaxation. The results of this study have been embedded in the CLP(FD) framework. The IHCS system (Incremental Hierarchical Constraint Solver) [6] is one of the successful system. IHCS works on intelligent backtracking and aims to provide incremental problem solving. It uses the same justification as DnAC4. Moreover, it also suggests the constraint identification for failure analysis with the help of a dependency graph. Despite its incremental handling of constraint additions, however, constraint relaxations are handled in the hard way: using backtracking *non-incrementally*. IHCS is then not a completely satisfactory answer to handle constraint modification problems arising in dynamic environments.

Fages *et al.* [7] proposed a reactive schema in CLP(FD) that can efficiently add and relax constraints. A sound abstracted framework is presented in terms of transformation of CSLD derivation trees. In [8], Fages *et al.* proposed an extended execution scheme on finite domain for reactive systems by retaining similar dependency as [6]. The aim is not only to give a solution to an instance of a problem but also to maintain such a solution through interactions with computations directly depending on the modified the users. The execution model had successfully applied to practical planning problems in a reactive system.

Therefore, the general idea of maintaining information seems the wisest thing to do when handling dynamic updates. Our proposal provides the necessary machinery from this point of view and extends Fages' work [7].

3. FORMAL SPECIFICATION

This section briefly introduces some of the concepts and terms associated with the problem described in this paper. It first presents a graph-theoretic formulation of the constraint system.

3.1 Graph-Theoretic Formulation

Constraint problems and the algorithms that satisfy them are commonly expressed in terms of graphs. Let $G = (V,C,E)$ be an undirected bipartite graph, where V and C are set of vertices representing the variables and constraints, respectively. E is a set of undirected edges denoting the graph-theoretic relationship between variables and constraints. For each variable v in a constraint c, E contains an edge between v and c. Figure 1.(a) describes a graph representation of a constraint system. The circles denote variables and the boxed denote constraints. For example, in Figure 1.(a), variables v, v' and v'' are *constraining variables* of the constraint c.

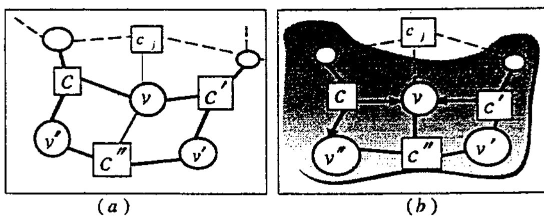

Figure 1: Graph Representation of A Constraint System

We consider a property P that is defined for sets of constraints[4]. P is necessary (but not necessarily sufficient) for the satisfiability of the constraint system. It thus represents a certain level of consistency. A constraint system is called *P-satisfiable* if P holds and *P-contradictory* otherwise. A constraint solver partially directs the edges in G by performing certain level of deduction in order to make the constraint system *P-satisfiable*. A directed graph DG is called a *solution graph* if all the enforced constraints in G are *P-satisfiable*. When a constraint removes values from one of its constraining variables, a directed edge is built from the constraint to the constraining variable. For exam-

[3] The notion of justification is inherited from the TMS community [4] to record constraint deductions.

[4] For instance, we may consider global consistency for rationals or local arc-consistency for finite domain or intervals [9].

ple, in Figure 1.(b), we represent a partially directed graph of Figure 1.(a).

In additions, some definitions must be introduced before preceding. *Constraint store S* contains a set of constraints ordered by topological order. A constraint is considered to be *active (or enforced)* if the constraint satisfies all its constraining variables in the constraint store[5]. In graph-theoretic terms, a constraint is considered to be enforced if the constraint solver includes the constraint in the solution graph. A *configuration* is a split of a given constraint system in two sets: the set *AS* of *active constraints* and the set *US* of *unenforced* ones. It is noted as <*AS, US*>. A configuration may be seen as a state of the evaluation where *AS* contains all the active constraints that might have deduced some domains of its variables and *US* is composed by the unenforced or retracted constraints that are the set of candidates waiting for activation or re-activation. A configuration is *promising* if all the constraints in its active store satisfy the property *P*.

