Artificial Intelligence Review **15:** 243–267, 2001.

Solution Techniques for Constraint Satisfaction Problems: Foundations

I. MIGUEL and Q. SHEN
School of Artificial Intelligence, Division of Informatics, University of Edinburgh, Edinburgh, EH1 1HN, UK

Abstract. The Constraint Satisfaction Problem (CSP) is ubiquitous in artificial intelligence. It has a wide applicability, ranging from machine vision and temporal reasoning to planning and logic programming. This paper attempts a systematic and coherent review of the foundations of the techniques for constraint satisfaction. It discusses in detail the fundamental principles and approaches. This includes an initial definition of the constraint satisfaction problem, a graphical means of problem representation, conventional tree search solution techniques, and pre-processing algorithms which are designed to make subsequent tree search significantly easier.

Keywords: Constraint Satisfaction Problems, constraint graph, tree search, pre-processing, consistency enforcing

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 1992; Tsang 1993) involves a fixed set of problem *variables*, each with an associated *domain* of potential values. A set of *constraints* range over these variables which specify the allowed combinations of value assignments. In order to solve a classical CSP, it is necessary to find one or all assignments to all the variables such that all constraints are satisfied. Although reviews of the CSP literature exist (Kumar 1992; Meseguer 1989), they tend to focus on specific approaches. This paper tries to provide a coherent view of existing solution methods for CSPs, using a unified representation and similar illustrative examples for different approaches.

A variety of different problems can, when formulated appropriately, be seen as instances of the classical CSP which can then be solved using some of the techniques reviewed herein. For example, an early application of constraint satisfaction techniques appears in the field of machine vision.

Waltz presents a method for producing semantic descriptions given a line drawing of a scene, leading to the Waltz algorithm (Waltz 1975). In this case, variables are picture junctions which have potential values consisting of possible interpretations as corners, and constraints specify that each edge in the drawing must have the same interpretation at both of its ends. Other examples of constraint satisfaction in machine vision can be found in Kolbe (1998); Mackworth (1977b); Montanari (1974); Rosenfeld et al. (1976); and Shapiro and Haralick (1981).

Another area that involves extensive use of constraint satisfaction is that of planning and scheduling. For example, Fox presents a system for the scheduling of factory job-shops (Fox 1987). This system is capable of representing a variety of constraint types, including organisational goals, physical constraints and user preferences. In addition it is able to selectively relax less important constraints in case of a conflict. Other planning/scheduling applications are presented in Blum and Furst (1997); Prosser (1980); and Rodosek and Wallace (1998).

The list of applications of CSP is as broad as it is deep. Further examples include belief maintenance (de Kleer 1989; Dechter 1987; Dechter and Dechter 1988; Doyle 1979; McDermott 1991), puzzles (Nadel 1989), temporal reasoning (Allen 1984; Rit 1986; Tsang 1987), graph problems (Bruynooghe 1985; Fowler et al. 1983; McGregor 1979; Ullman 1976), propositional satisfiability problems (Ginsberg and McAllester 1994; Zabih and McAllester 1988), logic programming (Borning et al. 1989; van Hentenryck 1989), systems simulation (Kuipers 1994; Miguel and Shen 1998; Shen and Leitch 1993), and circuit analysis (Stallman and Sussman 1977).

The rest of this paper is structured as follows. Section 2 presents a formal description of CSP and examines the *constraint graph* representation that is used for illustration throughout. Section 3 describes conventional tree search solution methods, beginning with *Backtrack* and describing how this basic algorithm can be improved leading to greater efficiency. *Conflict Recording* extracts information about the problem at a dead end in tree search. Finally, *Dynamic Backtrack* attempts to maintain some of the work that Backtrack discards while backtracking. Section 3 concludes with a note on the relative merits of the algorithms presented thus far. Section 4 describes pre-processing techniques which simplify a CSP to ease the workload of subsequent tree search. It begins by describing the method of filtering variable domain elements that cannot take part in any solution. The techniques of enforcing *arc consistency* as well as the more powerful (and more costly) *path consistency* are presented before the general concept of *k-consistency* is introduced. *Directional* pre-processing improves efficiency via a fixed instantiation order.

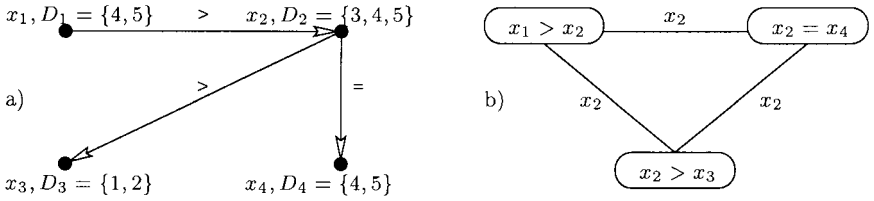

Figure 1. Constraint graph and dual graph representations of a CSP.

The review of the foundations of techniques for solving CSPs is concluded in section 5. A companion review concerning advanced constraint satisfaction techniques may be found in Miguel and Shen (2001).

2. Constraints and Constraint Graphs

A constraint satisfaction problem (CSP) involves a set of n variables, $X = \{x_1, \ldots, x_n\}$, and a set of constraints, C, over these variables. An associated domain, D_i, contains possible values for x_i. A constraint $c(x_i, \ldots, x_j) \in C$ specifies a subset of the Cartesian product $D_i \times \ldots \times D_j$ indicating the variable assignments that are compatible with each other. A *solution* to a CSP is a complete *assignment* of values to the variables such that all constraints are simultaneously satisfied. A problem solver must find one or all such solutions.

A CSP containing at most binary constraints may be viewed as a *constraint graph* which can guide a problem solver. Each graph node corresponds to a problem variable, and both unary and binary constraints are represented by labelled, directed arcs (in the case of a unary constraint the arc begins and ends at a single node). For each $arc(i, j)$ there is a second $arc(j, i)$, since $c(x_i, x_j) = c(x_j, x_i)$.[1] Figure 1a shows an example problem. Note that for pictorial clarity, only $arcs(i, j)$ are shown, where $i < j$.

Two alternatives are used to represent n-ary constraints. A *primal graph* (Dechter and Pearl 1989) is similar to a constraint graph: nodes represent variables and an arc connects any two variables that are related by a constraint. The primal graph representation of Figure 1a is similar since the constraints are binary. Each node of a *dual graph* (Dechter and Pearl 1989) represents a constraint with an associated domain of allowed assignment tuples. Arcs connect any two nodes that share common variables. Figure 1b shows the dual graph representation of the CSP shown in Figure 1a. Dual graphs allow binary CSP techniques to be applied to arbitrary n-ary CSP (Dechter 1992).

3. Backtrack Search and Its Derivatives

Generate and Test is a simple CSP solution technique. Each possible complete variable assignment is generated and tested to see if it satisfies all constraints. At worst, the number of complete assignments tested is the size of the Cartesian product of all variable domains. This brute force method is very expensive on problems of any reasonable size.

3.1 *Backtrack*

Chronological backtracking (or *Backtrack*) improves this initial scheme, viewing the search process as the incremental extension of partial solutions (Bitner and Reingold 1975). The *instantiation order* is the order in which variable assignments are made during search. It will be assumed that this order is fixed and follows the numbering scheme of the variables. At a particular *level*, i, of the instantiation order, *past* variables have indices less than i and have been assigned values. *Future* variables have indices greater than i and do not yet have assignments.

Figure 2 describes the Backtrack algorithm. For simplicity, the pseudo-code presented herein makes no distinction between finding no solution and finding no *more* solutions; `False` is returned on both occasions. Binary constraints are also assumed for clarity. This is not a weakness of the algorithms presented. The following notation is used:

- **n**. The number of variables in the problem.
- **Assignments [ ]**. This array of size n records the value currently assigned to each variable.
- **D[ ]**. The ith element of this size n array contains the sequence of domain elements associated with the variable x_i.

