A Truth Maintenance System for Supporting Constraint-Based Reasoning

Vasant DHAR
Department of Information Systems, New York University, New York, NY 10003, USA

Many types of choice problems that arise in design, be it architectural/engineering design or the design of economic models, can be formulated as constraint satisfaction problems (CSPs). In general, TMSs are a useful computational mechanism for maintaining consistent beliefs or assumptions in problems characterized by a set of constraints. They also enable a problem solver to explore a search space more efficiently by recording the causes of failed partial solutions, and provide a limited explanation capability since reasons for beliefs are recorded explicitly. In this paper we describe a Truth Maintenance System (TMS) whose architecture has been motivated by the structure of a commonly occurring type of CSP. We show that by exploiting structural features of dependency constraints involved in CSPs and adopting a certain delineation of responsibilities between the TMS and a problem solver, considerable simplicity in the TMS architecture and efficiency in its status assignment algorithms is achieved. We also compare how reasoning systems designed for solving constraint satisfaction problems using our specialized TMS differ from those using other truth maintenance models such as McAllester's RUP and de Kleer's ATMS.

Vasant Dhar is an Associate Professor of Information Systems at the Leonard N. Stern School of Business, New York University, where he has been on the faculty since 1983. His primary research interests are in the empirical and theoretical aspects of Artificial Intelligence. Much of his research involves empirical investigation of problem-solving processes in domains involving design, planning, and decision-making, and the development of representational formalisms needed to build intelligent systems in these domains. He has written a number of articles on knowledge representation and heuristic search – particularly in the context of truth maintenance systems, and methodological issues in the development of knowledge-based systems.

North-Holland
Decision Support Systems 5 (1989) 379–387

1. Introduction

Design problems constitute an important class of problems in Artificial Intelligence. Many such problems are characterized by the need to synthesize an artifact from discrete model fragments in the context of problem-specific constraints. Such design problems arise in various domains such as architecture, engineering, and operations research. While there is no general theory of design that characterizes all design problems, the synthesis process typically involves some form of "constraint satisfaction", and the problem is often referred to as a constraint satisfaction problem (CSP). The CSP has been described formally by Knuth (1975), Mackworth (1977), Gashnig (1979), Haralick and Shapiro (1980), and Dechter and Pearl (1988).

There are many techniques for solving constraint satisfaction problems such as optimization and heuristic search methods. For problems involving discrete data, optimization techniques such as integer programming can be computationally prohibitive. Such techniques have additional drawbacks in that there is no explanation, and incremental model revision is difficult since the formulation tends to be extremely brittle (i.e. translating real-world changes into the binary algebraic formulation can be difficult) (Dhar and Ranganathan (1989)). This can be a serious limitation for many problems where even though an initially optimal solution may be desirable, decisions can be constantly subject to change forcing decision makers to abandon optimality and make incremental changes based on pragmatic grounds. For such problems, once a solution is in place, decision makers make incremental decisions in the context of an evolving set of problem data.

Truth Maintenance Systems (TMS) appear to be well suited for managing the reasoning process involved in constraint satisfaction problems involving discrete data. Typically, such data consists of sets of competing alternatives or assumptions that on which a solution is based. From a decision

0167-9236/89/$3.50 © 1989, Elsevier Science Publishers B.V. (North-Holland)

support standpoint, TMSs can enable a user to analyze incrementally the impacts of changes on a solution or to explore alternative solutions. They can also be coupled with optimization solvers to conduct post-optimal what-if analyses. From a computational standpoint, smart algorithms (such as dependency directed backtracking) can exploit the data dependencies and failures recorded by the TMS to prune the search space.

A description and example of a constraint satisfaction problem involving discrete data that occurs commonly in industrial engineering and operations research was provided in Dhar and Croker (1988). Basically, this problem involves a selection of one alternative from each of a set of competing alternatives called choice sets. The set of alternatives selected are required to satisfy various dependency constraints defined over the choice sets. For this type of problem, of which project planning and scheduling are common exemplars, it is desirable to come up with one or a few "good" solutions and to modify these incrementally as the constraints and sets of alternatives (which in hypothetical reasoning problems represent "assumptions") change.

