A KNOWLEDGE REPRESENTATION FOR CONSTRAINT SATISFACTION PROBLEMS

by

Albert E. Croker

and

Vasant Dhar
Leonard N. Stern School of Business
Information Systems Department
New York University
40 West 4th Street
New York, NY 10003

Revised August 1990

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-90-9

This paper replaces Working Paper No. 191 entitled, "A PROBLEM-SOLVER/TMS ARCHITECTURE FOR GENERAL CONSTRAINT SATISFACTION PROBLEMS."

Abstract

In this paper we present a general representation for **constraint satisfaction problems** (CSP) and a framework for reasoning about their solution that unlike most constraint-based relaxation algorithms, stresses the need for a *"natural"* encoding of constraint knowledge and can facilitate making inferences for propagation, backtracking, and explanation. The representation consists of two components: a *generate-and-test* problem solver which contains information about the problem variables, and a *constraint-driven* reasoner that manages a set of constraints, specified as arbitrarily complex Boolean expressions and represented in the form of a **constraint network**. This constraint network: incorporates control information (reflected in the syntax of the constraints) that is used for constaint propagation: contains dependency information that can be used for explanation and for dependency-directed backtracking; and is incremental in the sense that if the problem specification is modified, a new solution can be derived by modifying the existing solution.

1. Introduction

Many problems can be formulated as constraint satisfaction problems. Expressing problems as constraint satisfaction problems requires a decomposition of the problem into parts, the generation or retrieval of alternatives for these parts, and the coordination of solutions for each part into an integrated whole (Simon, 1973). This general characterization applies to a variety of problems ranging from the design of a fugue (Reitman, 1965), a house (Alexander, 1964), or an engineered artifact (Simon, 1973), to that of a business plan (Dhar and Pople, 1987). Domain expertise is involved in deciding how best to decompose a problem into parts, in generating alternatives, recognizing constraints among the alternatives, and in resolving conflicts among them in a way that least impairs the quality of the overall solution. Solving such problems involves *constraint satisfaction.*

In this paper we present a general representation for *constraint satisfaction problems.* Each of these problems are characterized in terms of discrete sets of alternatives, which we call *choice sets*, and a set of constraints that are defined in terms of the choice sets. A solution to a constraint satisfaction problem is defined as a set of alternatives, one from each of the characterizing choice sets, that together satisfies each [i.e., does not violate any] of the characterizing constraints.

We also describe a system for reasoning about constraint satisfaction problems. This reasoning system consists of two components: a *problem solver* that contains domain knowledge, and a *constraint-directed reasoner* that keeps track of the status of constraints and focuses the problem solver's search. We show that by exploiting structural features of the problem and adopting a certain delineation of responsibilities between the constraint-directed reasoner and the problem solver, considerable simplicity in the architecture is achieved. In addition, the architecture is *incremental*; that is, a new solution is derived by modifying an existing solution if the problem description is changed. We provide a precise characterization of the overall reasoning process by describing the algorithms corresponding to the problem solver and the constraint-directed reasoner. We also contrast our representation with related work in operations research and artificial intelligence.

2. The Constraint Satisfaction Problem

Many types of design problems can be viewed as the task of making choices from among competing sets of alternatives. For example, the design (specification) of a computer system might require the selection of a processor, memory unit, operating system, etc. from among the various alternatives available for each. In turn, each choice may entail certain tradeoffs; for example, with respect to cost, performance, and compatibility with other components to be selected.

Often the designer is faced with a set of constraints that must be satisfied by the set of selected choices. Again, using the computer system design example, each set of choices has an associated set of attributes that characterize and distinguish the alternatives in the set. For example, each of the processors that can be selected has an associated **speed** and **cost**. Assuming **cost** is an attribute associated with each of the types of software and hardware components to be selected, then the designer may be faced with a budgetary constraint. That is, the total cost of the various components selected cannot exceed a specified amount.

2.1. Definitions

A *constraint satisfaction problem (CSP)* is characterized by an ordered set $\mathbf{X} = \{X_1, X_2, X_3, ..., X_n\}$ of *choice sets*, and a set $\mathbf{C} = \{C_1, C_2, C_3, ..., C_m\}$ of constraints. Each choice set $X_i = \{x_{i,1}, x_{i,2}, ..., x_{i,n_i}\}$ represents a set of alternatives. Corresponding to each X_i is a set of choice set *attributes* $\mathbf{A}_i = \{A_{i,1}, A_{i,2}, ..., A_{i,m_i}\}$ used to characterize each of the alternatives in that choice set. For example, if X_i is the choice set consisting of a set of computer processors, as discussed above, then **speed** and **cost** are two of the attribute values associated with this choice set, meaning each processor in the set has an associated value for this attribute.

An *assignment* for the set of choice sets $\mathbf{X}$ is a sequence of alternatives $X = <x_{1,j_1}, x_{2,j_2}, ..., x_{n,j_n}>$ where $x_{i,j_i} \in X_i$.