All algorithms have access to `Test`(i, j), which returns `True` if $c(x_i, x_j)$ is satisfied by the current assignments to x_i and x_j. If $c(x_i, x_j)$ does not exist, `Test( )` returns `True` (this is *not* counted as a constraint check).

Each element of D_i is assigned to x_i in turn, extending an assignment to $x_1, \ldots, x_{i-1}$. If a constraint check fails against a past variable, this partial assignment is not extended further. If D_i is exhausted (a *dead end* or *domain wipeout*) the algorithm *backtracks* to level $i - 1$ to find another consistent assignment for x_{i-1}. If none exists it backtracks to x_{i-2}, and so on until a variable is found with another consistent assignment or the CSP is shown to have no solution. Given a consistent assignment (all constraint checks succeeded), the recursive call `BT`$(i + 1)$ is made or, if no future variables remain, the solution is output.

Backtracking may be viewed as tree traversal. Figure 3 shows how Backtrack solves the problem in Figure 1. Each branch represents a partial

```
Procedure BT (i)
  Foreach Val In D[i]
    Assignments[i] ← Val
    Consistent ← True
    For h ← 1 To i − 1 While Consistent
      Consistent ← Test(i, h)
    If Consistent
      If i = n
        Show − Solution()
      Else
        BT(i + 1)
  Return False
End Procedure
```

Figure 2. The Backtrack algorithm.

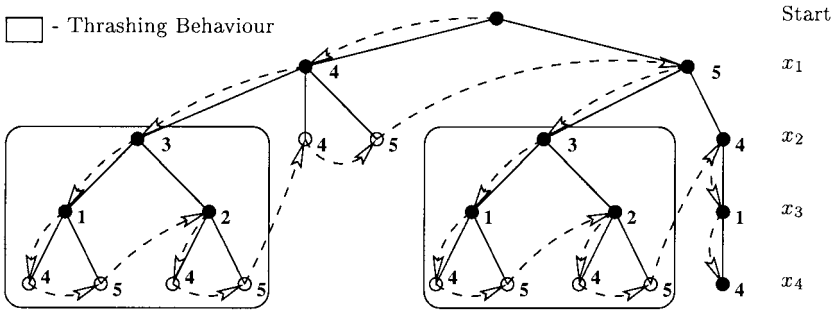

Figure 3. Backtrack as a process of tree search.

assignment (which is complete if the branch extends to the nth variable) and the ith level of the tree represents choices made for the ith variable in the instantiation order. The *depth* is the distance from the root. Black and white nodes represent successful and failed assignments at this point in the search respectively. 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.

By analogy, Generate and Test always extends each search branch to a complete assignment before making any constraint checks. Backtrack prunes subsections of the search space by rejecting a partial assignment as soon as it is clear that it cannot yield a solution to the CSP. However, the worst case time complexity of Backtrack is exponential in the number of variables (Mackworth 1977). It is inefficient in that it usually performs more operations than are necessary to find a solution.

3.1.1 *The thrashing problem*

Thrashing (Gaschnig 1979) is a major factor in the poor behaviour of Backtrack: search fails repeatedly for the same reason. For example: $c(x_g, x_i)$ specifies that a particular assignment to x_g disallows all potential values for

```
Procedure BJ (i)
  MaxCheckLevel ← 0
  ReturnDepth ← 0

  Foreach Val In D[i]
    Assignments[i] ← Val
    Consistent ← True

    For h ← 1 To i − 1 While Consistent
      Consistent ← Test(i, h)
    MaxCheckLevel ← h − 1; obtain last loop value

    If Consistent
      If i = n
        Show − Solution()
      Else
        MaxCheckLevel ← BJ(i + 1)
        If MaxCheckLevel < i
          Return MaxCheckLevel
    ReturnDepth ← Max (ReturnDepth, MaxCheckLevel)

  Return ReturnDepth
End Procedure
```

Figure 4. The Backjump algorithm.

x_i. Backtrack fails at level i for each element of D_i, repeating this failure for every combination of assignments in the intermediate levels, x_h, where $g < h < i$. Thrashing is present in Backtrack's attempt to solve the problem of Figure 1. Figure 3 shows the repeated failure of the search due to $c(x_2, x_4)$. Thrashing becomes steadily worse as the number of intermediate variables (and their domain sizes) increases.

3.2 *Backjumping*

Gaschnig (1979) proposed the *Backjump* algorithm to alleviate thrashing (see Figure 4). If a constraint check with a past variable x_h fails, the `MaxCheckLevel` (the deepest variable that a check is made against) is set to h.[2] After a successful assignment the `MaxCheckLevel` is $i-1$ (checks against all past variables succeeded). The `ReturnDepth` is the deepest level checked against by any element of D_i. If D_i is exhausted, Backjump 'jumps' back to the `ReturnDepth` level. A new consistent value is assigned at this level and search continues. Figure 5 shows how Backjump solves the problem of Figure 1. Thrashing is avoided by jumping back to the cause of the inconsistency: from x_4 to x_2.

Given a consistent assignment, Backjump proceeds as per Backtrack. If no future variables remain a solution has been found, otherwise Backjump is called recursively. The result of this invocation is the `ReturnDepth` at level $i + 1$. If this value, say h, is less than i, Backjump is in the process of 'jumping' back to level h, effectively bypassing level i. The invocations

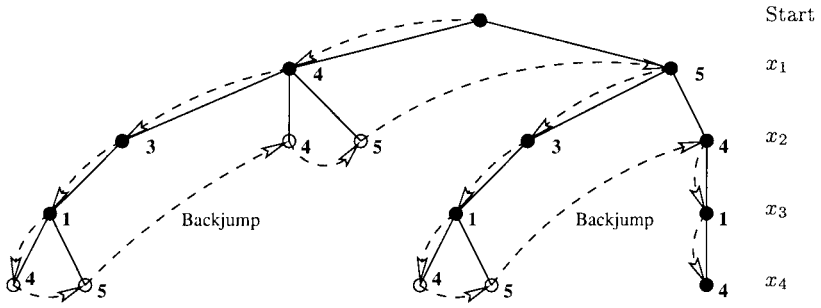

Figure 5. Example for the Backjump algorithm.

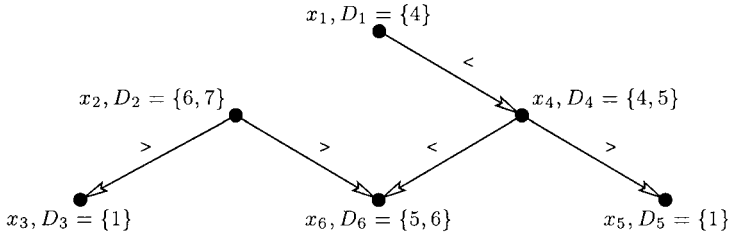

Figure 6. Failure example for max-fail Backjump.

are unwound until level h is reached. Otherwise, the algorithm has jumped (or indeed stepped) back to this level, i, so the main loop proceeds to select a new value to assign to x_i.

3.2.1 *Problems with Backjump*

By jumping back to the maximum level *checked* against, Backjump can only make one jump backwards when attempting to resolve inconsistencies. The level, say h, jumped back to always has a `ReturnDepth` of $h - 1$ because the current assignment to x_h must have passed all consistency checks for the search to have proceeded forwards to level i. Hence, if there are no further consistent values to be assigned to x_h, the algorithm resorts to chronological backtracking. If D_h is exhausted, there must be a previous assignment which is in conflict with levels h and i. It is not necessarily the case that level $h - 1$ is the conflicting level, so thrashing occurs until the conflicting variable is re-assigned.