The domain of the variable v is denoted by D_v and C_v ($C_v \subset C$) is the set of active constraints on v. For each constraint, the set of the constraining variables is designated by V_c ($V_c \subset V$). We also define the *clashing function* $\Delta_{v\downarrow c}$ to denote the set of deduced values from the variable v by the constraint c. A constraint c ($c \in AS$) is a *constrainer* of v ($v \in V_c$) if $\Delta_{v\downarrow c} \neq \phi$; that is, it actually causes the reduction of the domain of variable v. The current active constraint store is defined implicitly as $AS = \bigcup_{v \in V} C_v$. For example, as shown in Figure 1.(b), $C_v = \{c, c', c''\}$, $C_{v'} = \{c\}$, $C_{v''} = \{c, c''\}$, $V_c = \{v, v', v''\}$ so that $AS = \{c, c', c''\}$.

3.2 Dependency Graph

The dependency between the active constraints reveals certain information and must be kept and updated during the *enumeration phase* (see Section 4). When a contradiction occurs, the *evaluation algorithm* is evoked which will analyse the dependency between constraints to find out the pertinent causes of inconsistency and which will be affected by the relaxation of some constraints.

To represent the dependency relationship easily, we consider a graph-like description of the constraint store, called the *constraint dependency graph* (see Figure 2). This graph is a sub-graph of the constraint graph with a directed edge (with the dashed arrows) from c to c' iff variable $v \in V_c$ such that c is a constrainer of v which is checked by the constraint c'. Note that the constraint dependency graph is not optimal in the sense that it forgets which constraints removed which value from a variable domain.

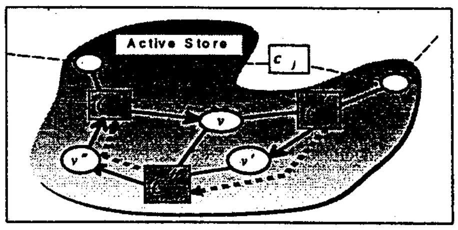

Figure 2: Representation of the Dependency Graph

A constraint c' is a *son* of c iff there is an arc from c to c' in the dependency graph. Conversely, c is a *parent* of c'. Furthermore, constraint c' is a *descendant* of c iff there exist a path from c to c' in the dependency graph and c is an *ancestor* of c'. Here we use the symbol "$\mapsto$" to denote the immediate dependency between a parent c and its son c' with the variable v in common. As shown in Figure 2, c'' is a descendant of c because $c \mapsto c'$ and $c' \mapsto c''$.

The dependency relation is based on *local propagation* of constraints in the following way: whenever a constraint c makes a deduction on some variable v, any other constraints which are the sons of the constraint c will be re-activated and probably cause the re-activation of further dependant constraints. The deduction performed by c will consequently affect all those *"downstream"* constraints and for this reason they will become dependent on c.

Note that the dependency graph is not optimal in the sense that all the analysis is done from the constraints so that a lot of information will be lost about values in domains of variables, then the dependency analysis becomes too general to handle the constraint modification. From our point of view, this dependency graph is improvable if the *value-based* information and operations are maintained. The set of value-based operations performed for the evaluation, such as removal or addition of a value from a domain variable, is called a *transaction.* The set of values and constraints modified by a transaction is called a *transition set.* The transition set includes newly added dependency constraints and any values previously dependent on updated constraints.

The main work of the algorithms given in this paper is to maintain the transactions of the transition set in an extended dependency graph. In our constraint system, two transactions called E_v and Δ_v are included. E_v denotes the current domain of the variable v and Δ_v is a *revision set* which covers a list of *labels* derived from the removal

[5] In a constraint graph *G*, *bald lines* are used to represent the active status of a constraint and *dash lines* are used if the constraint is still *un-enforced* (i.e. *inactive*).

of the domain values from the variable v complying with its constrainers. Therefore, the *clashing function* (defined in Section 3.1), denoted by $\Delta_{v\downarrow c}$, can be recognised as the *revision set* Δ_v *with respect to* the constraint c. In Section 4, we will show the gains of the conflict resolution procedure derived from this modified dependency graph.

