The Knowledge Engineering Review, Vol. 14:3, 1999, 199–220
Printed in the United Kingdom
Copyright © 1999, Cambridge University Press

Hard, flexible and dynamic constraint satisfaction

IAN MIGUEL and QIANG SHEN

School of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh, EH1 1HN, UK

Abstract

Constraint satisfaction is a fundamental artificial intelligence technique offering a simple yet powerful representation. An increasing amount of attention has recently been paid to the development of constraint satisfaction techniques, and it has become clear that the original formulation of a static Constraint Satisfaction Problem (CSP) with hard, imperative constraints is insufficient to model many real problems. Two important extensions to the classical CSP framework which address some of these deficiencies are flexible and dynamic constraint satisfaction. This paper examines in detail classical, flexible and dynamic CSP. It reviews the motivations behind both extensions, and describes the techniques used to solve each type of problem. The paper employs a running example throughout to illustrate the ideas presented.

1 Introduction

The techniques of constraint satisfaction are ubiquitous in artificial intelligence, mainly due to the simplicity of the structure underlying a constraint-based representation. The classical *Constraint Satisfaction Problem* (CSP) (Dechter, 1992; Mackworth, 1977; Tsang, 1993) involves a fixed set of problem *variables*. Each has an associated *domain* of potential values. A set of *constraints* range over these variables, which specify the allowed combinations of value assignments. To solve a classical CSP, it is necessary to find one or all assignments to all the variables such that all constraints are simultaneously satisfied. Solution techniques for classical CSP are many and varied, encompassing tree search, pre-processing algorithms which work to ease the workload of subsequent tree search, and hybrids which contain both pre-processing and tree search components.

A variety of different problems can, when formulated appropriately, be seen as instances of the classical CSP. Some examples include: machine vision (Boyle & Thomas, 1988; Mackworth, 1977; Montanari, 1974; Rosenfeld et al., 1976; Shapiro & Haralick, 1981; Waltz, 1975); planning and scheduling (Dubois et al., 1995; Fox, 1987; Joslin & Pollack, 1995; Stefik, 1981); belief maintenance (Dechter & Dechter, 1988; Doyle, 1979; de Kleer, 1986; McDermott, 1991); structural design (Guan & Friedrich, 1992); circuit analysis (Stallman & Sussman, 1997); logic programming (Borning et al., 1989; van Hentenryck, 1989; van Hentenryck & Provost, 1991); and systems simulation (Kuipers, 1994; Miguel & Shen, 1998; Shen & Leitch, 1993).

As the techniques of constraint satisfaction have been applied to real-world problems, it has become increasingly clear that classical *hard* constraints (which are imperative and are either fully satisfied or fully violated) are not able to capture the full subtlety of such problems. In light of this, recent approaches have extended the classical CSP framework to create *flexible* constraint satisfaction techniques (Dubois et al., 1996; Freuder & Wallace, 1992). A simple example is the expression of *preferences* among the set of assignments that constitute a solution. A preference is not a hard constraint, since it is ignored if it cannot be satisfied. *Prioritised* constraints are also useful in the context of an unsolvable problem: by removing constraints with the lowest priority, a solution that is still useful may be found.

Classical constraint satisfaction assumes a static problem. However, in many real situations, it is possible for the problem to change as the solution to the original problem is being generated/executed. Consider a large scale scheduling problem, such as scheduling astronomical observations on the Hubble Space Telescope (Minton et al., 1992). Not only is the initial problem extremely difficult to solve (tens of thousands of observations must be scheduled each year, subject to a great variety of constraints), but it also changes continually: new observations may be submitted at any time for scheduling. To address this type of problem, the techniques of *dynamic* constraint satisfaction have been proposed (Dechter & Dechter, 1988; Wallace and Freuder, 1998). Unfortunately, current dynamic constraint satisfaction research is founded almost exclusively on classical CSP, unable to take advantage of flexible constraints in a dynamic environment.

This paper presents a review of the existing approaches to CSP as classified above. The rest of the paper is structured as follows. Section 2 first introduces a running example to illustrate the strengths and weaknesses of the various techniques to be described. It then reviews classical CSP techniques, showing the power of a constraint-based representation. Section 3 reveals some of the inadequacies of the classical formulation and introduces flexible constraint satisfaction. It then discusses how flexible CSP allows greater subtlety in stating and solving constraint satisfaction problems and examines some flexible CSP techniques. Section 4 examines problem solving in a dynamic environment, and introduces the technique of dynamic constraint satisfaction which addresses this area. Finally, section 5 concludes the paper; as a concluding remark, it suggests a possible method for integrating flexible and dynamic constraint satisfaction to create a technique that retains the benefits of both constituent approaches.

2 Classical constraint satisfaction

2.1 A running example

It will be useful throughout this paper to continually refer to a specific example to illustrate the strengths and weaknesses of the techniques described. A series of simple problems will be used consisting of the assignment of a letter to each of several problem variables in order to make a word. This assignment must be made such that constraints between the variables are satisfied. The exact structure of such a problem will be described in tandem with classical CSP itself initially to form a concrete example of the ideas presented.

2.2 Classical CSP definition

A classical constraint satisfaction problem involves a set of n variables, $X = \{x_1, ..., x_n\}$, and a set of constraints, C, over these variables. Each x_i has an associated domain, D_i, describing its potential values. A variable *assignment* is an assignment to a variable, x_i, of one of the values from its associated domain, D_i. Table 1 shows the four variables in the example problem and their associated domains.

Table 1 Variables and associated domains for the example problem

Variable	Domain
x_1	$\{P, K, B\}$
x_2	$\{U, L, I\}$
x_3	$\{V, U, N\}$
x_4	$\{K, J, E, B\}$

Table 2 Constraints for the example problem

Constraint	Definition
$C(x_1, x_2)$	$x_1 < x_2$
$C(x_1, x_3)$	$x_1 < x_3$
$C(x_1, x_4)$	$x_1 < x_4$
$C(x_2, x_3)$	$x_2 < x_3$
$C(x_2, x_4)$	$x_2 < x_4$
$C(x_3, x_4)$	$x_3 > x_4$

A classical (hard) constraint $c(x_i, ..., x_j) \in C$ specifies a subset of the Cartesian product $D_i \times ... \times D_j$, indicating variable assignments that are compatible with each other. These constraints are imperative (each one *must* be satisfied) and inflexible (they are fully satisfied or fully violated). Table 2 shows the constraints that exist in the example problem. They specify a lexicographic ordering such that, for example, $x_1 < x_2$ requires that the assignment to x_1 is strictly lexicographically less than that to x_2. A *solution* to a classical CSP is a complete value assignment to the problem variables satisfying all constraints simultaneously. A CSP may contain several solutions, and the task for a constraint-based problem-solver is to find one or all of these.

2.3 Graphical representation

A CSP can be represented graphically as a *constraint network* (Dechter, 1992). The structure of the network is then used to guide CSP solution techniques. A CSP containing at most binary constraints (i.e., those involving two variables) may be viewed as a *constraint graph*. Each node represents a problem variable, and constraints are represented by labelled, directed arcs. For each $Arc(i, j)$, there exists $Arc(j, i)$, since $c(x_i, x_j) = c(x_j, x_i)$ due to the fact that $D_i \times D_j$ defines the same set of assignments as $D_j \times D_i$. Figure 1 shows a constraint graph representation of the example problem. For clarity, only arcs $Arc(i, j)$ are shown, where $i < j$.

It is possible to represent n-ary constraints via either a *primal* or *dual* graph (Dechter & Pearl, 1989). The former is a generalised constraint graph in that nodes represent variables, and an arc connects any two variables that are related by a constraint. In a dual graph, each node represents a particular constraint and has an associated domain of allowed assignment tuples, i.e., combinations of variable assignments which satisfy the constraint. Arcs connect any two nodes which share a common variable or variables. It is specified that a variable must take the same value at each end of an arc. A dual graph representation of the example problem is shown in Figure 2. Using the dual

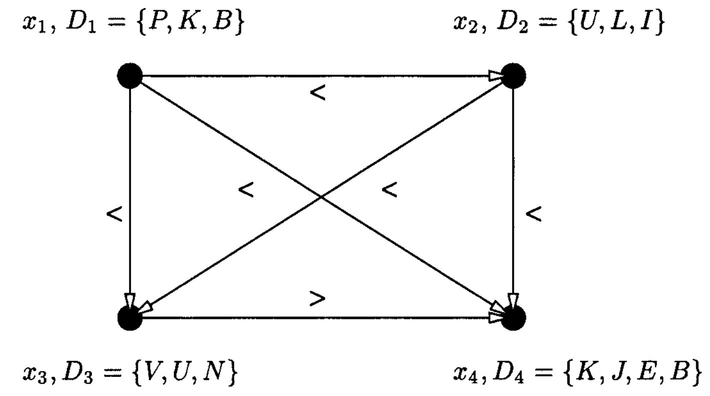
x_1, $D_1 = \{P, K, B\}$
x_2, $D_2 = \{U, L, I\}$
<
<
<
<
<
>
$x_3, D_3 = \{V, U, N\}$
$x_4, D_4 = \{K, J, E, B\}$

Figure 1 Example CSP: constraint graph representation

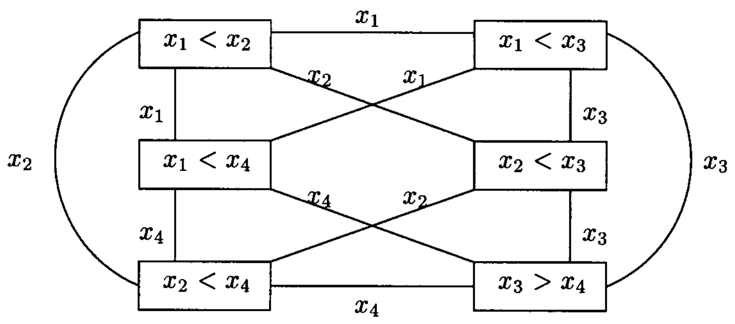

Figure 2 Dual graph representation of example of Figure 1

graph representation, arbitrary n-ary CSP can be solved using techniques developed for binary constraint graphs.

2.4 Tree search

A simple (yet computationally very expensive) approach to solving a CSP is known as *generate and test*. Each possible combination of of variable assignments is generated and subsequently tested to see whether any of the specified constraints are violated. In the worst case, the number of combinations to be generated and tested is the size of the Cartesian product of all variable domains. This brute force attempt will be very time-consuming on problems of any reasonable size.