One remedy is to record the deepest *failure* level and jump back to that, but this leads to an incomplete algorithm. Consider the CSP shown in Figure 6 and the search tree generated by Backjump altered to record the deepest level at which a constraint check failed (Figure 7a). The algorithm fails to find the solution. Elements $\{5, 6\} \in D_6$ conflict with the current assignments to x_4 and x_2 respectively. The algorithm jumps back to the deepest (x_4), but

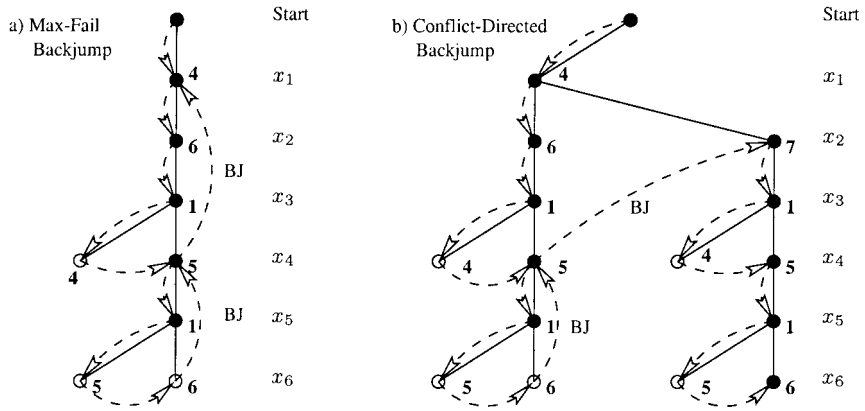

Figure 7. Attempts to solve the problem specified in Figure 6.

no alternative consistent assignment exists. Element $\{4\} \in D_4$ is in conflict with the current assignment to x_1. The algorithm thus tries to find a new value for x_1 and fails. The fact that x_6 was also in conflict with x_2 was discarded when the jump was made from x_6 to x_4. Therefore, the second jump from x_4 to x_1 jumps over x_2 and accidentally prunes the solution.

3.3 *Conflict-directed Backjumping*

Conflict-directed Backjump (CBJ) (Prosser 1993) improves Backjump further (see Figure 8). A series of jumps is made by recording the *Conflict Set* (`CSet`$[i]$ at level i) of past variables that *failed* constraint checks with the current variable. Jumps are made to the deepest value in this set. When jumping from i to h, `CSet`$[h]$ is set to the union of the conflict sets at levels i and h. A further jump, if necessary, is made to the deepest variable in conflict with either x_i or x_h, avoiding the problems encountered by max-fail Backjump.

At each invocation of `CBJ`(i), `CSet`$[i]$ is reset to clear the conflict set for this level of any information from previous invocations. If a solution is found, $\{n-1\}$ is added to the conflict set to ensure that CBJ backtracks to level $n-1$ when D_n is exhausted. It is only possible to backtrack one level in this case because there is no inconsistency with this level (a solution was found here). If a successful assignment is found but future variables remain, CBJ is called recursively, and the result stored in `ReturnedDepth`. This value is used, as by Backjump, to discern whether the algorithm has returned to this level to select a new value, or is in the process of jumping back over this level.

Figure 7b shows the search tree for CBJ when solving the problem in Figure 6. The critical point follows the initial jump back from level 6 to

```
Procedure CBJ(i)
  CSet[i] ← {0}
  Foreach Val In D[i]
    Assignments[i] ← Val
    Consistent ← True
    For h ← 1 To i − 1 While Consistent
      Consistent ← Test(i, h)
    If Not Consistent
      CSet[i] ← CSet[i] + {h − 1}; obtain last loop value
    If Consistent
      If i = n
        Show − Solution()
        CSet[i] ← CSet[i] + {n − 1}
      Else
        ReturnedDepth ← CBJ(i + 1)
        If ReturnedDepth < i
          Return ReturnedDepth
  ReturnDepth ← Max(CSet[i])
  CSet[ReturnDepth] ← Union(CSet[ReturnDepth], CSet[i] − {ReturnDepth})
  Return ReturnDepth
End Procedure
```

Figure 8. The conflict-directed Backjump algorithm.

level 4. Where max-fail Backjump took into account only those variables in conflict with x_4, CBJ remembers that x_2 was also in conflict with x_6 and jumps back to level 2 as the deepest conflicting assignment.

3.4 *Backmarking*

Gaschnig (1979) also proposed the *Backmark* algorithm (see Figure 9), which minimises the execution of redundant constraint checks in conjunction with chronological backtracking. First, Backmark avoids performing constraint checks that must fail: if an assignment previously failed against a past variable whose instantiation has not changed, the same assignment will fail again. Secondly, an assignment that succeeded against a set of past variables whose instantiations are unchanged will satisfy repeated consistency checks against the same past assignments.

In order to make use of these observations, two arrays are necessary:

- **Mark[ ]** is dimensioned by the number of variables, n, and by d, the size of the largest variable domain. `Mark`$[i, a]$ contains the most shallow level at which a consistency check failed for the assignment of $d_{i_a} \in D_i$ to x_i.
- **MinBackupLevel[i]** records the most shallow level at which an assignment has changed since the last assignment at level i.

Backmark generates the same search path as Backtrack, but makes less constraint checks. Consider Figure 3: Backmark makes four savings: within

```
Procedure BM(i)
  For Dindex ← 1 To Length(D[i])
    If Mark[i, DIndex] ≥ MinBackupLevel[i]
      Assignments[i] ← nth(DIndex, D[i])
      Consistent ← True

      For h ← MinBackupLevel[i] To i − 1 While Consistent
        Consistent ← Test(i, h)
      Mark[i, DIndex] ← h − 1

      If Consistent
        If i = n
          Show − Solution()
        Else
          BM(i + 1)

  MinBackupLevel[i] ← i − 1
  For Index ← i + 1 To n
    MinBackupLevel[Index] ←Min(MinBackupLevel[Index], i − 1)
End Procedure
```

Figure 9. The Backmark algorithm.

the thrashing region when Backtrack attempts to find a solution with $x_3 = 2$, and $x_4 = 4$, or $x_4 = 5$, the two assignments to x_4 must fail since the assignment to x_2 has not changed since the last failure.

Backmark counts through the elements of D_i, using this index to access `Mark[ ]`. For each element of D_i, it checks whether `Mark`$[i, DIndex]$ is less then `MinBackupLevel`$[i]$. If so, the assignment to x_i must fail, since the past variable that caused it to fail previously is unchanged. It is possible to skip to a new assignment without redundantly checking this value. If `Mark`$[i, DIndex]$ is greater or equal to `MinBackupLevel`$[i]$, some saving can still be made. Past assignments before `MinBackupLevel`$[i]$ are unchanged and the shallowest level of inconsistency with the current assignment was deeper than this level, so consistency checks with assignments before `MinBackupLevel`$[i]$ must succeed. Only checks with the levels between `MinBackupLevel`$[i]$ and i exclusive are necessary. If a constraint check fails, its level is recorded in `Mark`$[i, DIndex]$.

If D_i is exhausted, chronological backtracking occurs. Since the algorithm is about to backtrack to level $i-1$, it is known that the assignment at that level will change. Hence, `MinBackupLevel`$[i]$ is updated to $i - 1$. In addition, the `MinBackupLevel` of the levels deeper than i may be updated if $i - 1$ is less than their current value.

3.5 *The Backmark hybrids*

Hybrid tree search algorithms combine Backmark and either Backjump or CBJ to create BackmarkJump (BMJ), and Backmark-Conflict-Directed-Backjump (BM-CBJ) respectively. This is possible because Backmark

reduces the number of consistency checks, whilst Backjump and CBJ reduce the number of search tree nodes to be explored. BMJ was first suggested in Nadel (1989) and subsequently presented in Prosser (1993). It maintains the information required by *both* base algorithms: in addition to `Mark[ ]` and `MinBackupLevel[ ]`, BMJ also maintains `MaxCheckLevel[ ]` so that a jump backwards may be performed in the event of a domain wipeout. Similarly, BM-CBJ (Prosser 1993) maintains the conflict set required for conflict-directed Backjump.