A constraint $C_j \in \mathbf{C}$ can be viewed as a Boolean mapping defined over the set of assignments for $\mathbf{X}$. That is, $C_j: \times_{i=1}^{n} X_i \rightarrow \{T,F\}$. An assignment X for $\mathbf{X}$ is said to *satisfy* the constraint C_i if $C_i(X) = T$; otherwise X is said to *violate* the constraint. An assignment for $\mathbf{X}$ is called a *satisficing assignment* for the CSP characterized by the set of choice sets $\mathbf{X}$ and the constraint set $\mathbf{C}$ if $\forall\ C_i \in \mathbf{C},\ C_i(X) = T$.

We specify constraints in the form

$$t_1, t_2, t_3, ..., t_{n-1} \rightarrow t_n$$

which we call a *dependency constraint*. Each constraint term t_i is a Boolean-valued expression over a set of constants and variables, where each variable is specified in the form $X_i.A_{i,j}$, and denotes the value associated with attribute $A_{i,j}$ of the alternative selected in choice set X_i. Thus, a constraint term states a relationship between various of the choice set attributes and constants, and denotes (assuming each of the variables over which it is defined has a value) either the value TRUE or the value FALSE.

A constraint, specified in the form shown above is interpreted as the material implication

$$t_1 \wedge t_2 \wedge t_3 \wedge \ldots \wedge t_{n-1} \rightarrow t_n$$

We thus call each term that occurs to the left of the arrow in a constraint an *antecedent* term, and the term to the right the *consequent* term. A constraint that has no antecedent terms is called a *premise* constraint. Under this semantics, a constraint is satisfied if each of its terms denotes a value, and either its consequent term denotes TRUE or at least one of its antecedent terms denotes FALSE. Therefore, if each of the relationships specified by the conjuncts in the antecedents of the constraints holds (i.e., denotes TRUE), then the relationship specified by the consequent t_n must also hold. The consequent of a premise constraint must always hold.

2.2. Efficiency and Representation Issues in CSPs

Constraint satisfaction problems are NP-complete and can be solved by backtrack search. Several relaxation methods have been developed for preprocessing a problem so as to reduce the search effort. Waltz (1972) has developed a *"filtering"* algorithm for scene labeling, now commonly known as an *arc-consistency* or *2-consistency* algorithm since it eliminates values from variable domains considered two at a time. Montanari (1974) extended arc-consistency to deal with three variables at a time; the resuling algorithm is known as a *path-consistency* or *3-consistency* algorithm. For definitions of these algorithms, the reader is refered to Mackworth (1977). The basic idea underlying such algorithms is that local constraints become propagated globally through the iterative application of relaxation rules. Freuder (1978) generalized the concept of relaxation by providing an algorithm for achieving *k-consistency*, that is, pruning variable domains by considering *k* variables at a time. When *k* equals the total number of problem variables, the algorithm, in effect, generates all solutions. More recently, Montanari and Rossi (1990) have defined efficient methods that apply every relaxation rule only once.

A CSP, as we define it, is amenable to the application of a *k*-consistency relaxation algorithm. Applying the algorithm would result in the compilation of a list of untenable combinations of up to *k* values. (Freuder actually eliminates untenable values, but they can be useful for explanation, a point we return to later.) There is not enough evidence about the tradeoffs involved in applying the various relaxation algorithms, although preliminary results suggest that node and arc-consistency are almost always useful whereas achieving higher levels of consistency is not generally worthwhile (Dechter, 1989).

While the need to solve constraint satisfaction problems as efficiently as possible is important, it is equally important to have a rich representation for expressing these problems, both from a user and a computational standpoint. For a user, it is important that the problem be expressible as naturally as possible and that a representation be able to provide explanation and *"what-if"* capabilities. From a computational standpoint, the representation should allow a system to incrementally alter an existing solution when the problem description is modified, and to reduce search by being able to exploit control information that might exist in the syntax of the constraints, and by applying application-specific knowledge for dependency-directed backtracking. In the next two sections, we describe a representation that makes the above functionality possible. In simple terms, this is achieved by providing additional semantics to the nodes and links in the constraint network, and by distributing the overall responsibility for solving the problem into two distinct components, a problem solver and a constraint-driven reasoner. The resulting architecture turns out to have some interesting features in common with truth maintenance systems.

3. The Problem Solver

The problem solver is assigned the task of deriving a satisficing assignment for a CSP. Given a CSP characterized by {**X**, **C**}, where **X** is a set of choice sets, and **C** is a set of dependency constraints, it has the responsibility of selecting an appropriate alternative $x_{i,j}$ from each of the choice sets X_i in **X**. Together, the set of selected alternatives must satisfy each of the constraints in **C**.

The problem solver is restricted to making one selection from one choice set at a time. At each instance, the problem solver holds a set of *beliefs,* these beliefs corresponding to the set of alternatives that it has currently selected from various of the choice sets. In turn, a set of currently held beliefs, if retained, may limit the set of alternatives that can be selected by the problem solver from those choice sets for which a selection has yet to be made.