The search for a solution may be viewed as tree traversal. Figure 3 shows the search tree produced by generate and test in finding the solution $\{x_1 = B, x_2 = I, x_3 = V, x_4 = K\}$ to the example problem. Nodes are annotated with the domain values that were assigned to the current variable at this point in the search. The search path is indicated by the dashed, directed line. For clarity, an unsuccessful assignment to x_4 of each of the elements in D_4 has been replaced by a triangle. Each branch represents a partial assignment. Since this algorithm generates complete assignments before any constraint checks are made, each branch represents a complete assignment in this case. The *i*th level of the tree represents choices made for the *i*th variable in the instantiation order. The term *depth* is used to describe the distance from the root (level 0) which represents the empty set of assignments.

Backtrack is a more intelligent approach to solving a CSP, which is based on the incremental extension of partial solutions (Bitner & Reingold, 1975). The *instantiation order* is the order in which variable assignments are made; it will be assumed herein that it adopts the variable numbering scheme as follows. At a *level*, *i*, of the instantiation order, *past* variables have indices less than *i* and

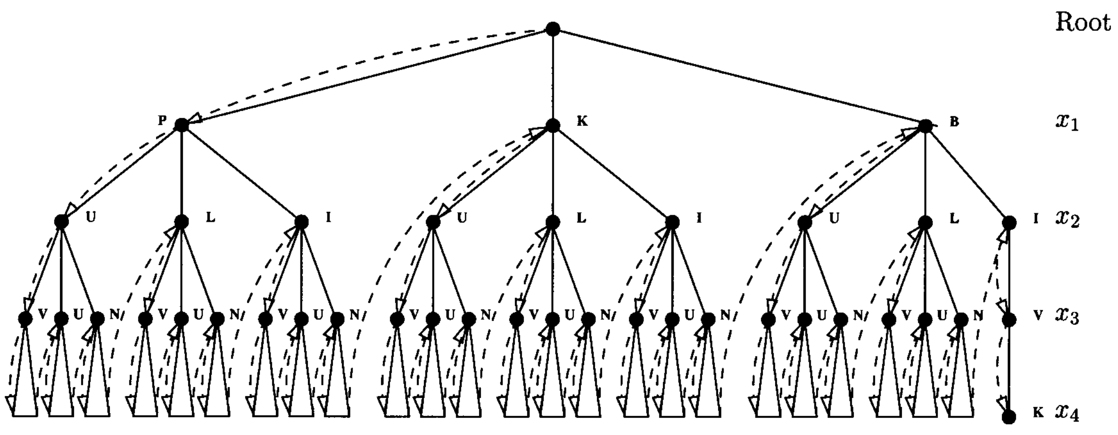

Figure 3 Generate-and-test as a process of tree search

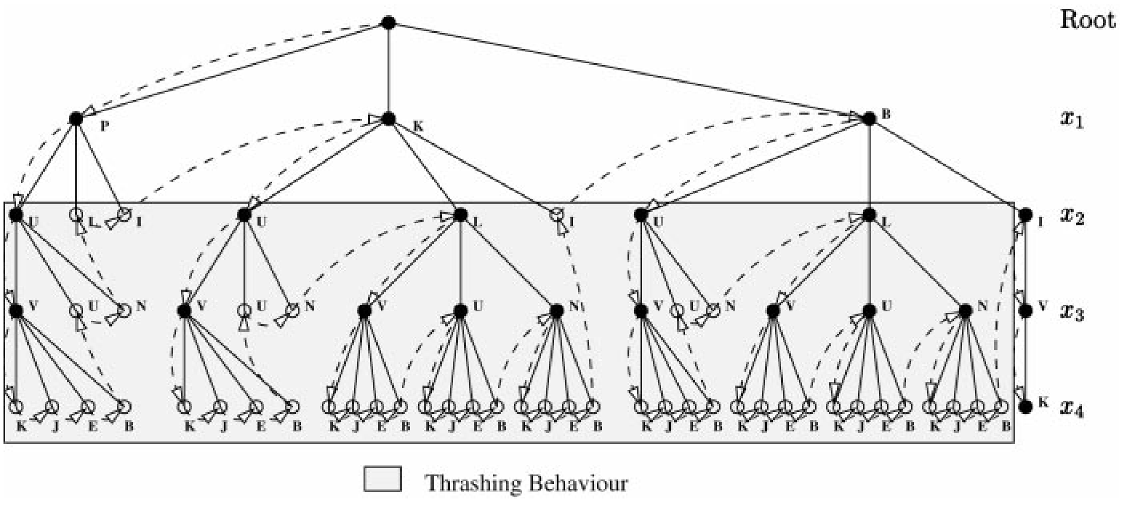

Figure 4 Improvement enabled by Backtrack

have been assigned values. *Future* variables have indices greater than *i* and do not yet have value assignments. Backtrack makes assignments sequentially such that x_i is instantiated to extend a partially complete assignment involving past variables, i.e., $x_1, ..., x_{i-1}$. The constraints dictate the subset of D_i that can be assigned to x_i given previous variable assignments. If there are no possibilities for x_i, then the algorithm *backtracks*, making a new choice for x_{i-1}. If there are no new choices for x_{i-1}, it backtracks until a variable is found with an untested domain element or the algorithm fails.

Figure 4 shows the search tree generated by Backtrack in producing the same solution to the example problem as that found by generate and test. Black nodes represent a successful assignment at this search point, and white nodes represent a failed assignment. The improvement in terms of computational effort can be seen immediately. Backtrack prunes the search space by abandoning attempts as soon as it is clear that the current partial assignment will not yield a solution.

The worst case time complexity of Backtrack is exponential in the number of variables (Mackworth & Freuder, 1985). It is generally inefficient since it usually performs more operations than necessary to find a solution. A major factor in this poor behaviour is that it suffers from *thrashing*: repeated failure of the search process due to the same reason (Gaschnig, 1979). For example: x_g and x_i are related via a constraint such that when x_g is assigned a particular value, the constraint disallows *any* of potential values for x_i. Backtrack fails at level *i* for each element of D_i. Further, this failure will be repeated for every combination of variable assignments to the variables in the intermediate levels, *h*, where $g < h < i$. The degree of thrashing depends upon the number of variables in the intermediate levels, and the size of their associated domains.

Thrashing is present in Backtrack's attempt to solve the example problem (see Figure 4). It is caused here because of $c(x_2, x_4)$: when $x_2 = U$, or $x_2 = L$ there is no element of D_4 that can be assigned to x_4 such that the constraint will be satisfied. This is not discovered by Backtrack until all possible assignment combinations to x_3 and x_4 have been tried.

Several improvements to the core Backtrack algorithm have been proposed to alleviate thrashing. One approach is to "jump" back to the level that disallows any assignment to the current variable. This lifts the restriction that the algorithm must unwind the set of instantiations sequentially, potentially saving much work. The *Backjump* algorithm (Gaschnig, 1979) embodies this approach, which is shown in Figure 5. It recognises that x_2 is the cause of the current conflict and jumps straight back to that variable, avoiding thrashing.

Backjump, however, suffers from only being able to make a single jump: if more backtracking is necessary, it resorts to the original chronological Backtrack. *Conflict-Directed Backjump* (CBJ) (Prosser, 1993) overcomes this problem, enabling multiple backward jumps, by recording the set of

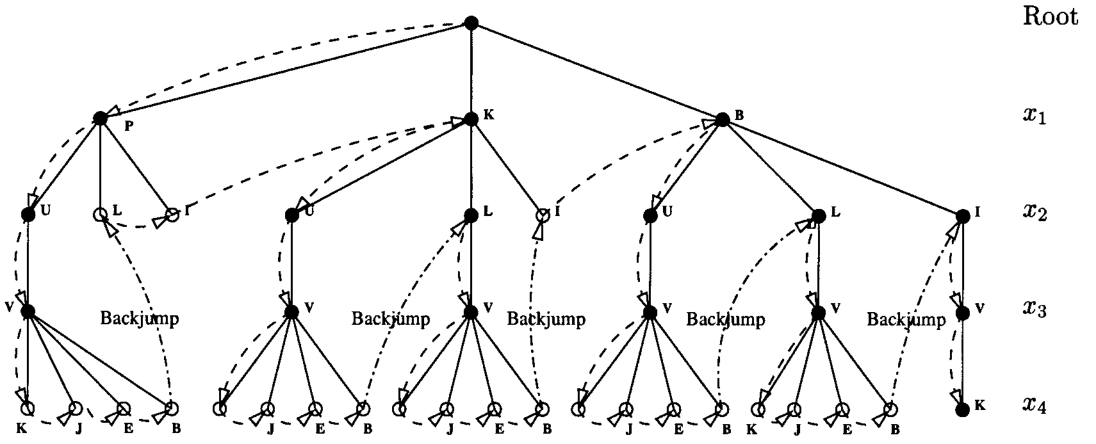

Figure 5 Further improvement enabled by Backjump

past variables (as opposed to just the deepest) that fail consistency checks with the current level. *Dynamic Backtracking* (Ginsberg, 1993) further refines this process by maintaining the assignments of variables which are jumped over but are not involved in the current conflict. This policy does have some drawbacks since, as has been shown (Baker, 1994), it is sometimes desirable to erase the assignments this algorithm maintains.

A second method minimises execution of redundant consistency checks (i.e., checking whether a set of variable assignments satisfy a constraint when the result of this check is already known). The *Backmark* algorithm (Gaschnig, 1979) is founded on this technique, and is used in conjunction with backtracking to make a saving in two ways. First, it avoids consistency checks that must fail: if a variable assignment previously failed against a variable whose instantiation has not changed, then the same assignment will fail again. Secondly, an assignment that succeeded against a set of variables whose instantiations have not changed will satisfy repeated consistency checks against the same past assignments.

Backmark may be combined with more powerful versions of Backtrack to create Backmarkjump (BMJ) and Backmark-Conflict-Directed Backjump (BM-CBJ), respectively (Prosser, 1993). Unfortunately, it is possible for this pair of algorithms to perform more consistency checks than Backmark itself, since the original algorithm was only designed for use with Backtrack. Recognising this problem, these methods have been modified to form the BMJ2 and BM-CBJ2 algorithms (Kondrak & van Beek, 1995), which are designed explicitly to work with a "jumping" method and always perform better than Backmark.