Unfortunately, an apparently attractive pair of algorithms can sometimes perform more consistency checks than the original Backmark. This is because Backmark was originally designed for use with chronological backtracking. When Backjump or CBJ jump over a level i, any remaining elements of D_i are not tested against previous assignments. The `MinBackupLevel[`i`]` is deprived of the information that it would have learnt from the consistency checks made against these untested domain elements, hence the sometime behaviour degradation.

Kondrak and van Beek (1995) extend `MinBackupLevel[ ]` to two dimensions to solve this problem. The extended array is dimensioned by the number of variables, n, and d, the size of the largest variable domain. A particular entry $[i, j]$ in this array stores the index of the most shallow level whose assignment has changed since x_i was last assigned the jth element of D_i. This method maintains the maximum amount of 'backup' information. The extended `MinBackupLevel[ ]` array is used as the foundation of Kondrak and van Beek's two hybrids BMJ2 and BM-CBJ2, both of which always perform better than Backmark (in terms of the number of constraint checks).

3.6 *Conflict recording*

At a dead end, the assignments to $x_1, \ldots, x_{i-1}$ form a *conflict set* with x_i. This information is used by recording one or more constraints (or *nogoods*) which aid future search by disallowing the inconsistencies that led to this dead end. Consider Figure 1 and the search tree generated by Backtrack (Figure 3). At the first dead end, the conflict set $CS = \{x_1 = 4, x_2 = 3, x_3 = 1\}$ conflicts with x_4. Recording CS is only useful if the CSP will be queried again: this exact assignment set will not occur again during this search. However, CS probably contains subsets in conflict with x_i. *Minimal conflict sets* (Bruynooghe 1985) contain no other conflict sets and are responsible for the current conflict. The search process benefits if these minimal sets are prevented from re-occurring.

Conflict recording can be viewed as *learning* (Dechter 1990): useful information uncovered by the problem solver is recorded for later use. *Deep learning* finds all minimal conflicts at a dead end, embodied by *Dependency-*

Directed Backtracking (Stallman and Sussman 1977) and commonly used in *truth-maintenance systems* (e.g. Doyle 1979). It is relatively expensive: each dead end presents a new opportunity to add constraints. Hence, a large proportion of the search space is recorded.

Shallow learning (Dechter 1990) limits the amount of work done at a dead end, say x_i, by removing all variable assignments from the conflict set which are irrelevant (i.e. consistent with all possible assignments) to x_i. *Graph-based shallow learning* uses the constraint graph to test (incompletely) for irrelevancy: x_h is irrelevant to x_i if the variables are not adjacent in the graph. In the above example, this rules out $\{x_1 = 4, x_3 = 1\}$, leaving $\{x_2 = 3\}$. This conflict is recorded by removing $\{3\}$ from D_2. *Value-based learning* removes *all* irrelevant variables from the conflict set. *Jump-back learning* uses the conflict set recorded by the CBJ algorithm to avoid extra work (Frost and Dechter 1994).

The *Stubbornness* approach (Schiex and Verfaillie 1994) makes unnecessary mistakes to extract the maximum information. Adding many constraints, however, hinders search by increasing the number of constraint checks. Recorded conflicts should have a high probability of being useful. Dechter (1990) limits the size of the conflicts recorded. *First-order learning* records conflicts concerning just one variable. *Second-order learning* adds constraints of up to two variables. First- and second-order learning can be used independently of the depth of learning.

Dechter (1990) compared shallow and deep first and second order learning. On easy problems learning has little effect, but on more difficult problems large gains in performance are seen. It is not uniformly the case that the strongest form of learning gave the most benefit. In some cases the overheads outweigh the benefits.

3.7 *Dynamic Backtracking*

Jumping or stepping back over a level erases the work done by a problem solver in determining a consistent assignment at that level. In many cases, levels backtracked over may have nothing to do with the inconsistency that the algorithm is trying to resolve (e.g. the thrashing behaviour described previously). Therefore, potentially useful work is needlessly discarded in the search for the real cause of the inconsistency.

Ginsberg (1993) presented *Dynamic Backtracking* (DBT) as an approach to this problem. Erasing information which is irrelevant to the current inconsistency is due to the nature of the algorithms presented thus far which must unwind assignments sequentially until the inconsistency can be resolved. DBT dispenses with a fixed order of instantiation, instead choosing variables to assign values to in a dynamic manner (hence the name). This dynamic

ordering is beneficial when an inconsistency is detected. DBT effectively changes the instantiation order such that the level responsible for the inconsistency becomes the deepest level, leaving intact all uninvolved variable assignments.

For each $d_{i_a} \in D_i$, DBT maintains a set of *eliminating explanations* which record the reason why d_{i_a} is currently excluded from consideration. These explanations are defined in terms of the current assignments to past variables. When a jump back over x_i to x_h is necessary, DBT retains the current assignment to x_i and all of its eliminating explanations except those involving x_h (whose assignment is about to change).

The use of eliminating explanations is similar to the *nogoods* mentioned in the context of conflict recording. The prohibitive space complexity that is incurred by storing all derived nogood information is avoided by DBT. Eliminating explanations are kept for only the *current* assignments to other variables; once an assignment changes, any explanations involving it are discarded. Hence, DBT represents a compromise in the tradeoff between space and time complexity.

DBT does not, however, maintain a uniform improvement over existing techniques. This is because it is sometimes *desirable* to erase intermediate information. In comparisons using random propositional satisfiability problems and employing standard heuristics (Miguel and Shen 1999), Baker shows that DBT performs worse than Backjump by a factor that is exponential in the size of the problem (Baker 1994). The problem is that the variable assignments that DBT does not discard often have no justification (according to the heuristics) once an inconsistency is found. Maintaining these variable assignments therefore degrades the operation of the heuristic. In light of this, Baker proposes a method of selectively erasing variable assignments that, following a detected inconsistency, no longer have a heuristic justification.

3.8 *Relative evaluation*

It is difficult to pick one 'best' algorithm from those presented. There have been several attempts to compare a number of them empirically (Baker 1994; Gaschnig 1979; Jonsson and Ginsberg 1993; Prosser 1993). The means of comparison is usually the number of constraint checks made and/or the number of nodes in the search tree. These measures fail to take into account the overhead of individual algorithms in terms of maintaining data structures. To try to quantify this overhead, some attempts have been made to use run time, flawed though it is, also as a means of comparison (Prosser 1993).

Kondrak and van Beek (1995) have made a purely theoretical comparison of tree search algorithms. The comparison was based on both the number of search tree nodes explored and number of consistency checks. In terms of

the former, Backtrack and Backmark (which has no effect on the number of search nodes) are unsurprisingly the worst. The most efficient were CBJ and its Backmark hybrids. In terms of consistency checks, Backtrack was again the worst algorithm, whilst the hybrids of Backmark were seen to be the best, in particular BM-CBJ2.

4. Pre-Processing Techniques

Pre-processing a CSP creates an *equivalent* (i.e. with the same solution set) but simpler problem. Tree search can solve the resulting problem with much less effort than would otherwise be required. The original CSP is modified as follows. First, variable domains may be *filtered* to remove elements that cannot take part in any solution. Second, the set of constraints may be modified to disallow inconsistent assignment combinations. Modifications are made via *constraint propagation*: a local group of constraints is used to deduce information, which is recorded as an updated constraint or variable domain. This local update is the basis for further deductions, hence the result of any change is gradually propagated through the problem.