In this paper, we provide a description of our TMS that works under a CSP solver. We describe the relationship between the TMS and the problem solver, delineating the responsibilities and functionality of each component. The TMS also has several novel features. Firstly, by exploiting structural features of terms involved in the dependencies, the size of the dependency network is small; the number of datum nodes is at most equal to (and usually less than) the number of unique terms in the dependency constraints. Secondly, the dependency network is compiled from the constraints when the problem is specified and remains fixed in size. [1] Thirdly, the justifications of TMS nodes are never modified as problem solving proceeds, they are only evaluated whenever necessary in order to propagate constraints or detect contradictions. As we shall see, these features simplify considerably the status assignment algorithms of the TMS and lead to a clear separation of responsibility between it and the problem solver.

[1] However, the size of untenable sets of alternatives (called NOGOODS after Stallman and Sussman (1977)) grows; we discuss NOGOODS in section 3.5.

The remainder of this paper is organized as follows. In the following section we provide a formal description of the type of problem that has motivated the design of our TMS. In section 3 we describe the structure and function of the TMS. In section 4 we discuss how reasoning systems constructed with other reason maintenance systems differ from that described in this paper.

2. Dependency Constraints

In order to facilitate an explanation of the architecture of our TMS we will discuss it in terms of one of the types of CSP constraints that it models, dependency constraints. We will not describe here the other types of knowledge involved in the CSP since these are not relevant to the TMS.

A CSP is characterized by a finite number of discrete sets that we call choice sets and a set of constraints defined over these choice sets. A solution to a CSP is a combination of alternatives, one from each choice set, such that together they satisfy each of the constraints. More formally, let $X = \{X_1, X_2, \ldots, X_n\}$ be a set of choice sets. Corresponding to each X_i is a set $D_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,p_i}\}$ that represents the domain of competing alternatives of X_i. Associated with each X_i is a set of attributes that characterize each of the alternatives in D_i.

A constraint for the set of choice sets X is a restriction on the combination of alternatives that can be selected from the choice sets in X. That is, if C_i is a constraint, then $C_i \subseteq D_1 \times D_2 \times \ldots \times D_n$. A combination of selections from the choice sets in X are said to satisfy constraint C_i if it forms a tuple in C_i in the obvious way.

Let $C = \{C_1, C_2, \ldots, C_m\}$ be a set of constraints for the set X. A solution to the CSP characterized by X and C is a sequence of selected alternatives $\langle a_{11,j_1}, a_{2,j_2}, \ldots, a_{n,jn}\rangle$ where $a_{i,j_i} \in X_i$ that satisfies each of the constraints in C. Such a solution is also called a satisficing assignment for the CSP.

Constraints can be expressed in several ways. One type of constraint, a dependency constraint is expressed in terms of relationships that must hold between the attribute values associated with the selected alternatives. A dependency constraint is expressed as $t_1, t_2, \ldots, t_{n-1} \rightarrow t_n$ and states that if

a combination of selected alternatives from the choice sets satisfies each of the Boolean-valued terms t_1 through t_{n-1} (i.e. they are all true), the antecedent, then they must also satisfy the term t_n, the consequent. A special type of dependency constraint, called a premise constraint has no antecedent and must always hold.

In one general type of dependency (and premise) constraint that arises in problems modeled by us, the term t is a relational expression of the form $\alpha_1 \theta \alpha_2$, where each α_i is an expression involving choice set attributes and/or constants (at least one of α_1 and α_2 must involve an attribute), and θ is a relational operator. If α_i involves an attribute, then it designates the attribute value corresponding to the alternative selected in the associated choice set. (See Dhar and Croker (1988).)

3. The Reasoning System: Problem Solver and TMS

In the reasoning system we have designed for solving CSPs, the task of the problem solver is to make assignments, one at a time, that satisfy the constraints. After each assignment, the task of the TMS is to perform constraint propagation. This involves determining which constraint terms must necessarily hold, given the current set of assignments. This process constrains the selection that the problem solver can make in the next cycle. In effect, the overall reasoning process in which the TMS participates can be characterized as a "heuristic-deduction" cycle where a heuristic selection is made by the problem solver, and deductions are made by the TMS. When no more deductions are possible, the constraint set is said to be relaxed. If no contradictions are detected by the TMS, control passes back to the problem solver and the cycle resumes.

If constraint violations are detected, the sets of assignments contributing to the constraint violations are computed by the TMS and handed back to the problem solver, which decides what retractions to make in order to undo the violation. [2] These sets, also called NOGOODS, are maintained by the TMS. The problem solver is prohibited from attempting assignments that extend any set of assignments in NOGOODS.