A final method operates when a tree search algorithm reaches a dead end: the current set of assignments to x_1, ..., x_{i-1} is in conflict with x_i. This information is recorded as one or more constraints (or *nogoods*) which aid future search by disallowing the inconsistencies that led to this dead end. Conflict recording can be viewed as a form of learning in that useful information uncovered by the constraint satisfaction procedure is recorded for later use. Extracting all possible information out of a dead end, *deep learning* (Dechter, 1990), is embodied by *Dependency-Directed Backtracking* (Stallman & Sussman, 1977), and is commonly used in *truth-maintenance systems* (e.g., Doyle, 1979). Deep learning involves significant work: since each dead end enables more constraints to be added, which will eventually mean that a large proportion of the search space is recorded. *Shallow learning* (Dechter, 1990) performs only a limited amount of work at each dead end. It uses the constraint graph or, in the case of *jump-back learning* (Frost & Dechter, 1994), the set of variables recorded by CBJ to decide which variables are in conflict with the current level, and hence to form new constraints.

Adding too many new constraints can in fact hinder search because of the increase in constraint checks that must be performed. It is therefore important that any new constraints have as high a probability of being useful to further search as possible. As suggested by Dechter (1990), a good way to accomplish this is to limit the size of any constraints added to the problem.

2.5 Pre-processing techniques

A CSP may be *pre-processed* to create a simpler *equivalent* (i.e., with the same solution set) problem. A tree search algorithm can solve the resulting problem with much less effort than originally required. The underlying mechanism used is *constraint propagation*: a local group of constraints is used to deduce information, which is recorded as changes to the problem. This update forms the basis for further deductions, hence the result of any change is gradually propagated through the entire CSP. Basic pre-processing consists of *filtering* domain elements that cannot take part in any solution from the problem, essentially adding and/or enforcing unary constraints, $c(x_i)$. Two examples of this type of pre-processing are *node consistency* and *arc consistency*, which will be described below.

Node consistency involves any unary constraints, $c(x_i)$, that are specified by a CSP. If for all $x \in D_i$, $c(x_i)$ is satisfied, x_i is *node consistent*. This is enforced by filtering all elements of D_i that do not satisfy $c(x_i)$. $Arc(i,j)$ is *arc consistent* if x_i and x_j are node consistent and for each $x \in D_i$, there exists $y \in D_j$ such that $c(x_i = x, x_j = y)$ is satisfied.

Figure 6 shows how arc consistency is enforced on the two arcs, $Arc(1,4)$ and $Arc(4,1)$ that form the constraint $c(x_1, x_4)$ from the example problem. Figures 6(a) and (b) show $Arc(1,4)$ before and after arc consistency is enforced. Elements P and K are removed since there are no corresponding elements in D_4 such that $c(x_1, x_4)$ could be satisfied. As can be seen, arc consistency is *directional*: the fact that $Arc(1,4)$ is consistent does not mean that $Arc(4,1)$ is also. A further operation is required to enforce consistency on the latter arc, as shown in Figure 6(c). The element B is removed, leaving only elements in D_4 for which corresponding elements in D_1 can be found such that $c(x_1, x_4)$ may be satisfied.

Global arc consistency is obtained by enforcing arc consistency on each arc in the problem. After each $Arc(i,j)$ is made consistent, the other arcs are checked to see which might have been affected by the removal of elements from D_i (since they may rely on a now filtered value in D_i to maintain their own arc consistency). This is simply a matter of finding the arcs which connect other variables to x_i in the constraint graph. Arc consistency must be verified/enforced on all such arcs. Figure 7 shows the example problem after global arc consistency has been enforced. Solutions to this (relatively simple) problem may now be extracted immediately.

Arc consistency is the most popular pre-processing type due to its potential for dramatically reducing the workload of subsequent tree search via relatively modest amounts of computation. An early implementation of an arc consistency algorithm was proposed by Waltz (1975). A generalised version of this algorithm, AC-3, has a worst case time complexity bounded by $O(ed^3)$, where e is the number of constraint arcs, and d is the maximum domain size (i.e., the maximum number of elements in any D_i) (Mackworth, 1977). Improving this bound has been the focus of much research effort. AC-4 (Mohr & Henderson, 1986) has an optimal worst case bound of $O(ed^2)$, whilst AC-5

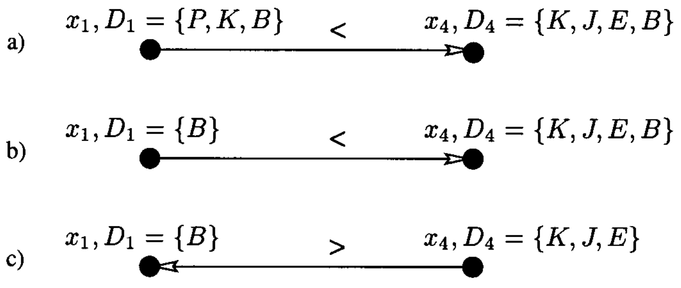

Figure 6 Effect of enforcing arc consistency

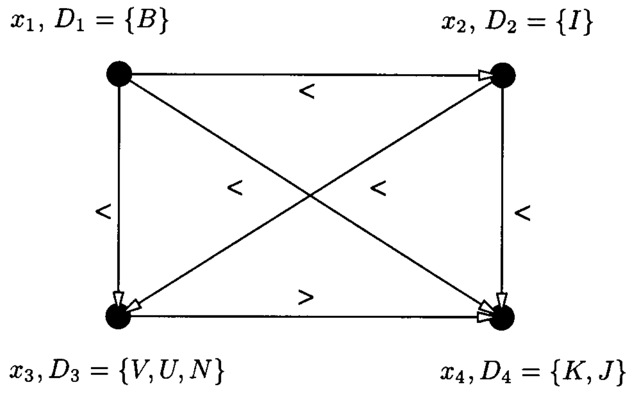

Figure 7 Arc consistent version of problem shown in Figure 1

(Deville & van Hentenryck, 1991) achieves the bound $O(ed)$ for restricted constraint types. AC-6 (Bessiere, 1994) maintains the optimal bound of AC-4, but improves its prohibitive space complexity. AC-7 is a specific implementation of a method for using inference to reduce arc consistency computation (Bessiere et al., 1995). It works by exploiting the bi-directionality property which indicates that arc consistency is satisfied on $Arc(i,j)$ by $x \in D_i, y \in D_j$ if and only if it is also satisfied on $Arc(j, i)$, and which is common to all binary constraints.

More complex pre-processing consists of modifying or adding n-ary constraints to disallow inconsistent assignment combinations (Cooper, 1989; Mackworth, 1977; Mohr & Henderson, 1986). *K-consistency* (Freuder, 1978) is a general theory of consistency levels. A CSP is k-consistent if, given assignments to $k - 1$ variables satisfying all constraints among them, a value may be chosen for any kth variable satisfying constraints among all k variables. *Strong k-consistency* is defined as j-consistency for all $j \leq k$ (Freuder, 1982). Node and arc consistency correspond to strong 1 and 2-consistency respectively.

A lack of 3-consistency (also known as *path* consistency (Han & Lee, 1988; Mackworth, 1977; Mohr & Henderson, 1986) can be seen in the example problem in Figure 1. Consider the assignment $\{x_1 = P,\ x_3 = U\}$, which satisfies the single constraint $c(x_1, x_3)$ between the two variables. There exists no element of D_2 that, when assigned to x_2, can satisfy the constraints amongst all three variables, hence this pair of assignments cannot be part of any solution. This situation is avoided by modifying $c(x_1, x_3)$ such that the assignment pair $\{x_1 = P, x_3 = U\}$ is disallowed, irrespective of the variable instantiation order.

Consistency levels higher than 2-consistency usually involve adding/modifying at least binary constraints, and are significantly more expensive to enforce than node or arc consistency. It is possible to enforce a consistency level where solutions may be found without backtracking (i.e., with *backtrack-free* search (Freuder, 1982)), but this is usually much more expensive than applying Backtrack to the original CSP. To avoid enforcing consistency for constraint arcs that are unnecessary for the search process, *directional* pre-processing algorithms (Dechter & Pearl, 1988) may be used to exploit the instantiation order of subsequent tree search.

2.6 Hybrids

A *hybrid* (Prosser, 1993) embeds a consistency-enforcing component within a tree search algorithm. The motivation is to produce a technique which retains the power of each component approach to create a more powerful algorithm. At each search node, consistency is enforced with respect to current variable assignments. Through further assignments, the problem is divided into simpler sub-problems within which consistency is enforced (Figure 8). Here, the original problem has been split into two sub-problems where the first is committed to the assignment $x_1 = D_{1-1}$, and the second is committed to $x_1 = D_{1-2}$. Backtracking occurs when any variable domain becomes empty as a result

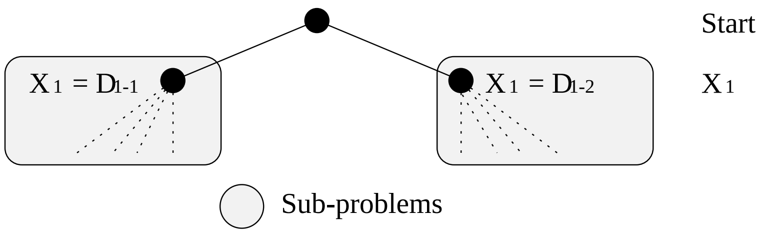

Figure 8 Problem decomposition by a hybrid algorithm

of consistency enforcing based on the current assignments: the sub-problem is inconsistent given these assignments.

A trade-off exists between pruning nodes from the search tree versus performing an increasing amount of work per node. It is usually not cost effective to even apply arc consistency as part of a hybrid algorithm (Dechter & Meiri, 1989), however. The most popular hybrids are those which employ *partial* arc consistency procedures at each search node (Haralick & Elliot, 1980), where a subset of the arcs are made consistent.

2.7 *Heuristics*

Certain operations performed by a constraint-based problem solver, such as variable instantiation and value assignment, may be ordered heuristically to reduce the workload of solving a given problem. A general strategy followed by many CSP heuristics is known as the *fail-first* principle (Haralick & Elliot, 1980). To limit the amount of computation required to solve a problem, it is clearly advantageous to discover early which branches of the search tree cannot lead to a solution.

The instantiation order can have a marked effect on efficiency. Various heuristics exist to choose an order which will have a favourable effect on subsequent search. *Search Rearrangement* (Bitner & Reingold, 1975; Dechter & Meiri, 1989), for example, first assigns the variable with the smallest number of remaining values that are consistent with current assignments to encourage early failure. Generally, this will result in different instantiation orders in the different branches of the search tree.