The limitations faced by a problem solver arise as a result of the problem's set of characterizing constraints. Each constraint term specifies a relationship between various of the alternatives. When a constraint term occurs on the right hand side of a constraint it defines a limitation that may have to hold at various times during the problem solving task. Premise constraints, having no left hand side, specify limitations that are in effect throughout problem solving, regardless of the problem solvers current state of beliefs.

The problem solver extends its set of beliefs through the action of making selections. As the set of beliefs expands, the problem solver may become more limited in the future actions that it may take. As the number of limitations grows it may reach a point where the problem solver cannot take any action that will not result in the violation of at least one constraint.

In order to remedy a conflict, the problem solver must change some of its currently held beliefs, supplanting them with other beliefs. This remedy is effected by retracting some currently selected alternatives, and substituting other alternatives from the same choice sets. Thus, the set of beliefs held by the problem solver can grow non-monotonically.

Corresponding to each choice set X_i the problem solver maintains a *selection* variable $\mathbf{X}_\mathbf{I}$ that is used to designate the alternative that it has selected from that choice set. This compound variable consists of one component, designated $\mathbf{X}_\mathbf{I}.A_{i,j}$, for each attribute $A_{i,j}$ over which the associated choice set is defined. At the beginning of the problem solving task each component of each selection variable is initialized to the value **UNKNOWN** indicating that no alternative has been selected from any of the choice sets. For a selection variable $\mathbf{X}_\mathbf{I}$, we represent this initial state as $\mathbf{X}_\mathbf{I}$ = **UNKNOWN**. When the problem solver selects an alternative from the associated choice set it sets each of the components of the selection variable to the corresponding attribute value of that alternative.

Since the problem solver can only select one alternative form one choice set at any instance, this task must be ordered. Although the order in which alternatives from the choice sets are searched must not affect whether or not a satisficing assignment is eventually found -- the search procedure must be exhaustive -- it is likely to determine which of several satisficing assignment is found. In the system that we have implemented the search can be biased by specifying a preference for the order in which choice sets, and alternatives withing choice sets, will be considered. Typically, the ordering is specified in terms of function defined over the choice set attributes. This is analogous to a utility function in decision theory.

Once the problem solver has selected an alternative from a choice set it must then determine a new set of relationships (i.e., limitations) that, based on this selection, must hold among the alternatives, both those that have already been selected, and those that will be selected. To perform this part of its task, it uses a *constraint-directed reasoner (CDR).*

4. The Constraint-Directed Reasoner

The CDR subcomponent is designed to be separate from, but interact closely with the problem solver. With respect to control, the CDR is subordinate to the problem solver. Specifically, with each new belief communicated to it by the problem solver, the CDR computes incrementally the relationships as expressed by the constraint terms that must hold. Also it must be able to detect contradictions in the current set of beliefs. The problem solver is informed of any contradictions that arise, and has the responsibility of resolving them.

The basic unit manipulated by the CDR in carrying out its task is a *constraint term node*. With one exception the CDR maintains a node for each constraint term, regardless of the number of times that constraint term appears among the constraints. The exception to this scheme occurs when one constraint term, say t_i, is the logical negation of another constraint term t_j, that is, $t_i = \neg t_j$. Here one node is used to represent both terms. A constraint term node, designated

<constraint-term-label, constraint-term-value, justifications, consequents>

consists of four components, each of which we describe below.

A *constraint-term-label* designates the constraint terms to which the containing node corresponds. The constraint-term-label of a node explicitly specifies a single constraint term t_i that appears in the antecedent or consequent of one or more dependency constraints. We call this constraint term the *prime designee* of the node.

In addition to its prime designee, a node designates the logical negation of its prime designee. (This logical negation need not appear in a dependency constraint.) Two benefits derive from the ability of a node to designate two constraint terms. First, the number of nodes needed to designate the various constraint terms may be reduced since each constraint term and its negation does not need a unique designator. Second, as will be shown, it provides a convenient mechanism for detecting certain contradictions that, based on the set of beliefs, may arise among the derived relationships.

The *constraint-term-value* component is used to record whether or not the relationship specified by the prime designee, and similarly its negation, is to hold. This value, one of **TRUE**, **FALSE**, **UNKNOWN**, or **T/F** is stated with respect to the prime designee and is implicit for its negation. If constraint term t_i is the prime designee of the node, then a value of **TRUE** indicates that, based on the current set of beliefs of the problem solver, the relationship expressed by t_i must hold, and, equivalently, that expressed by its negation $\neg t_i$ must not hold. Similarly, a value of **FALSE** indicates that the relationship expressed by t_i must not hold, and that that expressed by $\neg t_i$ holds. The value **UNKNOWN** indicates that it cannot be determined from the current set of beliefs whether or not the relationship specified by the designees of the node must or must not hold. If the problem solver has taken some action (i.e., selected an alternative) that leads to a contradiction in that its current set of beliefs is such that both the relationship expressed by t_i and that expressed by $\neg t_i$ must hold, then the *constraint-term-value* component is assigned the value **T/F**. As will be seen, this allows the reasoning system to function with inconsistencies until the problem solver chooses to resolve them.