Basic pre-processing involves a unary constraint, $c(x_i)$. *Node consistency* holds if for each $d_{i_a} \in D_i$, $c(x_i)$ is satisfied. To enforce local node consistency, all elements of D_i which do not satisfy $c(x_i)$ are removed. Global node consistency may be enforced in one pass: unary constraints affect single variables, so a local change cannot affect other variables. The following sections describe methods for enforcing higher consistency levels. Although, for presentational simplicity, the techniques are presented as operating on binary CSPs, they are applicable to n-ary CSP – for example by making use of the dual graph representation.

4.1 *Arc consistency*

The logical next step in terms of pre-processing is to consider a single $arc(i, j)$ (see section 2). Local *arc consistency* is defined as follows: both nodes x_i, and x_j must be node consistent, and for each $d_{i_a} \in D_i$, there is a corresponding element $d_{j_b} \in D_j$ such that $c(x_i, x_j)$ is satisfied by $x_i = d_{i_a}$, $x_j = d_{j_b}$. Global arc consistency holds if this condition exists between all pairs of variable nodes connected by an arc.

To enforce arc consistency on $arc(i, j)$, for each $d_{i_a} \in D_i$, an element $d_{j_b} \in D_j$ is sought such that the arc consistency condition holds; d_{j_b} *supports* d_{i_a}. All domain elements with no support are removed. Arc consistency is *directional*: in Figure 10 consistency has been enforced on $arc(1, 2)$ (Figure 10b), but a further operation is required to enforce consistency on $arc(2, 1)$

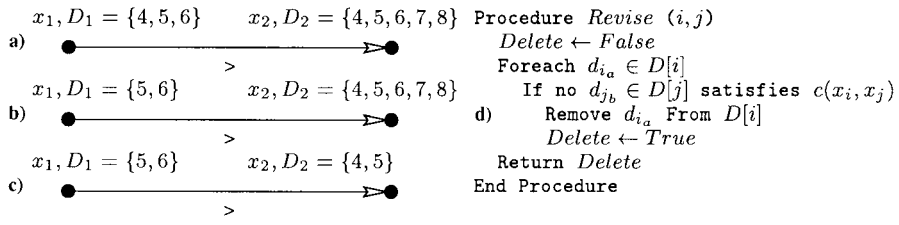

Figure 10. Enforcing arc consistency via `Revise( )`.

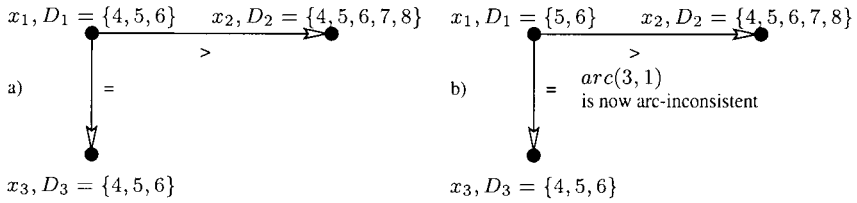

Figure 11. Global effect of enforcing arc consistency locally.

(Figure 10c). The process of enforcing local arc consistency is embodied by `Revise( )`, shown in Figure 10d (adapted from Mackworth 1977). `Revise( )` returns `True` if D_i is changed and `False` if not.

`Revise( )` cannot simply be applied sequentially to every arc in the problem since each arc involves two variables; enforcing local arc consistency might remove the support and so violate the consistency of a separate arc, as shown in Figure 11. Initially, $arc(1, 3)$ and $arc(3, 1)$ are arc consistent (Figure 11a). `Revise(1,2)` is then called, which has the effect of removing the element 4 from D_1. This removes the support for element 4 in D_3, violating the consistency of $arc(3, 1)$ (Figure 11b).

The simplest method of achieving global arc consistency is to iterate 'passes' of the revision of every arc in the problem until there is no change in the constraint graph following a particular pass. This approach is embodied by the algorithm `AC` $-$ 1 (adapted from Machworth 1977; see Figure 12a). The procedure `NC( )` enforces node consistency and `Q` is a queue of arcs to be processed. The inefficiency of this approach is clear: all arcs are re-revised regardless of the possibility of their becoming arc inconsistent. A local change to $arc(i, j)$ can only affect the arc consistency of $arcs(h, i)$, as per the example of Figure 11.

Waltz's (1975) algorithm enforces arc consistency in one pass through the constraint arcs. Arc consistency is achieved by making a local subset of the arcs consistent, then performing the revisions required to maintain consistency when a new node is added to the subset. The Waltz algorithm is sometimes named as `AC` $-$ 2 (Mackworth 1977).

```
a)  Procedure AC − 1
      For i ← 1 To n
        NC(i)
      Q ← all arcs(i, j)
      Repeat
        Change ← False
          Foreach arc(i, j) in Q
            Change ← Revise(i, j) Or Change
      Until Not Change
    End Procedure
```

```
b)  Procedure AC − 3
      For i ← 1 To n
        NC(i)
      Q ← all arcs(i, j)
      While Q Not Empty
        Remove any arc(i, j) from Q
        If Revise(i, j)
          Add to Q all arcs(h, i)
                    (h ≠ i, h ≠ j)
    End Procedure
```

Figure 12. The AC − 1 and AC − 3 algorithms.

1 $Q = \{arc(1,2), arc(2,1), arc(2,3), arc(2,4), arc(3,2), arc(4,2)\}$

2 *Revise*$(1,2)$
- D_1 is unchanged
- $Q = \{arc(2,1), arc(2,3), arc(2,4), arc(3,2), arc(4,2)\}$

3 *Revise*$(2,1)$
- $D_2 = \{3,4\}$ (5 removed)
- Add $arc(3,2)$, $arc(4,2)$ to Q (both already present)
- $Q = \{arc(2,3), arc(2,4), arc(3,2), arc(4,2)\}$

4 *Revise*$(2,3)$
- D_2 is unchanged
- $Q = \{arc(2,4), arc(3,2), arc(4,2)\}$

5 *Revise*$(2,4)$
- $D_2 = \{4\}$ (3 removed)
- Add $arc(1,2)$, $arc(3,2)$ to Q (latter already present)
- $Q = \{arc(3,2), arc(4,2), arc(1,2)\}$

6 *Revise*$(3,2)$
- D_3 is unchanged
- $Q = \{arc(4,2), arc(1,2)\}$

7 *Revise*$(4,2)$
- $D_4 = \{4\}$ (5 removed)
- No arcs to add to Q
- $Q = \{arc(1,2)\}$

8 *Revise*$(1,2)$
- $D_1 = \{5\}$ (4 removed)
- No arcs to add to Q
- $Q = \{\}$

Figure 13. Example for the AC − 3 algorithm.

AC − 3 (Mackworth 1977) (see Figure 12b) is a generalisation of AC − 2 which allows arcs to be revised in any order. Q is initialised to contain the full set of arcs. Subsequently, arcs are selected and removed from Q and Revise( ) is applied to them, enforcing local arc consistency. AC − 3 gains in efficiency since arcs are not added to the queue for further revision unless they are possibly affected by the revision just performed. Note the special case where enforcing arc consistency on $arc(i, j)$ cannot violate arc consistency on $arc(j, i)$: all elements removed from D_i were removed exactly because there were no corresponding values in D_j.

AC − 3 is a popular, efficient method of achieving global arc consistency. As an example, an application of AC − 3 to the problem of Figure 1 is shown in Figure 13. There are no unary constraints in this problem, and it is assumed that arcs are selected from the queue in a first-in first-out manner. The globally arc consistent state of the constraint graph is shown in Figure 14a, and the subsequent application of Backtrack search is shown in Figure 14b. In this simple case, global arc consistency entirely forestalls any form of thrashing.