Figure 9 shows the beneficial effect that this can have on the search for a solution to the example problem. Starting with x_1, the heuristic discovers that D_4 contains the smallest number of values consistent with the assignment $x_1 = P$. In this case there are no consistent values, and a new assignment for x_1 is selected immediately ($x_1 = K$). Again, D_4 contains no values consistent with this assignment, so the assignment $x_1 = B$ is made. At this point, there is no heuristic preference for instantiating any of the remaining three variables next. Hence, x_2 is chosen arbitrarily. After failing

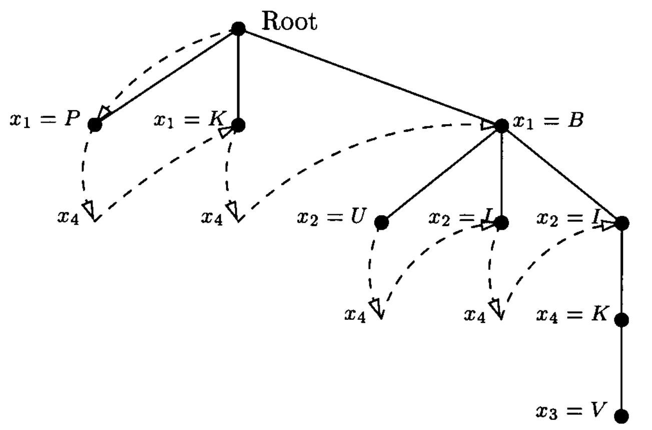

Figure 9 Search rearrangement heuristic applied to the example problem

to find any consistent assignment to x_4 given $x_2 = U$ or $x_2 = L$, the assignment $x_2 = I$ is finally made. The next variable to be assigned is x_4, since D_4 contains the fewest elements compatible with the current assignments. Finally, x_3 is assigned giving the same solution as for uninformed Backtrack, but with a greatly reduced amount of effort.

Other heuristics produce a fixed instantiation order which is a requirement for algorithms such as Backmark. The *maximum-degree* heuristic (Dechter & Meiri, 1989) constructs an order with variables involved in the most constraints placed highest. By assigning the most tightly constrained variables first, the likelihood of discovering conflicts early is increased. The *maximum cardinality* (Dechter & Meiri, 1989) heuristic chooses the first variable at random, then continually picks the variable which is connected to the largest group among the variables already selected, i.e., choosing the most tightly constrained variable with respect to those already assigned.

Value-based heuristics order the assignment of values to a variable. If a solution exists it can be found immediately by picking the "right" assignment at each stage. One heuristic selects the most constrained values first (that appear in the fewest allowed constraint tuples) to encourage early failure (Meseguer, 1989). By contrast, *Lookahead value ordering* (Frost & Dechter, 1995) selects the value that conflicts least with future domain elements in an attempt to maximise the chance of finding a solution on the current search branch. Other heuristics order constraint consistency checks: work is avoided by performing checks that fail early (again following the fail-first principle), for example by choosing the constraint that disallows the most assignments for the current variable (Haralick & Elliot, 1980).

3 Flexible constraint satisfaction

Classical constraint satisfaction techniques deal exclusively with *hard* constraints, which specify absolutely which combinations of assignments to each involved variable are allowed. To obtain a solution to a classical CSP, every constraint must be satisfied. This rigidity of structure has the advantage of simplicity: less information is necessary to describe the problem, which in turn means that problem-solving procedures are simpler. There are some problems inherent in this simple structure. For example, an over-constrained problem (one that admits no solution) requires *relaxation* (some constraints are removed or weakened) before a solution to a less-constrained (but still interesting) problem can be found. Without an indication of the relative importance of each constraint, it is difficult to do this consistently such that a useful solution will be found.

Real problems in general are not easily described in such definite terms as classical CSP requires. It is common for some kind of *flexibility* to be inherent in the problem. A simple example is the expression of *preferences* among the set of assignments, either to the whole problem or local to an individual constraint. *Prioritised* constraints are also useful in the case of an over-constrained problem. If constraints with a lower priority are relaxed it is more likely that the eventual solution will be useful. In addition, for real-time tasks finding flexible solutions may be the only option. Classical techniques find a perfect solution or no solution at all, presenting a serious problem if time runs out for the problem-solver. A flexible system could return the best solution so far, embodying an *anytime* algorithm, as noted by Freuder (1989).

Consider the example shown in Figure 10, which is a prioritised version of the original example of Figure 1. The constraint arcs are annotated with their relative priority. There is an additional imperative (priority 1) constraint $c(x_1, x_2, x_3, x_4)$ that stipulates that the word formed by the sequential assignment to these variable must be a colour. There are two possible words that satisfy this last constraint: *blue* and *pink*, but which constitutes a valid assignment such that all other constraints are satisfied? In fact it is not possible to find a perfect solution to this problem, but it *is* possible to find a "best" solution by selectively relaxing the least important constraints, as will be shown.

Without relative priorities it would not be clear at all which constraint to remove, disallowing a concept of "better" or "worse" solutions. This type of flexibility is useful for real problems, where

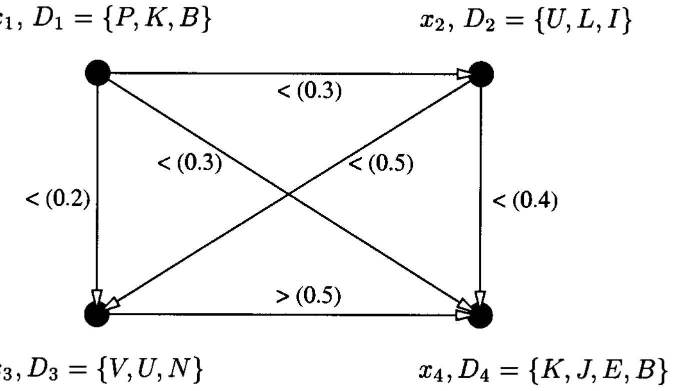
x_1, $D_1 = \{P, K, B\}$
x_2, $D_2 = \{U, L, I\}$
< (0.3)
< (0.3)
< (0.5)
< (0.2)
< (0.4)
> (0.5)
$x_3, D_3 = \{V, U, N\}$
$x_4, D_4 = \{K, J, E, B\}$

Figure 10 An example of flexible CSP

complete constraint satisfaction is often not possible. Flexible constraints are representative of the inherent *softness* found in real problems, and are essential in properly modelling them.

There are several existing extensions to classical CSP which support some types of flexibility (Dubois et al., 1996a; Freuder & Wallace, 1992). The common ground between them is the use of an operator to aggregate the satisfaction level of individual constraints. An overall rating for a complete variable assignment may then be computed and optimised to find the "best" solution to the problem, i.e., the assignment which offers the optimal balance of satisfaction/violation over the entire set of constraints. The interpretation of the rating varies between different approaches. In some cases it is interpreted as counting the cost of any necessary constraint violations, which should be minimised (see Figure 11), while in others it should be maximised to find the best solution to the problem. There are two main methods of combining rating information, as will be described below: *additive* methods (Fox, 1987; Freuder & Wallace, 1992) and *priority maximisation* methods (Dubois et al., 1996; Schiex, 1992).

In order to support such extensions, some modification of classical CSP solution methods must be made. Classical Backtrack can prune a branch of the search tree whenever a constraint is violated. Since flexible CSP tolerates some constraint violation, it is not possible to use simple Backtrack, but the intuition remains the same: a partial solution must be discarded as soon as it becomes clear that all possible extensions to full solutions are unacceptable.

Depth-first branch and bound, as the analogue of classical Backtrack, is a very common technique for solving flexible CSPs (Dubois et al., 1996a; Freuder & Wallace, 1992). The essential idea of this approach is to keep a record of the best solution found so far; this allows the satisfaction procedure to abandon partial assignments, as it becomes obvious that they cannot be extended to full assignments that are better than the current best solution. As the search progresses, the problem solver rates the current partial solution using one of the rating optimisation methods. When the first complete assignment is found, its rating is recorded as α, the *necessary bound* on future solutions.

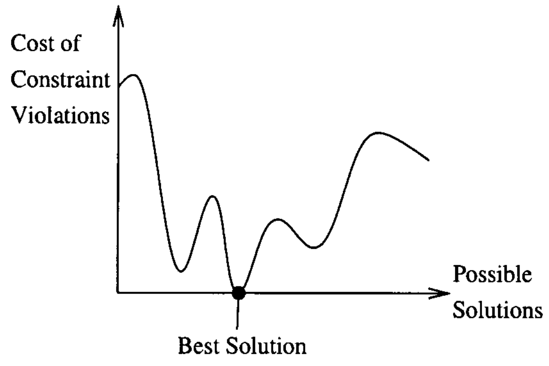

Figure 11 Rating optimisation by cost minimisation for flexible CSP

From this point on, partial assignments are pruned as soon as their rating reaches α. If a complete assignment is found with a better rating than α, α is updated to this new value and the process continues. The algorithm terminates when it is clear that none of the remaining avenues of search can lead to a solution that is better than the current best.

There are two ways to improve basic branch and bound (Freuder & Wallace, 1992). First, α, can be preset based on some prior knowledge about the search space. Hence, more of the space can be pruned. Secondly, a *sufficient bound*, S, can be used so that the algorithm will terminate when a solution of quality equivalent to S or better is found. Again, this allows the algorithm to terminate earlier if possible, searching less of the space.

3.1 Additive methods

Additive methods make use of a summation operator to compute a rating for a complete variable assignment, which is typically used in a branch and bound algorithm to find the optimal problem solution. MAX-CSP (Wallace, 1996) maintains a count of the number of satisfied constraints, which is maximised by an optimal solution. A natural extension to this initial scheme associates a *weight* with each constraint. This value expresses the relative priority or importance of an individual constraint with respect to the rest of the constraints in the problem. The set of constraints C can now be described as a set of pairs, (c_i, w_i), such that w_i is the weight of constraint c_i. An early technique that made use of this type of extension stemmed from the need in machine vision to perform an inexact match between the structural descriptions of two objects (Shapiro & Haralick, 1981). In this case, the weight of a constraint indicates how important a particular part of an object or a relationship amongst a subset of the parts of an object is. A complete assignment, a, in such a system is given a rating based on the following formula, where $V(a)$ is the set of constraints violated by the proposed variable assignment. Hence, the cost of any necessary violations is calculated, indicating the quality of the current assignment.

$$\sum_{c_i \in V(a)} w_i$$

Figure 12 shows how a simple additive solution procedure would solve the problem of Figure 1, if the priorities attached to the constraints are taken as weights to be summed upon violation. The detail of the diagram is restricted to the areas of the search tree surrounding the colour solutions, since any other solution cannot have a rating better (i.e., lower) than 1. This is because the weight 1 quaternary constraint will be violated by any such solution. Each node is annotated with its assignment, and the rating of the current partial assignment in parentheses. The dashed, directed line shows the course of the search.