The *justification* component provides bases for the relationships expressed by the designees of a node. This component consists of two subcomponents: a set of *t-justifications*, and a set of *f-justifications*. Each *t*-justification states the set of beliefs that together form a basis for the relationship specified by the prime designee of the node holding, and thus for the relationship specified by its negation not holding. Similarly, the *f*-justifications provide a basis for the relationship specified by the prime designee not holding, but the relationship specified by the secondary designee holding. As we will discuss shortly, the justification component is used by the CDR to establish or confirm the relationship specified by one of the designees of the node and to detect contradictions.

The *consequent* component of a node identifies those constraint terms, and thus nodes, that specify relationships whose value, that is whether or not they hold, may be affected by the current value of the node containing this component. The identified nodes correspond to the consequent terms of those constraints where a designee of the current node appears as an antecedent term. Thus, consequent components establish dependencies among the designees of the constraint term nodes. A consequent component also consists of two subcomponents: a set of *t-consequents*, and a set of *f-consequents*. The *t*-consequents identify those nodes having a designee whose value may be dependent on the value of the prime designee of the current node. Similarly, the *f*-consequent identifies those nodes having a designee whose value is potentially dependent on the value of the secondary designee of the current node.

In identifying constraint term nodes the values of the *consequent* subcomponents, in effect, define edges between the containing node and the nodes identified by these values. These edges, along with the constraint term nodes define a *dependency net* that characterizes the set of constraints from which it is derived, and that is used for constraint propagation.

Since constraint term nodes correspond to dependency constraint terms they can, and are, created when the CSP is specified to the system. At this time one constraint term node is created for each term and, if present, its negation, encountered in the set of dependency constraints. The first of the two terms encountered becomes the prime designee of the created node.

The initial value of the *constraint-term-value* component of a newly created node is determined by the placement of the node's designees within the set of characterizing dependency constraints. If neither designee appears as the consequent term of a premise constraint, then the *constraint-term-value* is set to **UNKNOWN**, indicating that initially it is not known whether or not the relationships specified by the designees of the node must hold.

The relationships specified by premise constraint terms must always hold. Thus the nodes for which these constraints are designees must have a *constraint-term-value* that is not initialized to **UNKNOWN**. Rather, if the prime designee of the node occurs in a premise constraint, then the *constraint-term-value* is initialized to **TRUE**. Similarly, it is initialized to **FALSE** if the negation of the prime designee occurs in a premise constraint. The occurrence of both designees of a node in premise constraints indicates an inconsistency in the set of characterizing constraints, knowledge of which the user is informed.

The two *justification* subcomponents have initial values that are also determined by the nature of the designees of the containing node. If initially the problem solver has no basis for belief in the relationship expressed by the designees of a node, that is, the initial *constraint-term-value* is **UNKNOWN**, then equivalently there must not be any justification for these relationships. Accordingly, the *t-justification* and *f-justification* are both initialized to **nil**.

An initial *constraint-term-value* of **TRUE** or **FALSE** in a node corresponds to the prime designee or the secondary designee, respectively, being a premise constraint term. For such nodes a special marker P is used to indicate that the relationship specified by one of the designees of the node holds because it was specified as a premise constraint. If the *constraint-term-value* is **TRUE**, then the *t-justification* is initialized to the set {**P**} and the *f-justification* is initialized to **nil**. Similarly, if the *constraint-term-value* is FALSE, then the *t-justification* and the *f-justification* are initialized to **nil** and {**P**}, respectively.

It should be noted that the constraint network is compiled when the constraints are specified, and does not change with the changing state of the problem. The size of the network is bounded by the number of constraint terms.

5. Implementation

In this section we describe the data structures and algorithms used in implementing the overall problem solving system. These descriptions are not intended to be exhaustive. Rather, they are intended to provide a somewhat simplified, and for reasons of exposition, ideal, view of how the system is constructed and functions. Thus the descriptions range from simple narratives when adequate, to more formal programming language-like descriptions using both structured and object-oriented language conventions.

5.1. Data Structures

The basic data structures manipulated by the problem solver are those that are used to represent choice sets. A data structure of type **choice-set** is a record-like object defined as

```
choice-set =
    object of
        selection: integer;
        alternative: array [1..#-of-alternatives]
            of attribute-indexed records
    end
```

The alternatives in a choice set are represented as elements of the array that is defined as the second component of a choice-set object. Each of these elements is an associative record structure (e.g., dictionary) that contains one value for each attribute over which the choice set is defined. The value for a particular attribute of an alternative is retrieved by specifying the appropriate attribute name. Thus, **cs.alternative[n].A** (or for brevity, **cs[n].A**, where **attribute** is understood) references the value associated with attribute **A** of the nth alternative in choice set **cs**. The first component of a choice-set object, referenced as **cs.selection**, specifies the index of the alternative that has currently been selected from choice set **cs**. A value of zero is used to indicate that no alternative has currently been selected from the specified choice-set object.