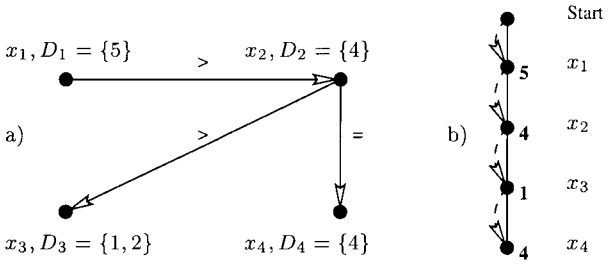

Figure 14. Impact of enforcing arc consistency on tree-search.

4.2 *Improving efficiency in enforcing arc consistency*

AC − 3 has a worst-case time complexity upper bound of $O(ed^3)$ (Mackworth and Freuder 1985); e is the number of constraint arcs, d is the maximum domain size. AC − 4 (Mohr and Henderson 1986) improves the worst-case bound to an optimal $O(ed^2)$ by enumerating the support for each domain element. A *counter* is associated with an arc-label pair, $(arc(i, j), d_{i_a})$, where $d_{i_a} \in D_i$. The counter's value equals the number of elements in D_j that support d_{i_a}. For each d_{j_b}, a set S_{j_b} is constructed which contains a set of pairs (i, d_{i_a}), such that d_{j_b} supports $d_{i_a} \in D_i$.

AC − 4 first constructs the aforementioned data structures, then filters domain elements. A list, L, is maintained of domain elements to be removed which is initialised with the domain elements that lack any support. Domain filtering progresses by selecting $d_{j_b} \in D_j$ from L, and removing it from D_j. S_{j_b} is then examined and the counter of each arc-label pair, $(arc(i, j), d_{i_a})$ such that $d_{j_b} \in D_j$ supported $d_{i\,a} \in D_i$ is decremented. All domain elements that consequently lack support (the counter reached 0 for a particular arc) are added to L for removal.

AC − 5 (Deville and van Hentenryck 1991) is a generic arc consistency algorithm which depends on two procedures, `ArcCons` and `LocalArcCons` which are specified but whose actual implementation is left open. AC − 3 and AC − 4 can be represented by a specific implementation of these two procedures. AC − 5 can also be implemented to produce an algorithm with a worst-case bound $O(ed)$ for monotonic and functional constraints.

Although AC − 4 has optimal worst-case time complexity, on *average* it performs less well than AC − 3 (Wallace 1993). It also has a high space complexity, which is a drawback for large problems. Hence, it is common for AC − 3 to be used in preference to AC − 4, especially for large problems (Bessiere 1994). AC − 6 (Bessiere 1994) maintains the optimal worst-case time complexity of AC − 4, but improves the space complexity. While AC − 4 records all support for a domain element, AC − 6 checks for only one

supporting value per domain element per constraint. Further support is looked for only when the current support is filtered out.

AC − 6 does not use the arc-label counters maintained by AC − 4. S_{j_b} now contains the set of pairs (i, d_{i_a}) such that $d_{j_b} \in D_j$ is the first element of D_j that supports $d_{i_a} \in D_i$ on $arc(i, j)$. The initialisation process searches for the minimum support required for each domain element, and adds to L any elements which lack this support. Subsequently, for each $d_{i_a} \in D_i$ on L alternative support is required for any element, $d_{j_b} \in D_j$ that depended on it. New support is found by checking forward from d_{i_a} in D_i until another element which supports d_{j_b} is found. If there is no new support, d_{j_b} is added to L and filtered.

The *AC-inference* schema (Bessiere et al. 1995) exploits knowledge about the properties of constraints to reduce the cost of consistency checking by *inferring* support for domain elements. A specific instance of this schema. AC − 7, makes use of a property common to *all* binary constraints: *bi-directionality*. That is, $d_{i_a} \in D_i$ supports $d_{j_b} \in D_j$ if and only if $d_{j_b} \in D_j$ supports $d_{i_a} \in D_i$. Hence, if support is found in one direction it can be inferred in the other. This algorithm is also the basis of *Lazy Arc Consistency* (Schiex et al. 1996), which specifically aims to detect global inconsistency of the problem with minimum effort.

4.3 *Path consistency*

Path Consistency (Mackworth 1977) is a more sophisticated level of consistency. Local path consistency holds if, given a path of length l through the nodes $(x_1, \ldots, x_l)$, for all values $d_{1_a} \in D_1$ and $d_{l_b} \in D_l$ such that $c(x_1)$, $c(x_l)$, and $c(x_1, x_l)$ hold, there is a sequence of values: $d_{2_c} \in D_2, \ldots, d_{l-1_c} \in D_l$ such that $c(x_2), \ldots, c(x_{l-1})$, $c(x_1, x_2)$, $c(x_2, x_3), \ldots, c(x_{l-1}, x_l)$ hold. Figure 15 shows an example for the path, (x_1, x_2, x_3). The initial state is shown in Figure 15a. A new constraint is added (Figure 15b) which disallows the *pair* of assignments $x_1 = 4, x_3 = 3$: given these assignments, there is no assignment to x_2 satisfying $c(x_1, x_2)$, $c(x_2, x_3)$. Enforcing arc consistency does *not* disallow this assignment pair.

Global path consistency holds if any pair of assignments which are consistent with a direct constraint between two variables are also consistent with all paths between the two nodes. If every path of length 2 in a *complete* constraint graph is path consistent, then path consistency holds globally (Montanari 1974). In a complete graph, a constraint arc connects each possible pair of nodes. An incomplete graph can be made complete by creating 'True' constraints that allow all possible combinations of assignments between two variables. Path consistency is enforced by modifying the direct constraint between the start and end points of a path to disallow all

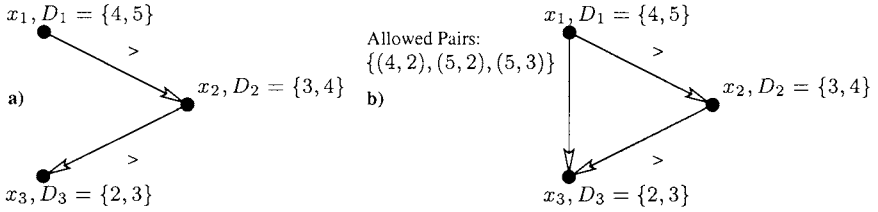

Figure 15. Effects of enforcing path consistency.

assignment pairs which violate the required condition. 'True' constraints may be updated, effectively adding a new explicit constraint to the problem (see Figure 15b).

The first path consistency algorithm was presented in Montanari (1974), and is equivalent to PC − 1 (Mackworth 1977). Mackworth (1977) improved the efficiency of PC − 1 using a revision-based method, as per AC − 3, to create PC − 2. PC − 3 (Mohr and Henderson 1986) improves efficiency via the enumeration of support, as per AC − 4. PC − 3 is, however, not completely correct. PC − 4 (Han and Lee 1988) is a correct version. Further improvements can be found in Chmeiss and Jegou (1996). Path consistency removes more of the workload of subsequent tree search, but with a significant computational overhead. The potential improvements must be weighed against this overhead.

4.4 *K-consistency*

A constraint graph is k-consistent (Freuder 1978) if, having assigned values to $k-1$ variables such that all constraints among them are satisfied, it is possible to choose a value for any kth variable that satisfies the constraints among all k variables. *Strong k-consistency* is defined as j-consistency for all $j \leq k$ (Freuder 1982). In this framework, node consistency corresponds to strong 1-consistency, arc consistency to strong 2-consistency, and path consistency to strong 3-consistency.