[2] This is referred to as dependency directed backtracking.

Retraction also becomes necessary if all possible selections within a choice set have been attempted in a certain context without success. This happens when all selections lead to extensions of a nogood.

In addition to knowledge about what to retract in backtracking situations, the problem solver also contains knowledge about how to extend a partial solution. This knowledge is reflected in an ordering of choice sets, and an ordering of alternatives within each choice set. With respect to the latter, the ordering can be expressed in terms of attribute values (i.e. "minimize cost/performance" where cost and performance are choice set attributes). In addition, the problem solver maintains state information, specifically, whether an alternative has been selected within a choice set, and if so, what that alternative is. This state information is used by the problem solver and is also readable by the TMS.

The problem solver and TMS are connected via intermediate nodes that serve as "entry points" into the dependency network. Each choice set, represented as a structured object, has an associated entry node which is linked to all TMS nodes (defined below) that refer to that choice set.

The basic data structure used in the TMS is a constraint term node. Each of these nodes corresponds to a term that occurs in one or more dependency constraints. Nodes representing antecedent terms are referred to as antecedent nodes whereas those referring to consequent terms are referred to as consequent nodes. A node can be both an antecedent node and a consequent node. A node has the following structure:

$\langle$*term-exprn*, *term-value*, *csets*, *sl*, *nl*, *s*, *n*$\rangle$,

where *term-exprn* corresponds to a term in a dependency constraint; *term-exprn* contains access functions set up at compile time which can access state information maintained by the problem solver. In this way, the TMS has read access to the state maintained by the problem solver, *term-value* is the value of *term-exprn* (true, false, or unknown), *csets* is a set of choice sets referenced in *term-exprn, sl* is a set of sets of nodes (terms), where each set independently supports belief in the term-expression corresponding to the node. Specifically, the relationship specified by *term-exprn* must hold (be true) if all terms in any set of *sl* hold. This is similar to Doyle's (1979) support-list

justification; *nl* is a set of sets of terms, where each set independently supports belief in the negation of the term-expression corresponding to the node, *s* is a set of nodes, each of which can hold if *term-exprn* is true, *n* is a set of nodes, each of which can hold if *term-exprn* is false.

The set of instances of term-node defines a dependency network.

A node has a valid positive justification if any expression in *sl* is satisfied. An expression is satisfied if it is non-negated and evaluates to true or if it is negated and evaluates to false. A node has a valid negative justification if any expression in *nl* is satisfied. Having the *nl* and *n* avoids the need for a separate node for the negation of the term-expression. Also, associating the *sl* and *nl* with the same expression makes it easy to detect constraint violations.

An interesting property of the network is that its structure is completely specified in terms of the set of dependency constraints and thus does not change with the changing problem state of the problem solver. Typically, the number of nodes in the network is small, depending on the number of unique constraint terms. The justification structure of each node is also static, only the truth value of its expression changes with problem state. The TMS deals only with constraint terms, which are general boolean-valued expressions defined over the choice sets. In contrast, the selection datum is manipulated by the problem solver. This separation of responsibilities leads to considerable simplicity and efficiency in the status assignment algorithms of the TMS and in the architecture of the overall reasoning system.

It should be noted that other than the *term-value* slot, the term node structure is static and state-independent; all state information is maintained by the problem solver. The term-value slot is useful in detecting certain types of contradictions (i.e. when the true value of *term-exprn* becomes the negation of the recorded value) and for efficiency purposes (i.e. recognizing when propagation is redundant).

3.1. Non-monotonic inference

When the constraint set is relaxed, a selection is attempted by the problem solver in some choice set, based on domain-specific knowledge. If a selection is made, the problem solver notifies all nodes involving that choice set in their term-expressions. Each of these nodes computes the value of its term-expression in order to check whether constraint propagation should be initiated.

If the expression evaluates to true (false) and does not have a valid positive (negative) justification, the node can be viewed as having non-monotonic support. However, the non-monotonic justification is not explicitly recorded for the following reasons.