All objects of type **choice-set** are maintained by the problem solver. In particular, the problem solver is responsible for setting the value of the **selection** slot of these objects to indicate which of the alternatives in the corresponding choice set it has selected. In order for the CDR to be able to determine the affect of a newly made selection (or a change in a selection) on its belief of which relationships hold, it is given read access to each instance of an object of type **choice-set**. This read access is provided so as to simplify parameter passing in the system.

The basic data structures manipulated by the CDR are objects of type **c-term-node** that, as described in the previous section are defined to correspond to constraint terms. Together, instances of **c-term-node** objects are used to implement a dependency net that models the set of constraints **C** that characterize the target problem. A node of type **c-term-node** is defined as follows:

```
c-term-node =
    object of
        c-term-label: c-term-func;
        c-term-value: extended-Boolean[1];
        t-justif: set of support-sets;
        f-justif: set of support-sets;
        t-conseq: set of c-term-nodes;
        f-conseq: set of c-term-nodes;
    end
```

The **c-term-label** component of a **c-term-node** object is implemented as an extended-Boolean-valued

[1]We define an *extended-Boolean* as consisting of, depending on the context, a specified set of other values in addition to those of **TRUE** and **FALSE**. *In particular we allow the values* **UNKNOWN** *and* **T/F**.

function (c-term-func) that is derived from the constraint term that is the prime designee of the node. When executed this function accesses the appropriate **choice-set** instances (those over which the corresponding constraint term is defined) and returns a value that results from computing the relationship expressed by the constraint term. If too few of the choice sets over which this relationship is defined have had alternatives selected, preventing a value of **TRUE** or **FALSE** from being returned, then **UNKNOWN** is returned as the value of c-term-label.

The value of a **c-term-value** component can be **TRUE**, **FALSE**, **UNKNOWN**, or T/F. If neither designee of the c-term-node is the consequent of a premise constraint, then the c-term-value is initialized to the value **UNKNOWN.** The c-term-value component is initialized to **TRUE** if the prime designee of the node is the consequent of a premise constraint, and to **FALSE** if the secondary designee of the node is the consequent of premise constraint. Since the the c-term-value component reflects the current belief in the relationship specified by a designee of the c-term-node, its value can be expected to be changed by the CDR throughout the course of the problem solving task.

The **t-justif** and **f-justif** components of a c-term-node object corresponds, respectively, to the t-justification and f-justification subcomponents described in the previous section. Each of these components is implemented as a set of objects of type **support-set**.

Each element of a **support-set** object is a structured object of type **support-element** consisting of two components. The first component is an instance of a **c-term-node** object, and the second component is one of the Boolean values **TRUE** or **FALSE.**

The **t-justif** and **f-justif** components of a node contain one support-set object for each problem constraint in which the prime designee (implemented as the associated c-term-label) and its negation, respectively, appear as the consequent term. Each object of type **support-set** contains one **support-element** object for each antecedent term in the corresponding problem constraint

A **support-set** object is used by the CDR to determine if belief in the relationship specified by a designee of the containing c-term-node is derivable from (i.e., supported by) belief in each of the relationships specified by the antecedent terms of the corresponding problem constraint. The first component of each **support-element** of a **support-set** identifies the c-term-node associated with one of these antecedent terms. The second component, the Boolean value, specifies which of the two designees of the identified c-term-node object corresponds to the antecedent term. The value **TRUE** indicates the prime designee, **FALSE** its negation.

The **t-conseq** and **f-conseq** of a c-term-node object are implemented as sets. Each element of each of these sets identifies a c-term-node object that has a designee that is the consequent term of a problem constraint for which the prime designee, in the case of **t-conseq,** and its negation, in the case of **t-conseq,** of the current node appears as an antecedent term.

5.2. Algorithms

Each of the algorithms, presented here in the form of a function or a procedure, comprise one or the other of the CDR or the problem solver. In the overall control scheme the problem solver, using some heuristic, selects some alternative from a choice set. Based on this selection and the set of problem constraints the CDR makes a series of deductions that determine what relationships must hold among various of the choice set alternatives. When no more deductions are possible, the constraint set is said to be *relaxed*. If no constraint violation (i.e., inconsistencies in the set of relationships that must hold) are detected by the CDR, control passes back to the problem solver and the cycle continues. If any violations are detected, the CDR performs dependency analysis in order to determine those sets of selections, such that each set identifies those selections that together leads to at least one of the detected violations.