A sufficient consistency level may be enforced such that no backtracking is required, i.e. for *backtrack-free* search (Freuder 1982). It is first necessary to define some terms relating to the structure of a constraint graph. An *ordered constraint graph* is an arrangement of the constraint graph nodes in a fixed linear order, equivalent to maintaining a fixed instantiation order during tree search. The *width* at an ordered node is the number of constraint arcs that link it back to a previous node in the order. The width of an ordering is the maximum width of any of its nodes, and the width of a constraint graph is the minimum width of all its orderings. The following theorem (Freuder 1982) specifies the necessary conditions for backtrack-free search:

- There exists a backtrack-free search order for a CSP if the level of strong consistency is greater than the width of the graph.
- A search order is backtrack-free if the strong consistency level is higher than the width of the corresponding ordered constraint graph.

Consider a tree-structured constraint graph. If this type of graph can be made arc consistent, then subsequent search will be backtrack-free. This can be seen for the problem given in Figure 1, and the search tree of Figure 14b having enforced arc consistency (see Figure 14a).

Freuder provides algorithms to determine the width of a constraint graph and to enforce k-consistency. In addition Cooper (1989) presents a worst-case optimal algorithm to enforce k-consistency in a constraint graph. Unfortunately, these algorithms are very time consuming. In addition, to enforce a level of above 2-consistency it is often necessary to add new constraints to the graph (see Figure 15). The width of the graph must then be re-determined to see if a higher level of consistency is required, and so on. It is this high cost that makes it doubtful whether, overall, it is worth using this process to eliminate backtracking search.

4.5 *Directional pre-processing*

To improve the efficiency of pre-processing a CSP, Dechter and Pearl (1988) proposed *directional* pre-processing algorithms that exploit the order of instantiation that a tree search algorithm will subsequently use. This class of algorithms therefore avoids enforcing consistency for many constraints that are not necessary for the search process. Consider the ordered constraint graph corresponding to an instantiation order: it is sufficient to ensure that at any node the domains of any future variables it is linked to contain at least one value that is consistent with the current assignment. Assignments inconsistent with previous (instantiated) nodes will be avoided automatically by Backtrack.

This observation leads naturally to the development of a directional arc consistency algorithm (DAC). Variables are processed in the reverse order to which they will be instantiated by Backtrack, ensuring that all arcs that lead to the current level from past variables are made arc consistent. DAC revises each arc in the forward direction only once: for arc consistency on $arc(h, i)$ to be lost some values would have to be filtered from D_i. This will never happen since arc consistency forward from level i has already been achieved; no further values will be removed from D_i. This is a significant advantage over full arc consistency algorithms that must re-revise arcs that are possibly affected by the removal of any domain value. Indeed, when applied to a tree structured constraint graph, DAC will render the problem backtrack-free in

$O(d^2)$ (Dechter 1990), leading to a solution procedure of time complexity $O(nd^2)$.

Directional arc consistency extends naturally to directional path consistency (DPC). DPC directionally enforces path consistency for paths of length 2, thus ensuring path consistency for the whole constraint graph, with respect to the instantiation order. This process can continue towards k-consistency by enforcing just enough consistency at each node to ensure a backtrack-free search (*Adaptive Consistency* (Dechter and Pearl 1988)). ADAPT also processes the nodes in reverse instantiation order. The width of each node is examined and the appropriate amount of consistency with connected past variables is enforced.

DAC, DPC, and ADAPT were compared experimentally against Backtrack and Backjump (Dechter and Meiri 1989). Although the worst case analysis of Backtrack is exponential, the average case is much better It is therefore not generally cost-effective to do even the restricted pre-processing offered by ADAPT or DPC. However, DAC appears to perform the right amount of pre-processing to ameliorate the performance of Backtrack and will generally perform better even than Backjump.

5. Conclusion

This paper has presented the *constraint satisfaction problem* (CSP) and the fundamental techniques employed to solve this type of problem. Initially the different graphical means of representing a CSP were examined which can then be used to guide a solution method.

Tree search solution techniques were discussed, beginning with *Backtrack*. The *thrashing* problem was examined, with attempts to avoid it: *Backjump* and *Conflict-Directed Backjump* reduce the number of search nodes. Conversely, *Backmark* reduces the number of constraint checks. Tree search hybrids exploit savings made by both approaches. *Conflict Recording* extracts information from a dead end in the search. *Dynamic Backtracking* dynamically re-arranges the search order to avoid erasing potentially useful work during a backward jump. No single tree search algorithm uniformly maintains the best performance – CSP structure is crucial. A CSP of arbitrary size may be defined upon which Backtrack outperforms more sophisticated algorithms by making the 'right' choices without the overhead incurred by more informed algorithms.

Pre-processing a CSP eases the workload of subsequent tree search by filtering domain elements and/or modifying constraints. Enforcing *arc consistency* is a popular technique due to its relatively low cost. Enforcing *path consistency* modifies constraints to rule out inconsistent *pairs* of values.

k-consistency was discussed in the context of achieving a *backtrack-free* search. Finally, a fixed instantiation order may be exploited to avoid much consistency processing. Although easier problems are produced, the overhead of pre-processing combined with subsequent tree search may be more than the effort required by a single tree search algorithm applied to the initial CSP. Difficult CSPs benefit most.

The performance of the basic CSP solution techniques may be improved in various ways. For example, via *heuristics* which attempt to decide the best instantiation order of the variables or the assignment order of domain elements to a variable, to maximise the chance of finding a solution early. Tree search and consistency enforcing techniques may be integrated more closely to create *hybrids* which enforce a measure of consistency at each search node. The 'hard' nature of classical CSP (all constraints must be satisfied, each constraint is completely satisfied or completely violated) may be relaxed to create *flexible* constraints to better model the complexities of the real world. Such 'advanced techniques' are addressed in a companion review (Miguel and Shen 2001).

Acknowledgements

This work is partly supported by UK-EPSRC grant number 97305803.

Notes

[1] Consider that $D_i \times D_j$ is the same set of assignments as $D_j \times D_i$.

[2] It is assumed that, due to its nature, the FOR loop will terminate with h having a value 1 greater than the current level, which is compensated for by the algorithm.

References

Allen, J. (1984). Toward a General Theory of Action and Time. *Artificial Intelligence* **23**(2): 123–154.

Baker, A. B. (1994). The Hazards of Fancy Backtracking. In Proceedings of *The 12th National Conference on Artificial Intelligence*, 288–293.

Bessiere, C. (1994). Arc-consistency and Arc-consistency Again, *Artificial Intelligence* **65**: 179–190.

Bessiere, C., Freuder, E. C. & Regin, J. (1995). Using Inference to Reduce Arc Consistency Computation. In Proceedings of *The 14th International Joint Conference on Artificial Intelligence*, 592–598. Montreal, Quebec, Canada.

Bitner, R. & Reingold, M. (1975). Backtrack Programming Techniques. *Communications of the ACM* **18**(11).

Blum, A. & Furst, M. (1997). Fast Planning through Planning Graph Analysis. *Artificial Intelligence* **90**(1–2): 281–300.

Borning, A., Maher, M., Martindale, A. & Wilson, M. (1989). Constraint Hierarchies and Logic Programming. In Proceedings of *The 6th International Conference on Logic Programming*, 149–164.

Bruynooghe, M. (1985). Graph Coloring and Constraint Satisfaction, *Technical Report CW-44*. Department Computerwtenschappen, Kathioleke Universiteit Leuven.

Bruynooghe, M. (1985). Solving Combinatorial Search Problems by Intelligent Backtracking. *Information Processing Letters* **12**(1).

Chmeiss, A. & Jegou, P. (1996). Path-Consistency: When Space Misses Time. In Proceedings of *The 13th National Conference on Artificial Intelligence*, 196–201. Portland, Oregon.

Cooper, M. C. (1989). An Optimal k-Consistency Algorithm. *Artificial Intelligence* **41**: 89–95.

Dechter, R. (1987). A Constraint-Network Approach to Truth-Maintenance, *Technical Report R-80*. Cognitive Systems Laboratory, Computer Science Department, University of California.