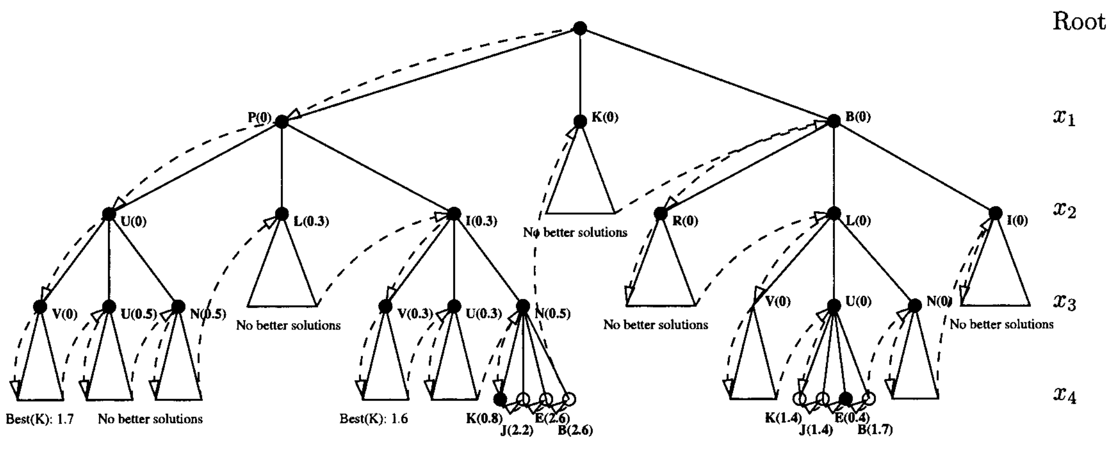

Figure 12 Solution to the problem of Figure 10 using additive methods

The first "colour" solution that is found is *pink*, which violates $Arc(1,2), Arc(1,3)$ and $Arc(1,4)$, and hence has a rating of $0.3 + 0.2 + 0.3 = 0.8$. This rating is now used as a bound to prune the remainder of the search tree whenever the rating of a partial solution exceeds it. The solution *blue*, however, violates only $Arc(2,4)$, earning a rating of 0.4. Hence, $\{x_1 = B, x_2 = L, x_3 = U, x_4 = E\}$ is returned as the optimal solution.

Although suitable for the simple example flexible CSP, the problem with this relatively simple scheme is its coarseness: the computed rating for a complete assignment does not take into account the *level* of satisfaction of individual constraints, i.e., there is no way to express a preference amongst different assignment tuples. A constraint is regarded as either completely satisfied, in which case it will not contribute to the cost, or violated and its weight is added to the cost of this assignment. It is therefore possible for two assignments which differ only in the extent to which they satisfy a particular constraint to be rated as the same. Consider a constraint $c(x_i, x_j)$ which calls for equality between x_i and x_j. Under the above scheme the assignments $\{x_i = 1, x_j = 20\}$ and $\{x_i = 1, x_j = 2\}$ are rated the same (i.e., the constraint is violated), despite the fact that the second pair are much more "equal".

One approach to combining weights and relative assignment preferences to gain a more detailed idea of the quality of a variable assignment is as follows:

$$\sum_{c_i} w_i f_i(a)$$

where f_i is a function which maps from the possible assignment tuples associated with c_i to a satisfaction scale in the range [0, 1] (with 0 denoting complete satisfaction and 1 denoting complete violation of c_i).

Partial Constraint Satisfaction (Freuder, 1989; Freuder & Wallace, 1992) is a general additive system which seeks to partially solve an over-constrained problem by satisfying a maximal number of constraints. A metric is used which is required to evaluate the utility of potential solutions. This metric is not fixed, but may vary from being as simple as a count of the number of violated constraints to being as complex as necessary to capture subtle preferences between solutions. The most general view of partial constraint satisfaction considers a space of problems which are ordered based on the associated set of solutions.

3.2 Priority maximisation methods

Priority maximisation methods attempt to minimise the priority of the most important violated constraint. They work based on the aggregation (generally using a *max* operator) of the satisfaction degrees of the constraints whose weights represent their individual priorities. An advantage of priority maximisation methods is the fact that the *max* operator is idempotent (i.e., for all possible arguments, x, it is the case that $x \otimes x = x$). As several researchers have shown (Dubois et al., 1996a; Schiex et al., 1995), this allows many techniques developed for classical CSP to be extended to this type of flexible CSP framework in a straightforward fashion. This is generally not the case for additive methods (Freuder & Wallace, 1992; Schiex et al., 1995).

3.2.1 Necessity-based constraints

The *necessity*-based flexible CSP system (Schiex, 1992) makes use of possibilistic logic (Lang, 1991) to model the gradual satisfaction of constraints. A possibility distribution, over the set of all possible complete variable assignments, A, is a function π from A to [0, 1]. This distribution supports a gradual notion of the consistency of a given assignment, encapsulating preferences among the set of assignments.

A distribution, π, induces the measures of *possibility*, Π_π, and *necessity*, N_π, over the set of constraints. The possibility, $\Pi_\pi(c_i)$ represents the possibility of constraint c_i being satisfied, given π. The necessity measure, $N_\pi(c_i)$, represents the extent to which the satisfaction of constraint c_i is entailed given π. The necessity of a constraint tends to 1 as the possibility of it being unsatisfied tends to 0. The definition of both measures follows (adapted from Schiex (1992)):

$$\begin{aligned}\Pi_\pi(c_i) &= Sup(\{\pi(a), a \models c_i\} \cup \{0\})\\ N_\pi(c_i) &= Inf(\{1-\pi(a), a \models \neg c_i\} \cup \{1\})\\ &= 1 - \Pi_\pi(\neg c_i)\end{aligned}$$

where $a \in A$ is a complete variable assignment; $\neg c_i$ is the complement of c_i: the subset of the Cartesian product of the domains of the constrained variables that does *not* satisfy c_i; and $a \models c_i$ signifies that assignment a satisfies c_i.

Possibilistic CSP introduces necessity-valued constraints to replace the original hard constraint type. The necessity value expresses the relative preference of satisfaction of each constraint, as per Figure 10. The set of constraints, C, is represented by the set of pairs, (c_i, α_i), with $\alpha_i \leq N_\pi(c_i) \leq 1$. Hard constraints can also be represented via $(c_i, 1)$. The measure of necessity depends upon the particular possibility distribution π. Hence, it is important to choose a distribution, π^*, which induces the required necessity on each constraint (Schiex, 1992).

Possibilistic CSPs are consistent to a *degree* rather than consistent or inconsistent as per classical CSPs. The degree of consistency of a possibilistic CSP is the maximum value assigned by π^* to an assignment $a \in A$, where A is the set of all possible complete variable assignments. The set of "best" assignments A^* for a particular problem contains those that receive this highest value assignment. A branch and bound algorithm is used to find such a best assignment. The upper bound on the rating of a complete assignment, β, may be computed by taking the lowest value assigned by π given the constraints applicable to the currently instantiated variables. This bound may only remain the same or decrease in the light of future assignments.

In summary, this system offers a possibilistic framework to support relative constraint priorities, but does not immediately support constraint assignment preferences. The basic branch and bound solution technique has been extended to use a form of the hybrid *forward checking* algorithm (Stallman & Sussman, 1977) to increase efficiency, and a method for achieving arc consistency is also defined (Schiex, 1992).

3.2.2 The fuzzy constraint satisfaction framework

The Fuzzy Constraint Satisfaction framework (Dubois et al., 1996a; Pires & Prade, 1998) supports both prioritised and preference constraints, again using possibility theory (Dubois & Prade, 1988). Both types of constraint are modelled by a fuzzy relation, R, which is in turn defined by μ_R, a *membership function* which associates a level of satisfaction of an assignment tuple in $D_1 \times ... \times D_n$ with a value in a totally ordered *satisfaction scale*, $L = [0, 1]$. These flexible constraints are used to order partial assignments in order to find an overall best complete assignment.

A preference constraint, c_i, amongst a set, A, of potential assignments to its constrained variables can be modelled as follows, where a, $a \in A$, is a particular assignment to the constrained variables. The membership function μ_{R_i} of a satisfying constraint c_i is defined on L:

$$\begin{aligned}\mu_{R_i}(a) &= 1 \text{ if } a \text{ totally satisfies } c_i\\ \mu_{R_i}(a) &= 0 \text{ if } a \text{ totally violates } c_i\\ 0 < \mu_{R_i}(a) &< 1 \text{ if } a \text{ partially satisfies } c_i\end{aligned}$$

Prioritised constraints are dealt with slightly differently. Another scale V (effectively L in reverse) is used, representing *possibility of violation* or equivalently the priority of a constraint. A priority degree, $Pr(c_j) \in V$, is associated with each prioritised constraint, c_j, indicating the necessity of its satisfaction. A priority of 1 indicates a completely imperative constraint (with no possibility of violation), and a priority of 0 indicates a totally irrelevant constraint. Clearly, the constraints of the example of Figure 10 are represented as prioritised constraints in this framework.