4. AN INCREMENTAL CONSTRAINT PROPAGATION APPROACH

4.1 The Enumeration Algorithm Overview

The enumeration algorithm is an adaptation of a local arc-consistency algorithm based on constraint propagation. In our implementation, we adapted the AC-5 like [10] approach. Whenever a new constraint is added to the current constraint store, the corresponding constraints will make some restrictions to some variable domains and such a restriction propagates to other variable domains through the constraint graph.

In words, this algorithm (see Figure 3 for details) considers a set of constraints added to the current active constraint store. Each constraint, say c, is incrementally added to the active store (Line (1)) to obtain a new promising configuration. A counter *order* (Line (2)-(3)) is increased any time a new constraint is introduced in the active store. The function *Revise* performs the removal of inconsistent values from the domain of its constraining variables V_c and updates a dependency graph. If the constraint c is enforced, all of its constraining variables will be considered (Line (7)-(10)). The procedure *Revise_Domain* (Line (12)) will then keep all the transactions made to the variable v in the revision set Δ_v. If the constraint c causes any reductions of variable v, it is pushed into the constrainer set CS_v for future reuse (Line (13)-(15)). If the restriction doesn't raise a *contradiction* (*i.e.* the current domain is not empty), all the consequent constraints of the constraint c are put successively into the stack to be propagated (Line (19)-(22)). Otherwise, the set of the constrainers in CS_v is considered as a *contradiction explanation* (i.e. a set of antagonistic constraints) (Line (16)-(17)) and is pushed into a *conflict set* ψ (see Section 4.3 for detail) and also the function *Revise* returns *false* (Line (18)). After the enumeration procedure, the *evaluation* procedure will proceed to implement conflict resolution, if the failure is detected.

4.2 Incremental Constraint Relaxation

In this section, we will present the incremental algorithm which will allows constraint relaxation from the constraint system for conflict resolution. Whenever any variable domain is modified, the domain of its dependent variables is changed as well, and this triggers a whole propagation enumeration where several domains may be changed until *satisfiability*. The operation of relaxing a constraint might cause a domain enlargement due to the less restrictive effects. Based on this propagation concept, we propose an incremental operation which redoing only a subpart of the constraint system directly depending on the modified constraint. This approach also enables us to retain efficiently the satisfiability from the contradiction (see Section 4.3 for details).

```
Procedure Enumeration(Direct_Change: Set of Constraints)
Global Var
    order : integer; ψ: conflict set;
Local Var
    v : variable;  c_new : constraint; S : constraint store;
Begin
  S := φ;
  For ∀ c_new ∈ Direct_Change Do Begin
    Add(c_new);
    If Failure Then Evaluation(c_new); End;

Procedure Add(Con: constraint);
Begin
(1)  Push(Con, S);
(2)  order := order + 1;
(3)  c_new.order := order;
(4)  While ¬EmptyStack(S) Do Begin
(5)      Pop(c);
(6)      IF ¬Revise(c) Then Failure := true; End;
// Perform the removal of inconsistent values from the domain of
 c constraining variables and update their revision functions;

Function Revise(Con: constraint);
Begin
(7)  Con.Enforced:= true;
//Constraint Con is activated
(8)  For ∀ Var ∈ $V_{Con}$ Do Begin
(9)       Push(Con, $C_{var}$);
(10)      Var.Influenced := false;
(11)      Old.Δ := Var.$\Delta_{Var\downarrow con}$;
(12)      Revise_Domain(Con, $\Delta_{var}$);
//Perform Local Arc-Consistency and delete inconsistent values from
 $D_{Var}$ and put in Var.$\Delta_{Var\downarrow Con}$
(13)      If Var.$\Delta_{Var\downarrow con}$ ≠ Old.Δ Then Begin
(14)        Var.Influenced := true;
//Mark the Variable Var constrained by Con
(15)        Push (Con, $CS_{var}$); //Collect Clashing Constraints
(16)        If $E_{Var} = \phi$ Then Begin
(17)                Push($CS_{Var}$, ψ);
//Push all the contradiction explanations to the conflict set ψ
(18)                Return false; End;
//Variable reaches the dead-end, return failure
(19)      For ∀ Var ∈ $V_{Con}$ Do Begin
(20)                If Var.Influenced Then
(21)                For ∀ c_son ∈ $C_{Var}$ \ {Con} Do
(22)                If c_son.Enforced Then
                    Push(c_son, S);
//Maintain the Dependency of the constraint Con
Return true;
```