From a conceptual standpoint, recording the non-monotonic justification would imply that the selection was made because of a lack of belief in its competitors (or because one cannot prove its negation). While it is certainly true that a specific selection can be assigned to a choice set only as long as none of its competitors are, this situation does not represent the reason for making the selection. The reason for making the selection is that it is preferred to other allowable alternatives, knowledge about preference being part of the problem solver. Therefore such information is not recorded in the TMS.

3.2. Constraint Propagation

Constraint propagation is initiated at all TMS nodes related to the choice set where a selection is made. If the *term-exprn* of the node evaluates to true (false), the TMS follows the $s(n)$ links to the consequent nodes to check whether they have valid justifications. For each node with a valid positive justification the TMS labels *term-value* true whereas for a valid negative justification it is labeled false. This process is repeated recursively for each node with a valid justification until the constraint set is relaxed.

It should be noted that as a consequence of constraint propagation, nodes may have term values of true (false) even though their term-expressions might be unknown. This establishes a restriction for the problem solver in the selections that are admissible within the context established by the current set of assignments.

3.3. Contradictions

Contradictions can arise in two ways: if *term-exprn* evaluates to true (false) and a node has a valid negative (positive) justification, or if a

node has valid positive and negative justifications simultaneously.

3.4. Dependency Analysis and Belief Revision

Dependency analysis is invoked when a contradiction is detected. However, it should also be noted that backtracking is required if all possible alternatives from a choice set have been attempted unsuccessfully. This happens when all alternatives lead to extensions that are known to be untenable (nogood). In this situation the TMS passes back the set of nogoods associated with each of the attempted selections, which is analyzed by the problem solver using domain-specific knowledge.

Dependency analysis involves finding all combinations of support for the node where a contradiction is detected. This involves an analysis of its *sl* and *nl*. Specifically, for each node corresponding to the expression in a satisfied sublist, all combinations of support are found recursively. Dependency analysis results in an AND/OR graph indicating what combinations of choices can be retracted to fix the violation. This is best illustrated through an example.

Consider the following dependency constraints where lower case letters denote terms:

$a, b \rightarrow c,$

$e, c \rightarrow d,$

$b, e \rightarrow \sim d.$

For simplicity, assume that each term references one choice set, which we shall designate with the corresponding uppercase letter. If all terms in the constraints above evaluate to true, a contraction is detected at the node corresponding to d and $\sim d$. Let us further assume that the alternative selected from choice set D is such that the term d is true. In this situation, dependency analysis would yield the AND/OR graph of fig. 1. The subgraph on the left indicates combinations of retractions that will invalidate support for d. Likewise, the subgraph on the right shows the two ways of invalidating support for $\sim d$. Either can be used to fix the contradiction. The former requires retracting selections in D and E, or D and C and one of A and B. The latter requires retracting a selection in either E or B. The decision about what to retract is that of the problem solver. The TMS has no knowledge about what retractions are the most prudent in a given context.

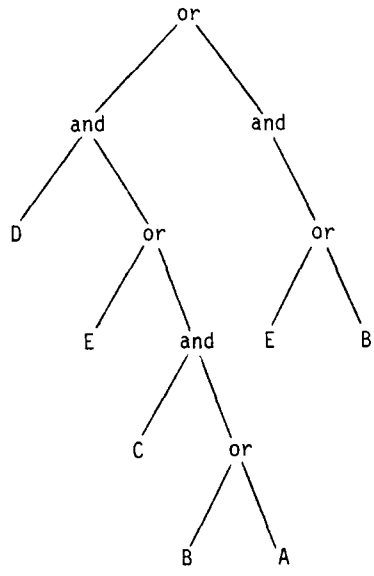

Fig. 1.

3.5. NOGOODS

A special set, called NOGOODS, contains sets of choices that in conjunction are known to be untenable. This set, corresponding to the combinations constructed by dependency analysis (the AND/OR graph above), is the only dynamic data structure maintained in the reasoning system.

Determining whether a partial solution is nogood involves checking whether the current set of assignments is the same as any element of NOGOODS. This test becomes increasingly expensive as the NOGOODS set grows. It is therefore important to have a representation that is efficient in time and space.

There are several possible ways of representing the NOGOODS and performing the subset test. The representation we employ currently is a discrimination net that has an ordering imposed on it based on choice sets. For example, denoting choices by lower case letters (where the same lower case letters indicate the choice set), the NOGOODS set consisting of

((a1 b1 c1)(a1 b1 c2)(a1 b2 c1)(a1 b2 c2)
(a1 b2 c3))

would be represented in LISP-like notation as

(a1 (b1(c1 c2))
(b2(c1 c2 c3))).