The bijection $b(\cdot)$ is used to map from V to L such that $L = b(V)$. The quantity $b(Pr(c_j)) \in L$ indicates the satisfaction level of the complement of c_j. Therefore, a prioritised constraint c_j with

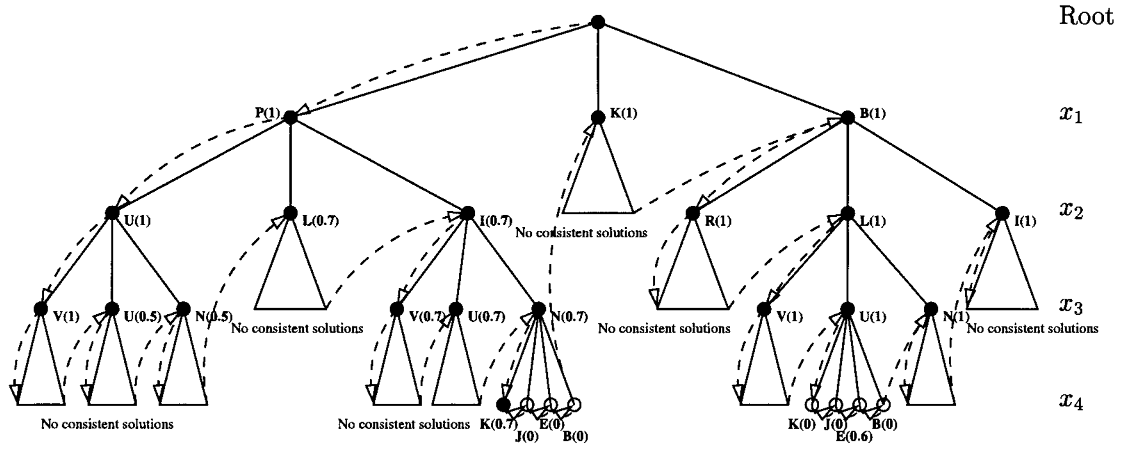

Figure 13 Solution to the problem of Figure 10 using fuzzy constraint satisfaction

priority $Pr(c_j)$ is modelled as follows: it is rated as fully satisfied by an assignment if c_j is satisfied and to a degree of $b(Pr(c_j))$ if an assignment does not satisfy c_j.

$$\mu_{R_j}(a) = 1 \text{ if } a \text{ satisfies } c_j$$
$$\mu_{R_j}(a) = b(Pr(c_j)) \text{ if } a \text{ violates } c_j$$

A constraint which is prioritised *and* expresses a preference amongst assignments is defined via a fuzzy relation, R_k, with μ_{R_i} describing the preference component. Hence a constraint c_k, represented by R_k will be satisfied to at least the degree $b(Pr(c_k))$ since the priority degree defines a bound on the damage to the solution that is incurred by the violation of this constraint. The constraint may be satisfied above this level by satisfying the preference component to a higher degree.

$$\mu_{R_k}(a) = max(b(Pr(c_k)), \mu_{R_i}(a))$$

A fuzzy CSP involves a set of flexible constraints of the types described above over a set of n variables. The consistency level of a partial assignment is calculated from the aggregated membership values of the constraints involving the subset of the variables which currently have values assigned. This is used as the basis of a depth-first branch and bound search to find the best solution to the problem. The standard consistency level calculation provides quite a coarse definition of the best solution to the problem, since only the most violated constraint is used. Approaches exist which discriminate amongst solutions with the same basic consistency degree at a greater computational cost (Dubois et al., 1996b; Meseguer & Larrosa, 1997).

Figure 13 shows the search tree generated by Fuzzy Constraint Satisfaction in solving the problem of Figure 10. The difference to the additive methods is clear; instead of accumulating a cost through constraint violation, this method attempts to minimise the priority of the most important violated constraint. Hence, the algorithm attempts to maximise the rating of the overall solution. Again, *pink* is the first colour solution found, with a rating of 0.7. This algorithm regards *pink* as a better solution than *blue* (rating 0.6) since the constraints it violates have a priority of 0.3 or less.

3.3 Solution via graph decomposition

A general flexible solution method, based on dynamic programming (Bertele & Brioschi, 1972), which exploits graph decomposition is shown in Beale (1997). The constraint graph representing the whole problem is decomposed into subgraphs for which solutions are sought. An optimal solution is sought with respect to each possible value for the nodes which connect a subgraph to the rest of the graph. Solution synthesis then combines subgraphs via the connecting nodes to generate a solution to the whole problem.

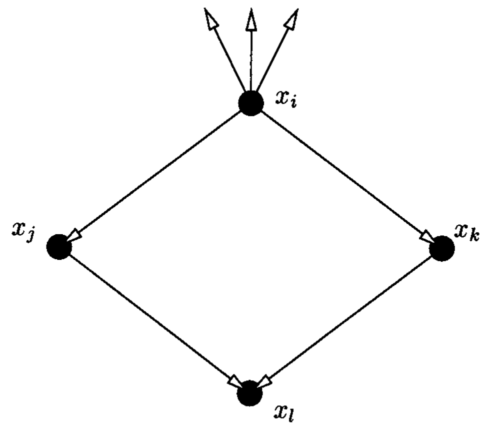

Figure 14 Subgraph. Optimise with respect to assignments to x_i (Adapted from Beale (1997))

An example is shown in Figure 14. Here, assignments to x_j, x_k, x_l are optimised with respect to each possible assignment from D_i to x_i which connects this subgraph to the rest of the graph. Depending on the assignment made to x_i as a result of solution synthesis, the optimal assignments to the other three variables are immediately available.

4 Dynamic constraint satisfaction

A combination of classical and flexible constraints allow many problems to be accurately modelled. However, one important aspect of many real problems has yet to be addressed. So far a *static* problem has been assumed, precluding any changes to the problem structure after its initial specification. Many problems, however, undergo changes, sometimes continually, which may occur while the solution to the original problem is being executed or, perhaps more critically, while a solution to the original problem is still being sought. The ability to reason about a dynamic environment is crucial in many areas of artificial intelligence. Unfortunately, the majority of CSP research has focused on static problems, i.e., the set of constraints and problem variables are known beforehand and fixed. However, there are techniques for solving *Dynamic Constraint Satisfaction Problems* (DCSPs), with two alternative formulations that make slightly different assumptions about the nature of the dynamic problems involved.

4.1 Dynamic constraint satisfaction problems

The first formulation (Dechter & Dechter, 1988), known simply as DCSP, views a dynamic environment as a sequence of CSPs linked by *restrictions* and *relaxations*, where constraints are added to and removed from the problem respectively (Figure 15). Note that the addition or removal of variable domain elements is effectively the addition or removal of unary constraints to the problem. Naively, each individual problem in the sequence may be solved from scratch using static CSP techniques, but this method discards all the work done in solving the previous (probably similar) problem. This wasted effort is usually unacceptable in real-time applications that can

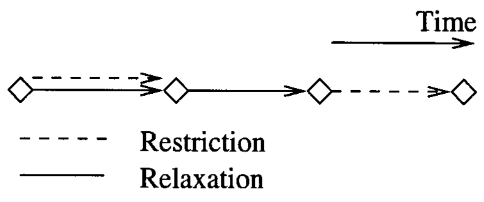

Figure 15 An evolving problem modelled via DCSP

experience rapid changes in the problem. The key to efficiently solving DCSPs is to re-use as much as possible of the effort required to solve previous problems in the sequence in solving the current problem.

The existing approaches to dynamic constraint satisfaction may be broadly classed into three categories. The first method (van Hentenryck & Provost, 1991) attempts to solve each problem in the sequence from scratch (i.e., an empty set of assignments) using a static CSP algorithm, but uses assignments from the solution to the previous problem to guide the current attempt. When considering an assignment for each variable, it tries the assignment used previously first.

A second method, known as *local repair* (Minton et al., 1992; Verfaillie & Schiex, 1994), maintains all assignments from the solution to the previous problem in the sequence to use as a starting point. This initial variable assignment is then progressively modified by a sequence of modifications to individual variable assignments until an acceptable solution to the current problem is obtained. Modifications take the form of element reassignment to resolve as many conflicts (i.e., constraint violations) involving this variable as possible. The *heuristic repair* algorithm (Minton et al., 1992) is a local repair algorithm which searches through the space of possible repairs using the heuristic that there should be as few constraint violations as possible at each stage to guide the search.

The *local changes* algorithm (Verfaillie & Schiex, 1994) maintains a list of currently unassigned variables. When the problem changes in a DCSP this would consist of any new variables added to the problem or variables whose current assignment has been lost. This repair algorithm repeatedly selects a variable from the set that are unassigned and assigns a value to it. If the assignment causes some constraint violations, local changes attempts to repair the current set of assignments to resolve the inconsistencies. Repairs consist of unassigning the subset of the assigned variables that is perceived to have caused the inconsistency, and attempting to reassign values to them such that all constraints are satisfied. The local changes algorithm obtains an improvement over heuristic repair by avoiding the repair of variables which have no bearing on the existing inconsistencies, and is therefore related to static CSP algorithms such as Backjump.

Finally, constraint recording methods (Schiex & Verfaillie, 1993; van Hentenryck & Provost, 1991) infer new constraints from the existing problem definition which disallow inconsistent assignment combinations not directly disallowed by the original constraints. The justifications of inferred constraints are recorded in order that they can be used in future problems where the same justifications hold to converge on a solution more quickly. In fact, the local changes algorithm can be improved through the use of such a scheme (Verfaillie & Schiex, 1994). Again the same caveat applies concerning the extra work incurred by checking many extra constraints.

4.1.1 A DCSP example

Returning to the original CSP as shown in Figure 1, consider the effect of changing this problem as shown in Figure 16. Constraints $c(x_2, x_3)$, $c(x_3, x_4)$ have been removed and the variables x_4, x_5 have

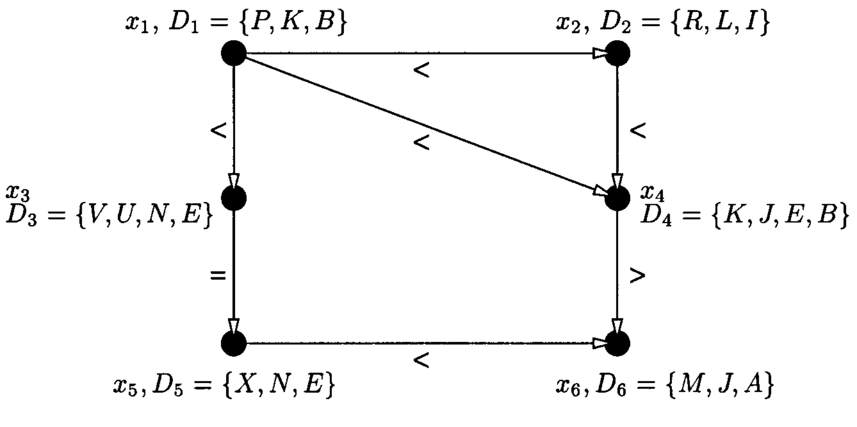

Figure 16 A DCSP example, updated from Figure 1

Initial: $x_1 = B, x_2 = I, x_3 = V, x_4 = K$ (consistent)
$V_1 = \{\}, V_2 = \{x_1, x_2, x_3, x_4\}, V_3 = \{x_5, x_6\}$
Select $x_5 \in V_3$. Consider $c(x_3, x_5)$
$x_5 \leftarrow X$. Violates $c(x_3, x_5)$: Choose x_3
$V_1 = \{x_5\}, V_2 = \{x_1, x_2, x_4\}, V_3 = \{x_3\}$

Select $x_3 \in V_3$. Consider $c(x_1, x_3), c(x_3, x_5)$
All assignments violate $c(x_3, x_5)$. x_5 is fixed. Fail.