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*, 276–285. John Wiley and Sons.

Dechter, R. & Dechter, A. (1988). Belief Maintenance in Dynamic Constraint Networks. In Proceedings of *The 7th National Conference on Artificial Intelligence*, 37–42. St Paul, Minnesota, USA.

Dechter, R. & Meiri, I. (1989). Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems. In Proceedings of *The 11th International Joint Conference on Artificial Intelligence*, 271–277. Detroit, Michigan, USA.

Dechter, R. & Pearl, J. (1988). Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence* **34**: 1–38.

Dechter, R. & Pearl, J. (1989). Tree Clustering for Constraint Networks. *Artificial Intelligence* **38**: 353–366.

Deville, Y. & van Hentenryck, P. (1991). An Efficient Arc Consistency Algorithm for a Class of CSP Problems. In Proceedings of *The 12th International Joint Conference on Artificial Intelligence*. Darling Harbour, Sydney, Australia.

Doyle, J. (1979). A Truth Maintenance System. *Artificial Intelligence* **12**: 231–272.

Fowler, G., Haralick, R., Gray, F. G. & Feustel, C. (1983). Efficient Graph Automorphism by Vertex Partitioning. *Artificial Intelligence* **21**: 245–269.

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

Freuder, E. C. (1978). Synthesizing Constraint Expressions. *Communications of the ACM* **21**(11): 958–966.

Freuder, E. C. (1982). A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM* **29**(1): 24–32.

Frost, D. & Dechter, R. (1994). Dead-End Driven Learning. In Proceedings of *The 12th National Conference on Artificial Intelligence*, 294–300.

Gaschnig, J. (1979). Performance Measurement and Analysis of Certain Search Algorithms, *Technical Report*. Department of Computer Science, Carnegie-Mellon University.

Ginsberg, M. L. (1993). Dynamic Backtracking. *Journal of Artificial Intelligence Research* **1**: 15–46.

Ginsberg, M. L. & McAllester, D. A. (1994). GSAT and Dynamic Backtracking. In Proceedings of *The 4th International Conference on Principles of Knowledge Representation and Reasoning*.

Han, C. & Lee, C. (1988). Comments on Mohr and Henderson's Path Consistency Algorithm. *Artificial Intelligence* **36**: 125–130.

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

Jonsson, A. K. & Ginsberg, M. L. (1993). Experimenting with New Systematic and Non-systematic Search Techniques. In Proceedings of *The AAAI Symposium on AI and NP-Hard Problems*.

de Kleer, J. (1989). A Comparison of ATMS and CSP Techniques. In Proceedings of *The 11th International Joint Conference on Artificial Intelligence*, 290–296. Detroit, Michigan, USA.

Kolbe, T. H. (1998). Constraints for Object Recognition in Aerial Images – Handling of Unobserved Features. In Proceedings of *The 4th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 1520, 295–309. Pisa, Italy: Springer.

Kondrak, G. & van Beek, P. (1995). A Theoretical Evaluation of Selected Backtracking Algorithms. In Proceedings of *The 14th International Joint Conference on Artificial Intelligence*, 541–547. Montreal, Quebec, Canada.

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

Kumar, V. (1992). Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*: 32–44.

McDermott, D. (1991). A General Framework for Reason Maintenance. *Artificial Intelligence* **50**: 289–329.

McGregor, J. (1979). Relational Consistency Algorithms and Their Applications in Finding Subgraph and Graph Isomorphism. *Information Sciences* **19**: 229–250.

Mackworth, A. (1977). Consistency in Networks of Relations. *Artificial Intelligence* **8**(1); 99–118.

Mackworth, A. (1977b). On Reading Sketch Maps. In Proceedings of *The 5th International Joint Conference on Artificial Intelligence*, 598–606, Cambridge, Massachusetts, USA.

Mackworth, A. (1992). Constraint Satisfaction Problems. In *Encyclopedia of Artificial Intelligence*, 285–293. John Wiley and Sons.

Mackworth, A. & Freuder, E. (1985). The Complexity of Some Polynomial Network-Consistency Algorithms for Constraint-Satisfaction Problems. *Artificial Intelligence* **25**: 65–74.

Meseguer, P. (1989). Constraint Satisfaction Problems: An Overview, *AI Communications* **2**(1): 3–17.

Miguel, I. & Shen, Q. (1998). Extending Qualitative Modelling for Simulation of Time-Delayed Behaviour. In Proceedings of *The 12th International Workshop on Qualitative Reasoning*, 161–166. Cape Cod, Massachusetts, USA.

Miguel, I. & Shen, Q. (2001). Solution Techniques for Constraint Satisfaction Problems: Advanced Approaches. *Artificial Intelligence Review* **15**: 267–291.

Mohr, R. & 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.

Nadel, B. A. (1989). Constraint Satisfaction Algorithms. *Computational Intelligence* **5**: 188–224.

Prosser, P. (1989). A Reactive Scheduling Agent. In Proceedings of *The 11th International Joint Conference on Artificial Intelligence*, 1004–1009. Detroit, Michigan, USA.

Prosser, P. (1993). Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* **9**(1).

Rit, J. F. (1986). Propagating Temporal Constraints for Scheduling. In Proceedings of *The 5th National Conference on Artificial Intelligence*, 383–386, Philadelphia, Pennsylvania, USA.

Rodosek, R. & Wallace, M. (1998). A Generic Model and Hybrid Algorithm for Hoist Scheduling Problems. In Proceedings of *The 4th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 1520, 385–399. Pisa, Italy: Springer.

Rosenfeld, A., Hummel, R. & Zucker, S. (1976). Scene Labeling by Relaxation Operations. *IEEE Transactions on Systems, Man, and Cybernetics* **6**: 420–433.

Schiex, T. & Verfaillie, T. (1994). Stubbornness: A Possible Enhancement for Backjumping and Nogood Recording. In Proceedings of *The 11th European Conference on Artificial Intelligence*, 165–169.

Schiex, T., Regin, J., Gaspin, C. & Verfaillie, G. (1996). Lazy Arc Consistency. In Proceedings of *The 13th National Conference on Artificial Intelligence*, 216–221. Portland, Oregon.

Shapiro, L. & Haralick, R. (1981). Structural Descriptions and Inexact Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **3**(5): 504–518.

Shen, Q. & Leitch, R. (1993). Fuzzy Qualitative Simulation. *IEEE Transactions on Systems, Man, and Cybernetics* **23**(4): 1038–1061.

Stallman, R. M. & Sussman, G. J. (1977). Forward Reasoning and Dependency-Directed Backtracking in a System for Computer Aided Circuit Analysis. *Artificial Intelligence* **9**: 135–196.

Tsang, E. P. K. (1987). The Consistent Labeling Problem in Temporal Reasoning. In Proceedings of *The 6th National Conference on Artificial Intelligence*, 251–255.

Tsang, E. P. K. (1993). *Foundations of Constraint Satisfaction*. Academic Press: London and San Diego.

Ullman, J. R. (1976). An Algorithm for Subgraph Isomorphism. *Journal of the ACM* **23**: 31–42.

Wallace, R. J. (1993). Why AC − 3 is Almost Always Better than AC − 4 for Establishing Arc Consistency in CSPs. In Proceedings of *The 13th International Joint Conference on Artificial Intelligence*, 239–245. Chamberg, France.

Waltz, D. (1975). Understanding Line Drawings of Scenes with Shadows. In Winston, P. H. (ed.) *The Psychology of Computer Vision*, 19–91. McGraw-Hill.

Zabih, R. & McAllester, D. (1988). A Rearrangement Strategy for Determining Propositional Satisfiability. In Proceedings of *The 7th National Conference on Artificial Intelligence*, 155-160. St Paul, Minnesota, USA.