Checking to see whether a set of assignments is a nogood involves imposing an order on the choices (corresponding to the order used in the NOGOODS structure) and traversing the discrimination net top-down. A similar procedure is used to extend the structure when a new nogood is found.

Another possible scheme is one used in de-Kleer's (1986) ATMS. A bit-vector is used to represent each nogood set where each possible assignment is given a unique position; a one bit indicates the presence of the corresponding assignment in the set. Set1 is a subset of set2 if the result of and'ing the bit-vector of set1 with the complement of set2 is zero. This scheme is efficient since the operations are performed by hardware. In general, however, if the bit-vectors are long, the advantages of this scheme are reduced.

We are seriously considering implementing the nogood check using the Rete match algorithm (Forgy, 1982) which is an efficient method for comparing a large collection of patterns to a large collections of objects. In the production system context, patterns are rules and objects are working memory elements which represent the state of the system. From the patterns, a network is constructed that can maintain state information. The efficiency of Rete is due to the fact that state changes very slowly with each problem solving cycle, and the effects of the state change (i.e. determining which new patterns are enabled and which already enabled patterns are disabled) can be computed incrementally. The details of Rete are beyond the scope of this paper; however, we should point out that the subset test involved in checking for nogoods is isomorphic to the Rete match problem if one considers NOGOODS as the patterns and partial assignments (which change slowly) as the objects.

4. Relationship to Other Work

Constraint satisfaction problems have been dealt with extensively in the OR and AI literature. In the former, the emphasis has been on developing optimizing problems solvers such as the Simplex and its variants and Karmarkar's (1984) algorithm. In AI, the emphasis has been around building languages for expressing various types of constraints and drawing useful inferences from them (de Kleer and Sussman, 1978; Steele and Sussman, 1978). TMSs have emerged as a powerful general computational model for various types of default, nonmonotonic, or assumption-based reasoning with constraints.

It is possible to model special (linear) cases of the constraint satisfaction problem we have described as optimization problems. It is also possible to model the problem using other TMSs in conjunction with an appropriately designed problem solver. In order to model it as an optimization problem, each alternative in a choice set would be characterized as a 0–1 variable. The problem constraints can be converted into clauses, each expressible as an algebraic constraint (for a detailed discussion of the relationship between symbolic and algebraic constraints the reader is referred to Hinton (1977) and Hooker (1988)). The solver would then use a combination of the Simplex (or equivalent) to solve the relaxed problem and branch-and-bound or cutting-plane techniques.

In the remainder of this section, we describe how it is possible to model the constraint satisfaction problem we have described using other TMSs – such as Doyle's TMS (Doyle, 1979), RUP (McAllester, 1982), PROTEUS (Petrie et al., 1987), or the ATMS (de Kleer, 1986). Using these systems requires a recasting of the problem and a carefully designed problem solver. We shall comment on some of the differences between our reasoning system and those that might result by using two of these well known TMSs, namely RUP – a justification-based TMS, and ATMS – an assumption-based TMS.

4.1. Comparison with RUP

RUP's TMS performs deduction using propositional logic. RUP's primitive data structures are nodes (corresponding to RUP terms), clauses, noticers and queues. Nodes contain noticers which trigger on events in an assertional database. More precisely, they trigger in response to the changes in the truth value of the node, not unlike the constraint propagation process in our TMS. However, unlike our TMS nodes which are restricted to performing constraint propagation when their term-expression value changes, RUP noticers can contain arbitrary LISP forms. While this gives the designer considerable flexibility, de Kleer (1986b) suggests that this flexibility can be dangerous in that it can lead to design where

exhaustivity in search might be sacrificed inadvertently.

In order to implement our model of the constraint-based reasoning process in RUP, dependency constraints would be implemented as noticers, with each constraint term corresponding to an antecedent of a noticer (a RUP term). Each selection being asserted would also be a RUP term, always accessible by the TMS. The overall problem solving cycle with a RUP-based problem solver would proceed as follows: the problem solver would "assert" a selection and collect all noticers whose antecedent patterns' enabling conditions were satisfied; if so, a term corresponding to the consequent would be assigned a truth value, which could continue to trigger other noticers. When no more noticers trigger, the problem solver would make another selection, and the cycle would repeat.