$x_5 \leftarrow N$. Violates $c(x_3, x_5)$: Choose x_3
$V_1 = \{x_5\}, V_2 = \{x_1, x_2, x_4\}, V_3 = \{x_3\}$

Select $x_3 \in V_3$. Consider $c(x_1, x_3), c(x_3, x_5)$
$x_3 \leftarrow V, x_3 \leftarrow U$ violate $c(x_3, x_5)$. x_5 is fixed.
$x_3 \leftarrow N$ succeeds.
$V_1 = \{\}, V_2 = \{x_1, x_2, x_3, x_4, x_5\}, V_3 = \{x_6\}$

Select $x_6 \in V_3$. Consider $c(x_5, x_6), c(x_4, x_6)$
$x_6 \leftarrow M$. Violates $c(x_5, x_6), c(x_4, x_6)$: Choose x_4, x_5
$V_1 = \{x_6\}, V_2 = \{x_1, x_2, x_3\}, V_3 = \{x_4, x_5\}$

Select $x_4 \in V_3$. Consider $c(x_1, x_4), c(x_2, x_4), c(x_4, x_6)$
All assignments violate $c(x_4, x_6)$, x_6 is fixed. Fail.

$x_6 \leftarrow J$. Violates $c(x_5, x_6)$. Choose x_5
$V_1 = \{x_6\}, V_2 = \{x_1, x_2, x_3, x_4\}, V_3 = \{x_5\}$

Select $x_5 \in V_3$. Consider $c(x_3, x_5), c(x_5, x_6)$
$x_5 \leftarrow X, x_5 \leftarrow N$ violate $c(x_5, x_6)$. x_6 is fixed
$x_5 \leftarrow E$ violates $c(x_3, x_5)$. Choose x_3
$V_1 = \{x_5, x_6\}, V_2 = \{x_1, x_2, x_4\}, V_3 = \{x_3\}$

Select $x_3 \in V_3$. Consider $c(x_1, x_3), c(x_3, x_5)$
$x_3 \leftarrow V, x_3 \leftarrow U, x_3 \leftarrow N$ violate $c(x_3, x_5)$. x_5 is fixed
$x_3 \leftarrow E$ succeeds.
Solution: $x_1 = B, x_2 = I, x_3 = E, x_4 = K, x_5 = E, x_6 = J$

Figure 17 Solution to the problem shown in Figure 16 via local changes

been added with constraints connecting them to each other and the rest of the problem. Lastly, the element E has been added to D_3. Clearly, some more work is required to solve this new problem. The naive reaction is to simply apply the original static CSP solution procedure (e.g., Backtrack) to the whole updated problem. This is wasteful of the work done on the original (similar) problem.

As an example of the savings that can be made using a DCSP technique, it is useful to examine how the local changes repair algorithm would go about solving this new problem on the basis of the original solution: $\{x_1 = B, x_2 = I, x_3 = V, x_4 = K\}$. Figure 17 shows the solution procedure, where V_1 is the set of variables whose assignments are fixed. In addition, V_2 is the set of variables which have assignments but that are not fixed, and V_3 is the set of unassigned variables. The algorithm terminates with the solution: $\{x_1 = B, x_2 = I, x_3 = E, x_4 = K, x_5 = E, x_6 = J\}$, having done just a fraction of the work that would be required to solve the problem from scratch.

Through making use of the previous solution in the sequence, there is a significant efficiency saving made by avoiding re-solving a large part of the problem. This simple example serves as an indication of the saving that can be made generally by maintaining as much of the effort expended on the previous solution as possible. In addition to efficiency, this type of technique offers the benefits of *stability* to more complex problems. That is, by maintaining as much of the previous solution as possible, there is as little disruption from one solution to the next as possible. A DCSP technique is much more likely to achieve a reasonable level of stability than a static CSP technique applied to the new problem, since the latter essentially throws away all the work done on the previous solution.

4.2 *Recurrent dynamic constraint satisfaction problems*

A second DCSP formulation was developed more recently, and is known as *recurrent* DCSP (Wallace & Freuder, 1998). This technique is founded on the fact that many alterations to a problem may be temporary and that these changes may be subsequently reversed. Recurrent DCSP concentrates on changes to the set of constraints that could potentially render the current solution invalid. These changes include the addition of unary constraints (domain element removal), the addition of n-ary constraints, and the loss of some acceptable tuples in existing constraints. The aim of recurrent DCSP is to find solutions that are not "broken" by such changes. A good solution to recurrent DCSP is therefore stable in a slightly different sense: it remains acceptable in light of alterations to the problem.

An important point to note is that all the changes supported by recurrent DCSP result in the temporary loss of one or more domain elements as potential members of acceptable solutions. In light of this, a probability of loss may be associated with each domain element. A stable solution may then make use of a set of assignments with as low as probability of loss as possible. Since it is unlikely in general that individual loss probabilities are known beforehand, this information is extracted dynamically by recording the frequency of loss of each value as the problem evolves.

Recurrent DCSP solution techniques have been developed using a local repair technique which uses hill-climbing, taking into account the loss probabilities to maximise stability. Although hill-climbing does not guarantee that an optimal solution will be found, this is a moot point since the loss probabilities are only estimates. The principle argument in favour of hill-climbing is that there is no restriction on the timing of changes to the problem. Hence, if the problem changes while a complete method is in the process of solving it, one of the assigned domain elements may be lost, forcing the retraction of any assignments after this point. Further, stability calculations may be rendered invalid, removing the guarantee of finding an optimally stable solution. Hill-climbing methods do not suffer from these problems. In the face of change, hill-climbing can proceed as normal, pausing only to reassign variables whose assigned elements have been removed.

This solution technique has been tested on graph-colouring and random CSPs (Wallace & Freuder, 1998). It was found that repairing a partial solution not only increases efficiency in finding a solution to the current problem, but is also effective in finding stable solutions (i.e., those likely to remain valid in the face of change). The technique of dynamically acquiring loss probabilities for individual domain elements was particularly successful in leading to stable solutions, particularly when there were very few stable solutions to be found.

5 Conclusion

This paper has examined the Constraint Satisfaction Problem (CSP) in both the classical (hard and static) formulation and in the light of flexible and dynamic extensions to the original framework. Returning to classical CSP, it was shown that Backtrack can solve any such problem, but at significant computational cost. Hence, various improvements have been made to this core algorithm which attempt to avoid both dead ends in the search space and performing redundant constraint consistency checks. In addition, the beneficial effects of pre-processing a CSP prior to tree search on the workload of the algorithm subsequently used to solve the problem were examined. Hybrid algorithms were introduced which embed such pre-processing techniques into tree search so that pre-processing is done at every search node in an attempt to combine the benefits of both approaches.

Classical CSP was seen to be insufficient to solve many real problems. The first major deficiency is in its rigidity of structure: it is common in reality for constraints to contain some degree of flexibility. Flexible CSP was shown as an extension of the classical framework which supports several types of 'soft' constraint. The second deficiency of classical CSP is in its assumption of a static problem, whereas many problems are subject to change over time. Dynamic CSP was presented as a means of allowing a CSP to change through constraint restriction and relaxation.

It may have been noticed, however, that the previous section made no mention at all of flexible CSP. It would of course be very useful to have a system which incorporates the ability to support flexible constraints and dynamic problems. If the problem shown in Figure 10 was to change, it would be necessary to consider which flexible constraints to relax to find the optimal solution to the new problem. Unfortunately, little work has been done in this area. Current research into dynamic constraint satisfaction is founded almost exclusively on classical hard constraints. Hence, it is not possible to take advantage of flexible constraints in a dynamic environment. A clear target, therefore, is to integrate two disparate extensions to classical CSP techniques, dynamic and flexible constraint satisfaction, to produce a system capable of dealing with a greater number of real problems than is currently possible (see Figure 18).

5.1 Dynamic flexible constraint satisfaction

Dynamic flexible constraint satisfaction must be able to model the changes in a dynamic environment, while retaining the greater expressive power afforded by flexible CSP. In a dynamic

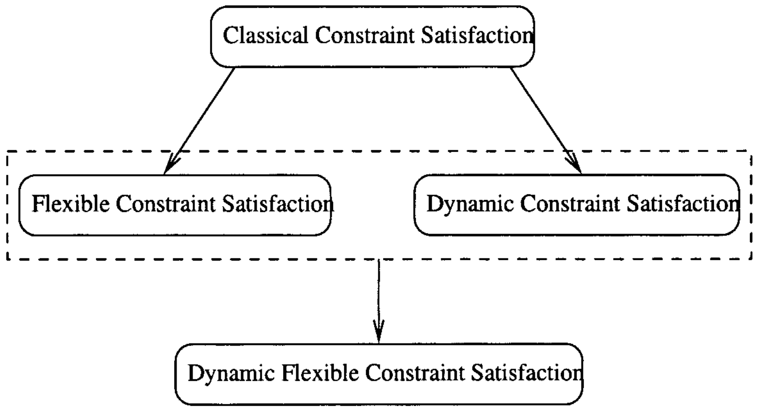
Classical Constraint Satisfaction
Flexible Constraint Satisfaction
Dynamic Constraint Satisfaction
Dynamic Flexible Constraint Satisfaction

Figure 18 Integration of disparate extensions of classical CSP

environment where time may be limited, the ability of flexible CSP to produce the best current solution *any time* and the use of a sufficiency bound to limit search will prove even more valuable.

A DFCSP may be thought of as a sequence of static flexible problems similar to the original hard DCSP definition (see Figure 15). All possible changes may still be realised through restriction and relaxation. However, since flexible CSP allows more powerful constraint types, these two operations can accordingly effect more subtle changes to the problem. A dynamic flexible problem will also evolve, for example, through changes to the priorities of individual constraints or to the balance of preferences assigned to individual constraint tuples.