Figure 3: The Enumeration Algorithm

Consider a constraint c to be retracted from a constraint system. The relaxation of c needs to be propagated. The aim of this propagation is to "*untrail*" the past effects of the retracted constraints and delete them. It is performed through the constraint graph as long as domain should be enlarged and terminates when reaching variables whose domain is irrelevant to the relaxation. That is, the enumeration algorithm has lead to several restrictions of D_v, the domain of v. Also, each restriction of D_v may have re-activated of all the consequences of c, leading to further domain restriction for other variables. All of these dependent reductions have to be undone when c is retracted. We thus achieve our goal of minimal modification of the constraint store, as those parts which are independent of the relaxation are not considered.

```
Procedure Retract (Con: constraint)
Begin
(1)  $S := \phi$;
//S stores the consequent constraints by the retraction of Con
(2)  $OS := \phi$;
//OS stores the consequent opponent constraints which deduce the constraints visited
(3)  Con.Enforced := false;
(4)  Push(Con, Retracted_Cns );  /
/Retract Con and push it into the retracted constraint store
(5)  Mark(Con);
(6)  While $S \neq \phi$ Do Begin
(7)      $c$ := Pop (S);
(8)      If $c$.Enforced Then
(9)      Mark( $c$ ); End;
(10) Local_Add(OS);

Procedure Mark(Con: constraint);
Begin
(1)  For $\forall$ $Var \in V_{Con}$ Do Begin
(2)  If $\neg Var.Mark$ and $\Delta_{Var \downarrow Con} \neq \phi$ Then Begin
     //Var is not visited yet and deduced by Con
(3)      $Var.Mark := true$ ;
//Enlarged VariableVar is marked as visited.
(4)      $D_{Var} := D_{Var} \cup \Delta_{Var \downarrow Con}$ ;
//Put back clashing values constrained by Con
(5)      $\Delta_{Var \downarrow Con} = \phi$ ;
(6)      Push ( $C_{var} \backslash \{Con\}$,S); End;
(7)  Else If $\Delta_{Var \downarrow Con} \neq \phi$
//Var is deduced by Con and had deduced by other opponent constraints of Con.
(8)      Push (Con, OS);
```

Figure 4: The Constraint Relaxation Algorithm

Some existing approaches to undo such domain reductions used traditional *trailing* mechanism to record during the enumeration (and thus during procedure *Add*) those revision values individually in a *trail* stack [6]. Here we present an extension of the constraint relaxation algorithm [5] which requires no extra trailing but maintains local propagation. This approach proceeds in the following phase (See Figure 4 for details):

1. Consider the retracted constraint c. The values of each constraining variable whose validity relies upon the retracted constraint c are put back to their respective domain. Hence, it will result in the enlargement of the variables. There are two stores used in procedure *Retract*. Store *S* stores the consequent constraints of the relaxation constraint while store *OS* keeps the opponent constraints. Line (5)-(9) in procedure *Retract* performs propagation to activate all the consequent constraints.

2. All of the variables affected by the relaxation of c are considered to be enlarged. However, a lazy strategy is used to prevent the enlargement in becoming too general[6] by considering constrainers constraining simultaneously on a variable. Each time when a variable is enlarged, we *mark* the corresponding variable and restore the reduced values to the current domain (see Line (2)-(6) in procedure *Mark*). If the variable is marked, no enlargement on this variable will be taken until the *untrailing* propagation is done. Instead, we push those constraints in an *opponent stack* for further consideration (see Line (8) in procedure *Mark*).