The detection of contradictions would require additional machinery. RUP's TMS is basically designed to detect logical contradictions. Specifically, contradictions are detected if all disjuncts of a clause evaluate to false (see McAllester, 1982). In contrast, we are concerned with general constraint violations. To illustrate, if the relational expression (also viewable as a clause) "engine horsepower must be greater than 75" is true, a contradiction would be detected if its negation is also true. Suppose that the above expression is true because of constraint propagation, and that a selection is attempted in the engine choice set whose horsepower value is 40. Our TMS detects this violation automatically (by recognizing that the truth value of the term-exprn becomes false whereas term-value is true). Additional noticers would be required to recognize this type of contradiction in RUP (and other TMSs), i.e. to enable the system to recognize that a value of 40 is inconsistent with the constraint that it should be greater than 75.

Special machinery would also have to be designed and "hooked" into the TMS for performing dependency analysis. Recall that in our TMS, if a term-value is true but no selections have been made from its corresponding choice sets, the term-label evaluates to UNKNOWN and the corresponding choice set is not inserted into the AND/OR structure that is handed back to the problem solver for making retractions. To accomplish this using RUP, a user supplied routine would have to be hooked into the TMS to perform this computation (RUP allows for extending its node structure via its plist slot, so this type of extension of its basic machinery is possible).

4.2. Comparison with ATMS

de Kleer's ATMS (1986a) is in many ways different from all other truth maintenance systems. In addition to justifications, each datum is labeled with the sets of assumptions (representing the contexts) under which it holds. It is therefore easy to determine whether a datum holds in a given context. In addition, there is no necessity that the overall database be consistent; rather, the idea is that the consequences of multiple, possibly contradictory, worlds can be pursued simultaneously.

In ATMS language, each choice set would be represented as a disjunction of assumptions. The primitive, control$\{C_1, C_2, \ldots, C_n\}$, specifies one of disjunction; additionally, it incorporates control information specifying the order in which assumptions will be explored. Similarly, a set of assignments corresponds to an ATMS environment. The set of all combinations of assignments can be viewed as a lattice structure. In the absence of any constraints, all vertices in the lattice would represent partial or complete solutions. Constraints have the effect of eliminating parts of the lattice as untenable. In fact, the process of constraint satisfaction can be viewed in terms of a cycle where the problem solver hands one constraint at a time to the TMS which goes about progressively eliminating parts of the lattice as untenable. Problem solving terminates when there are no more constraints to be imposed. This is the key idea behind de Kleer's consumer (de Kleer, 1986b) which is "run" once, that is, it hands over a justification to the TMS, and is discarded.

ATMS consumers are attached to TMS nodes or classes of nodes. Since class consumers are a more parsimonious encoding of knowledge, constraints would be encoded in terms of these consumers. Each constraint term would represent a node class. A term referencing n choice sets having an average of k alternatives could therefore generate up to k^n consumers.

The ATMS can also be viewed as providing a 0–1 integer programming type of formulation, with assumptions being boolean variables. The

environment lattice is basically the search space containing all possible sets of assignments; each justification introduced to the TMS essentially imposes a "cut" on the (evolving) feasible region. After all justifications have been specified (in other words, after all consumers have been run), the ATMS (environment lattice) contains all possible solutions to the problem.

While the ATMS architecture is well suited to pursuing multiple solutions, generating a single solution can require considerable additional effort. As de Kleer (1986b) points out, fewer required solutions call for a greater control effort. This consideration is particularly relevant for many decision-making problems which require a single "good" solution (which the decision-maker should then be able to explore incrementally or modify in response to changing assumptions or constraints). While the ATMS scheduler does maintain a "single current environment" which is guaranteed to satisfy every control disjunction and provide a single solution according to the preference order specified in each disjunction, this situation in effect makes it function like a justification based TMS. Essentially, the overhead involved in using the ATMS for such problems could outweigh its benefits.

5. Conclusion

Our research builds on earlier work of the Kleer and Sussman (1978) and Steele and Sussman (1978) on algebraic constraint propagation. We allow for arbitrarily complex expressions in the dependency constraints. We have also shown how a particular type of constraint based reasoning problem solver is integrated with a dependency network formulated in terms of the expressions that comprise part of the problem definition, and the responsibilities of each of the components in the overall reasoning system.