The problem solver is implemented by the procedure **PROBLEM-SOLVER** shown below.

```
Procedure PROBLEM-SOLVER ()
PS-1.    cs = select-unassigned-cs
PS-2.    if cs = undefined
PS-3.      then return (true)
PS-4.    cs.select = 1
PS-5.    while CDR-NOGOOD-VIOLATION(cs) = true
                 and cs.select ≤ number-of(cs.alternatives)
PS-6.      do cs.select = cs.select + 1
PS-7.    if cs.select > number-of(cs.alternatives)
           then
PS-8.        cs.select = 0
PS-9.        return (fail)
PS-10    conflict-set-list = ∅
PS-11.   CDR-PROPAGATE(cs, conflict-set-list)
PS-12.   if not-empty(conflict-set-list)
           then
PS-13.       retract-list = choose(conflict-set-list)
PS-14.       for each choice set C in retract-list
               do CDR-RETRACT(x)
PS-15.    while fail(PROBLEM-SOLVER)
                 and cs.select ≤ number-of(cs.alternatives)
PS-16.     do cs.select = cs.select + 1
PS-17.   if cs.select > number-of(cs.alternatives)
           then
PS-18.       cs.select = 0
PS-19.       return (fail)
PS-20.     else return (true)
```

The function **select-unassigned-cs** invoked in Step PS-1 of the problem solving algorithm encodes the heuristic for determining from which choice set an alternative will be next selected. If an alternative has currently been selected from each choice set, this function retrurns the value *undefined*.

After a choice set has been selected, the problem solver attempts to select an alternative from it by using the procedure **CDR-NOGOOD-VIOLATION** to successively test alternatives to find one that does not form in conjunction with other currently selected alternatives a combination that from past experience the CDR knows will lead to an inconsistency. (Each untenable combination of selections, called a *nogood*, when first detected by the CDR is added to a list. This list of nogoods is accessed by **CDR-NOGOOD-VIOLATION** in the performance of its task.)

Once the problem solver has selected a suitable alternative (i.e., one that does not lead to a combination of selected alternatives that encompasses a nogood) it informs the CDR of this selection through the invocation of the CDR module **CDR-PROPAGATE.** This module, which is described below, controls the constraint propagation function of the CDR. If no contradictions arise from the propagation, then the problem solver continues its task, through a recursive call to itself, by selecting another choice set from which to select an alternative. If a contradiction is detected during propagation then **CDR-PROPAGATE** provides, through its second argument, information about the combinations of selected alternatives that led to the contradictions, providing the problem solver with the information that it needs to so that it can take appropriate action to alleviate the problem before continuing the task of selecting alternatives.

The CDR is organized as a set of modules, each performing a specific aspect of the overall CDR function, and serving as an entry point to the CDR from the problem solver. One of these modules, **CDR-PROPAGATE,** is invoked by the problem solver to effect changes in the set of relationships, expressed by constraint terms, that, based on the set of alternatives that have currently been selected, are believed to hold.

Procedure CDR-PROPAGATE `(cs, conflict-sets)`

```
CDR-1.  for each c-term-node c in entry-nodes(cs)
         do CDR-LT-PROPAGATE (c, conflict-sets)
CDR-2.  return
```

The procedure **CDR-PROPAGATE** serves as an overall control module for the propagation function of the CDR. This module invokes the procedure **CDR-LT-PROPAGATE** for each c-term-node contained within the set indexed by the choice set denoted by the parameter cs.(i.e., entry-node(cs)). This indexed set identifies each of the constraint term nodes having a designee, and thus a label, that is defined in terms of the indexing choice set cs. Each of these nodes serves as an entry point into the dependency net, and allows the CDR to only have to consider those constraint term nodes that have a c-term-label value that can be affected by the alternative that was selected by the problem solver.

The procedure **CDR-LT-PROPAGATE** is used to determine if the value of the c-term-label of the constraint term node identified by its first parameter is affected by the the selected alternative. Such a change may, depending on the set of problem constraints and the current state of the set of c-term-nodes, necessitate the propagation of constraint term values.

Procedure CDR-LT-PROPAGATE `(c, conflict-sets)`

```
L1. if c.label = unknown or c.label = c.value or c.label = t/f
L2.   then return
      else
L3.    if c.value <> unknown
        then
L4.       let c.value = t/f
L5.       assumpt-sets = CONFLICT-ASSUMPTS (c)
L6.       nogoods = nogoods ∪ assumpt-sets
L7.       conflict-sets = conflict-sets ∪ assumpts-sets
```

```
           else
L8.           c.value = c.label
L9.        if c.label = true
             then
L10.          for each const-term-node x in t-conseq
L11.            do CDR-JT-PROPAGATE (x, conflict-sets)
             else
L12.          for each const-term-node x in f-conseq
L13.            do CDR-JT-PROPAGATE (x, conflict-sets)
```

Step L1 of **CDR-LT-PROPAGATE** is used to determine whether or not the propagation process should continue for the current constraint term node. If the c-term-label component of the node evaluates to unknown or to a value that is the same as that of the c-term-value component, or the c-term-value component has the value *t/f*, then propagation does not proceed; in the first case because a value of unknown indicates a lack of belief in whether the relationships corresonding to each of the designees of the node should hold, in the second and third cases because the current value of the c-term-value component indicates that propagation, if necessary, was performed during an earlier visit to the node when this value was originally determined.