3. After the *untrailing* propagation, another restrictive enumeration procedure on those opponent constraints will be activated to obtain *local satisfiability*. Line (10) in procedure *Retract* calls procedure *Local_Add(OS)* to perform domain revisions on those variables only which are constrained by the constraints in *OS*.

4.3 Evaluation Algorithm (Conflict Resolution)

As we mentioned before, one of the requirements of a constraint relaxation system is the ability to automatically identify the constraint to be relaxed for conflict resolution. In this section, we describe an evaluation approach to obtain this requirement. The evaluation algorithm is used to resolve any real causes of contradictions during the enumeration phase and resolve the inconsistency by performing minimal constraint relaxations. The basic technique for conflict resolution utilises intelligent backtracking to resolve the inconsistency during failure [11]. Naive backtracking upon failure consists in simply going back to the most recent choice point, removing the corresponding constraints and choosing an alternative solution. This may however not be enough to cure the failure, as the removed constraints may be independent of the previous failure, and this will lead to redoing the same failure and backtrack further.

4.3.1 Failure Analysis

In order to have a better behaviour and avoid useless computation/backtracking steps, one has to determine upon unsolvability of the constraint system the subsystem consisting of the constraints which are the real causes of the failure. Such a subsystem is called a "*conflict set*", and the

[6] If there are at least two constrainers influenced on the same value, say *val*, the restoration of the value *val* from one constraint may be removed again by a new constrainer. The restoration seems "*futile*" (not necessarily).

removal of a single constraint from the conflict set in the original system can "cure" the failure, i.e. restore satisfiability. However, one has to take care of the management of backtrack points as some implicit dependencies between constraints due to previous conflicts and as the constraint system indeed derives from an induced dependency relation tree. In this section, we will present the extensions which are required to achieve such a behaviour.

In the enumeration phase, the inconsistency of the constraint system is discovered as soon as the domain of (at least) one variable becomes empty (we called it a *dead-end*). In order to analyse the failure upon inconsistency of the constraint system, we have adopted a simple strategy based on the information contained in the contradiction explanations. This information should be kept when the domain of the variable is updated during the enumeration phase, and will contain the "trail" of the domain modification. We thus associate to each variable v a revision set Δ_v which contains a set of clashing values (i.e. removed values) with respect to its constrainers. When the domain of some variable becomes empty, the set of the constrainers terms as a *contradiction explanation*[7] will be pushed into the conflict set and the failure analysis procedure is triggered.

Failure analysis mainly relies on the key concepts of *contradiction explanations*. A contradiction explanation is a set of conflict constraints whose conjunction leads to a contradiction. It is a *justification* for the contradiction. Contradiction explanation is learnt from the dead-end [14]. If the dead-end occurs, the constrainer set CS_v becomes the *contradiction explanations* ξ of the variable v.

We observe that at least one constraint has to be retracted from each constradiction explanation to make the current configuration *promising*[8] since any existence of the contradiction explanation ξ will make the configuration *P-contradictory*. Therefore, the removal of some conflicts in the conflict set will give a solvable constraint system, and the conflict hence represents the set of backtracking choice points associated to the failure. For the main purpose here is to retract a *minimal* set of constraints to restore *satisfiability* (not *optimality*), a simple heuristic strategy is used to consider the most constraining constraints in the conflict set ψ. The heuristic approach is greedy since it attempts to minimise the number of updates after each constraint deduction. However, if there is a tie between them, we choose the one with the most recent backtrack point (i.e. the set with the later introduction order) for it leads to more complete procedure than native backtracking (see [11] for a deep analysis of this phenomenon).

[7] It is similar to the terms as *eliminating explanations* in [12] and *nogoods* in [13].

[8] A configuration $\Phi=<AS,US>$ is *promising* iff $\forall\, \xi \in \psi,\ \xi \cap US \neq \phi$ where ψ is a conflict set (see [15]).