An important step in creating this new technique is to choose dynamic and flexible components which are both powerful in their own right and complementary when integrated. As perhaps the most successful DCSP technique in the literature (Verfaillie & Schiex, 1994; Wallace & Freuder, 1998), local repair appears to be a good candidate for the dynamic component. Given a graded level of solution consistency, this approach could be implemented as either a hill-climbing (which would protect against changes in the problem during the solution period) or a complete method which would guarantee finding the optimal solution to each problem in the sequence. The flexible component may be implemented in a variety of ways, the *Partial Constraint Satisfaction* (Freuder & Wallace, 1992) and *Fuzzy Constraint Satisfaction* (Dubois et al., 1996a) frameworks are examples which offer a range of flexible constraint types. In addition, Beale (1997) suggests an extension to the graph decomposition algorithm described in section 3.3 to support dynamic problems. The potential applications of DFCSP are wide, encompassing many dynamic problems that require more flexibility than classical CSP can provide. Some examples include planning and scheduling, machine vision, model-based reasoning, and game-playing. Work on the integration of dynamic and flexible CSP is being carried out by the authors.

References

Baker, AB, 1994, "The hazards of fancy backtracking" *Proceedings of the Twelfth National Conference on Artificial Intelligence* pp 288–293.

Beale, S, 1997, "Using branch-and-bound with constraint satisfaction in optimization problems" *Proc. of AAAI-97* pp 209–214.

Bertele, U and Brioschi, F, 1972, *Nonserial Dynamic Programming* Academic Press.

Bessiere, C, 1994, "Arc-consistency and arc-consistency again" *Artificial Intelligence* **65** 179–190.

Bessiere, C, Freuder, EC and Regin, J, 1995, "Using inference to reduce arc consistency computation" *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* pp 592–598.

Bitner, R and Reingold, M, 1975, "Backtrack programming techniques" *Communications of the ACM* **18**(11).

Borning, A, Maher, M, Martindale, A and Wilson, M, 1989, "Constraint hierarchies and logic programming" *Proceedings of the Sixth International Conference on Logic Programming* pp 149–164.

Boyle, RD and Thomas, RC, 1988, *Computer Vision: A First Course* Blackwell Scientific.

Cooper, MC, 1989, "An optimal k-consistency algorithm" *Artificial Intelligence* **41** 89–95.

de Kleer, J, 1986, "An assumption-based tms" *Artificial Intelligence* **28** 127–162.

Dechter, R, 1990, "Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition" *Artificial Intelligence* **41** 273–312.

Dechter, R, 1992, "Constraint networks" In: *Encyclopedia of Artificial Intelligence* Wiley, pp 276–285.

Dechter, R and Dechter, A, 1988, "Belief maintenance in dynamic constraint networks" *Proceedings of the Ninth National Conference on Artificial Intelligence* pp 37–42.

Dechter, R and Meiri, I, 1989, "Experimental evaluation of preprocessing techniques in constraint satisfaction problems" *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* pp 271–277.

Dechter, R and Pearl, J, 1988, "Network-based heuristics for constraint-satisfaction problems" *Artificial Intelligence* **34** 1–38.

Dechter, R and Pearl, J, 1989, "Tree clustering for constraint networks" *Artificial Intelligence* **38** 353–366.

Deville, Y and van Hentenryck, P, 1991, "An efficient arc consistency algorithm for a class of csp problems" *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*.

Doyle, J, 1979, "A truth maintenance system" *Artificial Intelligence* **12** 231–272.

Dubois, D, Fargier, H and Prade, H, 1995, "Fuzzy constraints in job-shop scheduling" *Journal of Intelligent Manufacturing* **6** 215–235.

Dubois, D, Fargier, H and Prade, H, 1996a, "Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty" *Applied Intelligence* **6** 287–309.

Dubois, D, Fargier, H and Prade, H, 1996b, "Refinements of the maximin approach to decision making in a fuzzy environment" *Fuzzy Sets and Systems* **81** 103–122.

Dubois, H and Prade, D, 1988, *Possibility Theory: An Approach to Computerised Processing of Uncertainty* Plenum Press.

Fox, MS, 1987, *Constraint-Directed Search: A Case Study of Job-Shop Scheduling* Pitman, Morgan Kaufmann.

Freuder, EC, 1978, "Synthesizing constraint expressions" *Communications of the ACM* **21**(11) 958–966.

Freuder, EC, 1982, "A sufficient condition for backtrack-free search" *Journal of the ACM* **29**(1) 24–32.

Freuder, EC, 1978, "Partial constraint satisfaction" *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* pp 278–283.

Freuder, EC and Wallace, RJ, 1992, "Partial constraint satisfaction" *Artificial Intelligence* **58** 21–70.

Frost, D and Dechter, R, 1994, "Dead-end driven learning" *Proceedings of the Twelfth National Conference on Artificial Intelligence* pp 294–300.

Frost, D and Dechter, R, 1995, "Look-ahead value ordering for constraint satisfaction" *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* pp 572–578.

Gaschnig, J, 1979, "Performance measurement and analysis of certain search algorithms" *Technical Report CMU-CS-79-124*, Department of Computer Science, Carnegie-Mellon University.

Ginsberg, ML, 1993, "Dynamic backtracking" *Journal of Artificial Intelligence Research* **1** 25–46.

Guan, Q and Friedrich, G, 1992, "Extending constraint satisfaction problem solving in structural design" *Proceedings of the 5th International Conference IEA/AIE, Paderborn, Germany* pp 341–350.

Han, C and Lee, C, 1988, "Comments on Mohr and Henderson's path consistency algorithm" *Artificial Intelligence* **36** 125–130.

Haralick, RM and Elliot, GL, 1980, "Increasing tree search efficiency for constraint satisfaction problems" *Artificial Intelligence* **14** 263–313.

Joslin, D and Pollack, ME, 1995, "Passive and active decision postponement in plan generation" *Proceedings of the 3rd European Workshop on Planning*.

Kondrak, G and van Beek P, 1995, "A theoretical evaluation of selected backtracking algorithms" *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* pp 541–547.

Kuipers, B, 1994, *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge* MIT Press.

Lang, J, 1991, "Possibilistic logic as a logical framework for min–max discrete optimisation problems and prioritised constraints" *Proceedings of the International Conference on the Fundamentals of Artificial Intelligence Research* pp 112–126.

Mackworth, A and Freuder, E, 1985, "The complexity of some polynomial network-consistency algorithms for constraint-satisfaction problems" *Artificial Intelligence* **25** 65–74.

Mackworth, AK, 1977, "Consistency in networks of relations" *Artificial Intelligence* **8**(1) 99–118.

Mackworth, AK, 1977, "On reading sketch maps" *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* pp 598–606.

McDermott, D, 1991, "A general framework for reason maintenance" *Artificial Intelligence* **50** 289–329.

Meseguer, P, 1989, "Constraint satisfaction problems: An overview" *AI Communications* **2**(1) 3–17.

Meseguer, P and Larrosa, J, 1997, "Solving fuzzy constraint satisfaction problems" *Proceedings of FUZZ-IEEE* pp 1233–1238.

Miguel, I and Shen, Q, 1998, "Extending qualitative modelling for simulation of time-delayed behaviour" *Proceedings of the Twelfth International Workshop on Qualitative Reasoning* pp 161–166.

Minton, S, Johnston, MD, Philps, AB and Laird, P, 1992, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems" *Artificial Intelligence* **58** 161–205.

Mohr, R and Henderson, T, 1986, "Arc and path consistency revisited" *Artificial Intelligence* **28** 225–233.

Montanari, U, 1974, "Networks of constraints: Fundamental properties and applications to picture processing" *Information Science* **7**95–132.

Moura Pires, J and Prade, H, 1998, "Logical analysis of fuzzy constraint satisfaction problems" *Proc. of FUZZ-IEEE-98* pp 957–962.

Prosser, P, 1993, "Hybrid algorithms for the constraint satisfaction problem" *Computational Intelligence* **9**(1).

Rosenfeld, A, Hummel, R and Zucker, S, 1976, "Scene labeling by relaxation operations" *IEEE Transactions on Systems, Man, and Cybernetics* **6** 420–433.

Schiex, T, 1992, "Possibilistic constraint satisfaction problems, or how to handle soft constraints" *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence* pp 268–275.

Schiex, T, Fargier, H and Verfaillie, G, 1995, "Valued constraint satisfaction problems: Hard and easy problems" *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* pp 631–637.

Schiex, T and Verfaillie, G, 1993, "Nogood recording for static and dynamic CSP" *Proceedings of the fifth IEEE International Conference on Tools with Artificial Intelligene*.

Shapiro, L and Haralick, R, 1981, "Structural descriptions and inexact matching" *IEEE Transactions on Pattern Analysis and Machine Intelligence* **3**(5) 504–518.

Shen, Q and Leitch, L, 1993, "Fuzzy qualitative simulation" *IEEE Transactions on Systems, Man, and Cybernetics* **23**(4) 1038–1061.

Stallman, RM and Sussman, GJ, 1977, "Forward reasoning and dependency-directed backtracking in a system for computer aided circuit analysis" *Artificial Intelligence* **9** 135–196.

Stefik, M, 1981, "Planning with constraints (Molgen: Part 1)" *Artificial Intelligence* **16** 111–140.

Tsang, EPK, 1993, *Foundations of Constraint Satisfaction* Academic Press.

van Hentenryck, P, 1989, *Constraint Satisfaction in Logic Programming* MIT Press.

van Hentenryck, P and Provost, TL, 1991, "Incremental search in constraint logic programming" *New Generation Computing* **9** 257–275.

Verfaillie, G and Schiex, T, 1994, "Solution reuse in dynamic constraint satisfaction problems" *Proceedings of the Twelfth National Conference on Artificial Intelligence* pp 307–312.

Wallace, RJ, 1996, "Enhancements of branch and bound methods for the maximal constraint satisfaction problem" *Proc. of AAAI-96* pp 188–195.

Wallace, RJ and Freuder, EC, 1998, "Stable solutions for dynamic constraint satisfaction problems" In: M Maher and J-F Puget (eds) *Principles and Practice of Constraint Programming – CP98* Springer-Verlag.

Waltz, D, 1975, "Understanding line drawings of scenes with shadows" In: PH Winston (ed) *The Psychology of Computer Vision* McGraw-Hill, pp 19–91.