Assuming that the c-term-label component evaluates to a value other than *unknown*, and that this value is different from that of the c-term-value component, Step L3 of the procedure checks to see if a conflict has occured. If a conflict has occurred, that is, the value of the c-term-value component is also other than *unknown*, and by Step L1 is different from that of the c-term-label component, then the procedure sets c-term-value to *t/f*, and invokes the function **CONFLICT-ASSUMPTS** to determine the underlying set of the problems solvers beliefs that led to the conflict.

Finally, **CDR-LT-PROPAGATE** attempts to propagate the newly derived c-term-label value forward by invoking **CDR-JT-PROPAGATE.** This procedure is invoked using each of the c-term-nodes pointed to by the current nodes t-conseq component if the c-term-label value is *true*, or the f-conseq component if the c-term-label value is *false*.

The function **CONFLICT-ASSUMPTS**, when presented with a c-term-node for which a conflict has been detected, returns a set of **conflicting-assumption sets.** Each conflicting assumption set is a subset of those alternatives that have currently been selected from the various choice sets by the problem solver, and that together, in conjunction with the set of problem constraints, lead to the detected conflict. Each such conflict set is saved as a nogood for later use by the CDR module **CDR-NOGOOD-VIOLATION** in its task of helping the problem solver avoid remaking futile combinations of selections.

As an example of the formation of these nogoods consider the set of dependency constraints

$$t_A, t_B \rightarrow t_C$$
$$t_E, t_C \rightarrow t_D$$
$$t_B, t_E \rightarrow \neg t_D$$

where each constraint term t_i is defined in terms of a single choice set *i*. Assume that alternatives have been selected from each of the choice sets corresponding to the constraint terms shown above, and that these selections lead to a conflict that is detected in the c-term-node that has t_D and $\neg t_D$ as its designees. In addition, assume that the alternative that has been selected from choice set D is such that the relationship specified by t_D holds. We could resolve the conflict by retracting belief in (i.e., some of the selections that support) either the relationship specified by t_D or that specified by $\neg t_D$. The former requires the retraction of the selected alternatives in choices sets D and E or choice set D and C and either of A and B. The latter requires the retraction of the selected alternatives in either of choice sets B or E. Graphically, these combinations can be represented by the following AND/OR graph:

```
        or
       /  \
      /    \
     /      \
  and       and
  / \         \
 /   \         \
D     \         \
      or         or
      /\         /\
     /  \       /  \
    E    \     E    B
        and
         /\
        /  \
       C    \
            or
            /\
           /  \
          B    A
```

The leftmost subtree of the root (topmost or) node specifies those combinations of selected alternatives upon which belief in the relationship specified by t_D is based. Similarly, the right subtree shows the combination of alternatives that provide support for belief in the relationship specified by $\neg t_D$.

The procedures **CDR-LT-PROPAGATE** and **CDR-JT-PROPAGATE** invoke **CONFLICT-ASSUMPTS** to construct a set of conflict sets for each constraint term node for which a selected alternative leads to a conflict. The union of these sets of conflict sets are returned to the problem solver which has the task of deciding which of the selections should be retracted in order to eliminate the conflicts.

The procedure **CDR-JT-PROPAGATE** shown below is used by the CDR to determine if belief in the relationship specified by one of the designees of the c-term-node denoted by the first argument of the procedures has become newly justified. This justification of a designee is determined using the t-justif and f-justif components of the c-term-node, with the associated designee corresponding to a consequent constraint term. When a relationship is newly justified the c.value component of the node is set accordingly, and, depending on that value, propagation continues through a recursive call to **CDR-JT-PROPAGATE** using each of the nodes in either the t-conseq or the f-conseq component.

Procedure CDR-JT-PROPAGATE `(c, conflict-sets)`

```
J1. if not SATISFIED (t-justif) and not SATISFIED (f-justif)
```

```
J2.    then return
J3. if SATISFIED (t-justif)
     then
J4.     if c.value <> true
         then
J5.       if c.value = unknown
           then
J6.           let c.value = true
J7.           for each cnstr-term-node x in t-conseq
J8.             do CDR-JT-PROPAGATE (x, conflict-sets)
           else
J9.           let assumpt-sets = CONFLICT-ASSUMPTS (c)
J10.          let nogoods = nogoods ∪ assumpt-sets
J11.          let conflict-sets = conflict-sets ∪ assumpt-sets
J12.           if c.value = false
                then
J13.             let c.value = t/f
J14.             for each const-term-node x in t-conseq
J15.                do CDR-JT-PROPAGATE (x, conflict-sets)
J16. if SATISFIED (f-justif)
      then
J17.    if c.value <> false
         then
J18.      if c.value = unknown
           then
J19.          let c.value = false
J20.          for each const-term-node x in f-conseq
J21.            do CDR-JT-PROPAGATE (x, conflict-sets)
          else
J22.          let assumpt-sets = CONFLICT-ASSUMPTS (c)
J23.          let nogoods = nogoods ∪ assumpt-sets
J24.          let conflict-sets = conflict-sets ∪ assumpt-sets
J25.          if c.value = true
               then
J26.             let c.value = t/f
J27.             for each const-term-node x in f-conseq
J28.               do CDR-JT-PROPAGATE (x, conflict-sets)
```