Suppose we have a conflict set $\psi = \{\{c_1,c_2,c_4\},\{c_2,c_4\},\{c_1,c_3\}\}$, where the index of c_i denotes the inserting order and the elements of ψ denote contradiction explanations found from the dynamic enumeration. The set $\{c_1,c_2\}$ or $\{c_1,c_4\}$ then turns to be the conflict candidates to be retracted. However, $\{c_1,c_4\}$ is considered (for c_4 is inserted after c_2). Of course, this strategy still leaves some choices unspecified and makes the search incomplete so that it doesn't compute an optimal (minimal) subset. But it does have the effect of localising the conflicts to a subset of constraints easily determined by the dependency information on constraint propagation.

4.3.2 The Evaluation Algorithm Overview

The evaluation algorithm is now described more precisely (as shown in Figure 5). *Min_Conflict_Cns*(ψ) (Line (2)) checks any contradiction explanations in conflict set ψ and returns the minimal conflict candidate set ordered by descending order. *Eliminating_Explanation*(c) (Line (3)) removes the contradiction explanations which contain c from the conflict set ψ. *Retract*(c) (Line (4)) deletes c from the constraint system while maintaining property *P*. *Find_Opponents*(c) (Line (5)) finds out all of the c's opponents retracted before c. If the constraint c is retracted, all of its former opponents in the retracted queue can be "free" and put back to the active store. Since the antagonist is eliminated from the conflict set, the relaxation of its opponents is not necessary any more. Therefore, the *Incremental_Add* (Line (7)) operation is activated again to reset those opponent constraints. The algorithm terminates when the conflict set ψ becomes empty. The experimental performance will be discussed in the following section.

```
Procedure Evaluation(ψ: conflict set)

Begin
[1]  While ψ ≠ φ Do Begin
[2]      Con = Pop (Min_Conflict_Cns(ψ));
//Pick up Con from Minimal conflict set with the most
recent backtrack point
[3]      Eliminating_Explanation(Con, ψ);
//Eliminate the contradiction explanation ξ which
contains constraint Con
[4]      Retract(Con);
[5]      While Find_Opponents(Con, Retracted_Cns)
≠ φ
         Do Begin
[6].         Oc :=Pop(Find_Opponents(Con));
[7]          Incremental_Add(Oc);
//Incrementally re-activate the retracted constraints
which are the opponents of Con
```