In our implementation, a pre-processor transforms constraints into a dependency network structure and establishes the appropriate linkages between the TMS and the problem solver. The contents of choice sets, the alternatives, do not have to be completely specified when the dependency network is set up but can be added/deleted incrementally. In the dependency network, the justification structure is not manipulated, but only evaluated as selections are made. This leads to considerable simplicity and streamlining in the constraint propagation and belief maintenance machinery.

In terms of functionality for decision support, the TMS is useful in analyzing the repercussions of making various types of incremental changes. For example, it is easy to tighten or relax constraints, force the system to make specific choices in choice sets, or change the criteria for ordering selections within choice sets. In addition, it is easy to add new alternatives or delete existing ones, or to change the values of attributes associated with alternatives. This latter situation is typical of many real-world applications where alternatives are continually emerging, becoming foreclosed, or changing. In such situations, it is often desirable to analyze what parts of an existing model must be modified. Dhar and Croker (1988) provide examples of several types of what-if analyses that are useful to decision makers using planning/design models whose validity hinges on assumptions and constraints that are continually subject to change.

Finally, as we pointed out at the outset of this paper, for some problems it might be feasible to obtain an initially optimal solution which is then be subject to modification as the problem data change. For such problems the optimizer could serve as a search engine, communicate the results to the TMS, and allow a user to conduct analyses of the optimal solution using the dependency network. We are currently in the process of integrating our TMS with one such mathematical programming package.

References

Croker, A., Dhar, V., and McAllester, D., 1988, Dependency Directed Backtracking for Generalized Satisficing Assignment Problems, forthcoming.

de Kleer, J., 1986a, An Assumption-based TMS, Artificial Intelligence 28, no. 2, March.

de Kleer, J., 1986b, Problem Solving with the ATMS, Artificial Intelligence 28, no. 2, March.

de Kleer, J., and Sussman, G., 1978, Propagation of Constraints Applied to Circuit Synthesis, MIT AI Lab Memo 485, September.

Dhar, V., and Pople, H.E., 1987, Rule-Based Versus Structure-Based Models for Generating and Explaining Expert Behavior, Communications of the ACM 30, no 6, June.

Dhar, V., and Croker, A., 1988, Knowledge-Based Support for Business Problems: Issues and a Solution, IEEE Expert, Volume 3, No. 1, Feb.

Dhar, V., and Ranganathan, N., 1989, Experiments with an Integer Programming Formulation of an Expert System, Communications of the ACM, forthcoming.

Dechter, R., and Pearl, J., 1988, Network-Based Heuristics for Constraint-Satisfaction Problems, Artificial Intelligence 34, pp. 1–38.

Doyle, Jon., 1979. A Truth Maintenance System, Artificial Intelligence 12, June.

Forgy, C.L., 1982, Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence 19, pp. 17–37.

Gashnig, J., 1979, Performance Measurement and Analysis of Certain Search Algorithms, Ph.D. Dissertation, Technical Report CMU-CS-79-124, Department of Computer Science, Carnegie-Mellon University.

Haralick, R.M., and Shapiro, L., 1980, The Consistent Labeling Problem: Part I, IEEE Transactions on Pattern Analysis and Machine Intelligence 2, no 3.

Hinton, G.E., 1977, Relaxation and its Role in Vision, Ph.D Thesis, University of Edinburgh.

Hooker, J.N., ,1988, A Quantitative Approach to Logical Inference, Decision Support Systems 4, no. 1, March.

Karmarkar, N., 1984, A New Polynomial-time Algorithm for Linear Programming, Combinatorica 4.

Knuth, D., 1975, Estimating the Efficiency of Backtrack Programs, Mathematics of Computation 24 (129) pp. 121–136.

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

McAllester, D., 1980, An Outlook on Truth Maintenance, AI Laboratory Memo 551, August.

McAllester, D., 1982, Reasoning Utility Package, AI Laboratory Memo 667, April.

Petrie, C., 1987, Revised Dependency-Directed Backtracking for Default Reasoning, Proceedings of the Sixth National Conference on Artificial Intelligence AAAI-87, June.

Stallman, Richard and Sussman, Gerald., 1977, Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, Artificial Intelligence 9, no. 2, October, pp. 135–196.

Steele, G., and Sussman, G., 1978, Constraints, MIT AI Lab Memo 502, May.