Figure 5: The Evaluation Algorithm

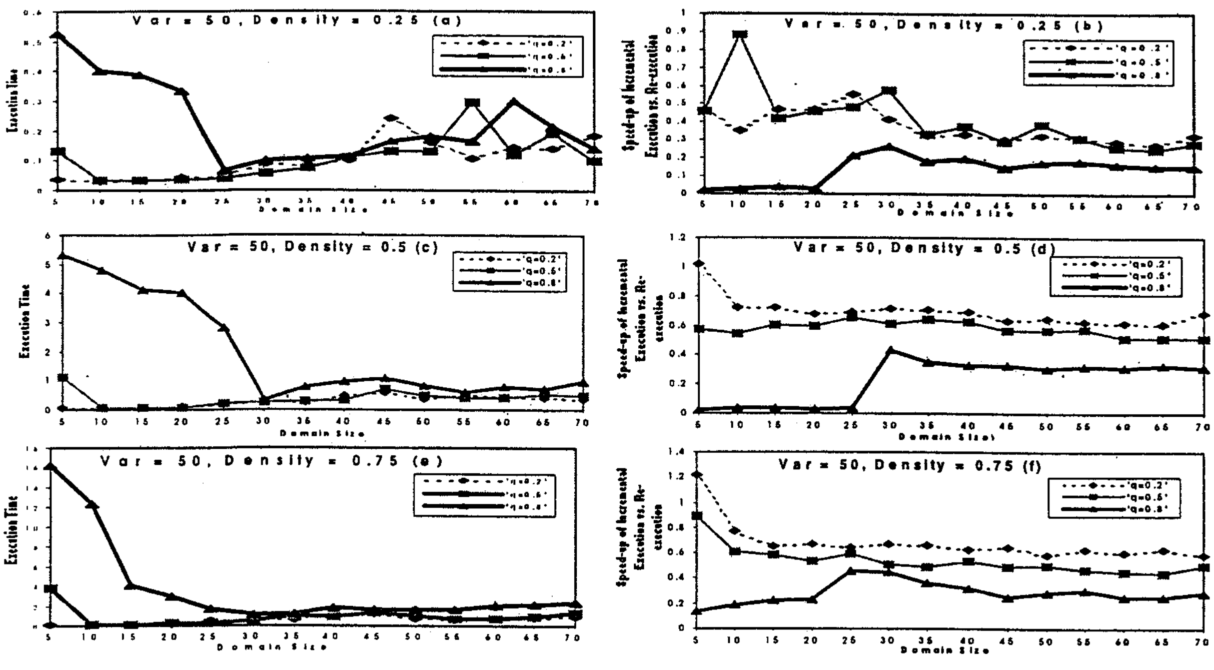

Figure 6: Comparison of Incremental Execution vs. Re-Execution on Random CSPs

5. EXPERIMENTAL RESULTS

5.1 Randomly Generated Problems

Currently, the incremental problem solving has been evaluated on a randomly generated CSP problem where the additions and retractions are performed with a generator of static CSPs. The experiment starts from a *promising* configuration, where all the active constraints are consistent. While the constraint network is not complete and no domain is empty we make an addition. If the local arc-consistency has removed all the values of a domain or if the graph is complete, we proceed to a retraction. The constraints to be retracted are randomly chosen with a uniform probability from the active store. In terms of *efficiency*, two usual measures were performed: *speed-up of constraint checks* for incrementally affected constraints vs. the whole active constraints and *speed-up of computation time* for incremental resolution vs. re-execution. The results presented in Figure 6 are mean values obtained on 500 instances. The set of parameters corresponds to $<n,d,p,q>$, where n is the number of variables, d is the size of variable domain, p is the connectivity of the problem and q is the tightness of the value pairs. Since the performances are very slightly dependent on n, we here fix n to 50, d grows from 5 to 70 with $p \in \{25\%, 50\%, 75\%\}$ and $q \in \{20\%, 50\%, 80\%\}$.

The results in Figure 6 show that the performances are very slightly dependent on the number of constraints but the tightness of the constraint graph. The tighter the problem to be solved, the more propagation of the domain revision will be needed. Therefore, the more percentage of execution time (or constraint checks) will be involved in the evaluation phase (see Figure 6.(a),6.(c) and 6.(e)). On the other hand, there is a *trade-off* between backtracks and constraint propagations. It confirms that it achieves a speed-up for the proposed incremental model w.r.t. the normal re-execution model, especially for over-constrained CSPs (those graphs with higher p and q). However, the approach is degraded for those problems with small tightness due to the unnecessary bookkeeping involved (see Figure 6.(f)). In this case, simple re-computation may be more efficient.

The connectivity of the problem also influences the performance of the relaxation problems whose main advantage is the possibility of not having to consider the whole problem again. For this reason, we want to see at which level of connectivity it is no longer convenient to use the incremental retraction algorithms. The results show that it is worth using the constraint retraction algorithm in almost cases, since the speed up is less than 1 with almost all the cases of our experiments.

5.2 Other Problems

In order to exploit the behaviour of our algorithm, we consider an increasing number of constraints and randomly generated 500 CSPs concerned with 10 variables of domains [1..10]. More precisely, we have generated 50 problems with 10, 20 to 100 constraints. Table 1 shows the results obtained from those problems. All the processed problems are over-constrained. We have noticed that a great majority of the processed problems do not require many number of backtracking for problem resolution. Therefore, we can expect the execution time to be evolved in a polynomial manner. Figure 7 confirms this expectation.

Constraints	RC	BT	PB
10	1.34	1.35	13%
20	3.54	1.94	18%
30	7.56	3.98	25%
40	16.54	3.87	41%
50	19.76	6.55	40%
60	22.23	5.77	37%
70	24.02	4.02	34%
80	29.34	3.66	37%
90	32.54	2.76	36%
100	38.15	2.45	38%

RC: the average number of relaxed constraints
BT: the average number of backtracking
PB: percentage of problems requiring backtracking

Table 1: Average Behaviour of the 10-Variable Problems

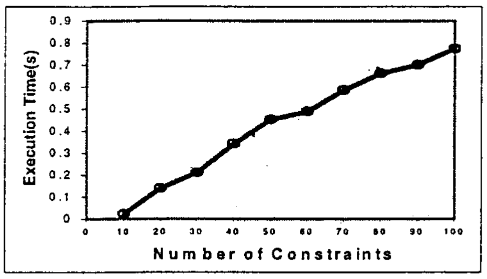

Figure 7: Average Execution Time of the 10-Variable Problems

6. CONCLUSION

We have proposed an incremental problem solving approach on the configuration space that is well-suited for over-constrained problems. This method has been instantiated for finite domain problems. Algorithms and implementations have shown that it is possible to efficiently implement the approach without any slow-down the problem, and in most cases with significantly twice speed-up with respect to a native algorithm which could achieve retraction by re-computing the desired constraints from scratch. Of course, we now need to experiment with more real-life problems to definitely assess our results. However, we believe that the proposed problem solving method may be very convenient for solving large-scale problems, especially for over-constrained problems which allow constraint relaxations in order to obtain a fast and acceptable solution.

REFERENCE

[1] Verfaillie G., and T. Schiex, Dynamic Backtracking for Dynamic Constraint Satisfaction Problems, in *Proceedings of ECAI-94: Workshop on "Constraint Satisfaction Issues Raised by Practical Applications"*, 1994.

[2] Bessiere C., Arc-Consistency in Dynamic Constraint Satisfaction Problems, in *Proceedings of AAAI'91*, pp. 2201-226, 1991.

[3] Debruyne R., Arc-Consistency In Dynamic CSPs Is No More Prohibitive, In *the 8th International Conference on Tools With Artificial Intelligence ICTAI'96*, pp. 299-306, 1996.

[4] Doyle J., A Truth Maintenance System, *Artificial Intelligence*, 12(3), pp. 231-72, 1979.

[5] Codognet P., Diaz D. and R. Rossi, Constraint Relaxation in FD, in, *Lecture Notes of Computer Science*, vol. 1180, 1996.

[6] Menezes F. and P. Barahona, Preliminary Formalisation of an Incremental Hierarchical Constraint Solver, in *Proceeding of EPIA'93*, pp. 281-296, 1993.

[7] Fages F., J. Fowler and T. Sola, A Reactive Constraint Logic Programming Scheme, in *International Conference of Logic Programming, ICLP'95, Tokyo*, 1995.

[8] Fages F., J. Fowler and T. Sola, Experiments in Reactive Constraint Logic Programming, *Journal of Logic Programming*, September, 1996.

[9] Mackworth K., Consistency in Networks of Relations, in *Artificial Intelligence*, 8, pp. 99-118, 1977.

[10] Van Hentenryck P., Y. Deville and C. Teng, A Generic Arc-Consistency Algorithm and its Specialisations, in *Artificial Intelligence*, 57, pp. 291-321, 1992.

[11] Codognet C., P. Codognet and G. File, Yet Another Intelligent Backtracking Method, in *Proceeding of the 5th ICLP*, pp. 447-465, 1988.

[12] Ginsberg L., Dynamic Backtracking, in *Journal of Artificial Intelligence Research*, 1, pp. 25-46, 1993.

[13] Schiex T. and Verfaillie G., Nogood Recording for Static and Dynamic Constraint Satisfaction Problems, *International Journal of Artificial Intelligence Tools*, 3(2), pp. 187-207, 1994.

[14] Frost D. and R. Dechter, Dead-end Driven Learning, in *Proceedings of AAAI'94*, 1994.

[15] Jussien N. and P. Boizumault, Best-First Search for Property Maintenance, in *Reactive Constraints Systems and Logic Programming: the Proceedings of the 1997 International Symposium*, pp. 339-353, 1997.