The function **SATISFIED** used by **CDR-JT-PROPAGATE** to determine if its argument, a t-justif or f-justif component, has a support-set that is satisfied in the sense that each of its support-elements identifies a *c-term-node* that has a *c-value* that is equal to the value specified by the second component of the support-element. If such a support-set is found, then **SATISFIED** returns the value *true*; otherwise it returns the value *false.*

In addition to **CDR-PROPAGATE**, the CDR provides the problem solver with two other entry modules: **CDR-RETRACT** and **CDR-NOGOOD-VIOLATION**. The first of these modules is used by the problem-solver to undo the affects on the dependency net of a selection that it has retracted. The function and structure of this module is similar to that of **CDR-PROPAGATE**, and will not be further elaborated on here.

The module **CDR-NOGOOD-VIOLATION** maintains a database of nogoods, and is used by the problem solver to determine if a prospective alternative that it would like to select from a choice set will lead, in conjunction with other of the alternatives that it has selected from other choice sets, to a conflict. Unlike the other two CDR modules, **CDR-NOGOOD-VIOLATION** does not access any of the c-term-nodes that make up the dependency net.

5.3. Incrementality

Design and planning problems, including those that can be modelled as CSPs are often subjected to *incremental* changes to the problem specification. For example, it might be desirable to perform impact analysis on a problem solution or to otherwise modify the problem specification based on the current solution.

With respect to CSPs, an *incremental* change is a single modification to the set of constraints or choice sets that characterize a CSP: a choice set or constraint might be added to or deleted from the problem specification, or a choice set might be modified by adding or deleting an alternative within it. (The modification of an existing constraint or choice set alternative can be effected through a deletion and insertion of constraints or choice set alternatives, respectively, and thus we will not explicitly discuss them.)

The architectural framework that we have defined accomodates incremental changes to the specification of a CSP. The impact on an existing problem solution of an incremental change will depend on the nature of the modification made. Some modifications to the problem specification will have no affect on an existing solution, others will require that different alternatives be selected for some or all of the choice sets.

Deleting an existing constraint from the problem specification, adding a new choice set, or adding a new alternative to a choice set has the least impact on an existing solution, and is thus the easiest type of modificaton to handle within our framework. The deletion of a constraint from a set of constraints has the affect of relaxing the set of constraints. Any CSP solution that satisfies the unmodified set of constraints will therefore satisfy the modified set of constraints. (In deleting a constraint it is also necessary to modify the dependency net that is manipulated by the **CDR** so that it reflects the remaining set of constraints.)

If a CSP is modified by adding a new choice set, or adding a new alternative to an existing choice set, then what had been the existing solutions must still satisfy the set of constraints since these have not been modified. However, it becomes necessary to select an alternative for the new choice set. This process is effected by invoking the **PROBLEM-SOLVER** which will then attempt to extend the existing solution by selecting one of the alternatives from the new choice set. Since (by definition) the existing constraints cannot have been defined over the new choice set any of its alternatives can be selected without violating any of these constraints.

A CSP can also be modified by removing one of its characterizing choice sets. If no constraints are defined over this choice set, then no additional solving problem is required since the existing set of selected alternatives from the remaining set of choice sets will still satisfy the set of problem constraints. If there are constraints that are defined over the choice set that is to be removed from the problem specification, then the removal of the choice set will cause these constraints to become invalid. Within the context of our framework, we require that these constraints first be incrementally removed from the problem specification, at which point the target choice set can then be removed.

The deletion of an alternative from an existing choice set will only impact on a problem solution if that alternative is part of the solution. In this situation the **PROBLEM-SOLVER** is invoked to attempt to reextend what has become a partial solution to a full solution by selecting a different alternative from the choice set.

When modifying a CSP by adding a new constraint to its specification, it is necessary to modify the **CDR** dependency net to reflect the new set of constraint. Unlike the situation that existed when the dependency net was originally defined, it is now possible to specify the *constraint-term-value* component of each of the constraint term nodes that are inserted into the dependency net since, assuming that the constraint is valid, alternatives have been selected for each of the choice sets over which the constraint has been defined. Similarly, it is possible to determine if the existing solution satisfies the new constraint. If it satisfies the constraint, then it is also a solution for the modified problem, and no further problem solving is required.

The situation that arises when an existing solution violates a newly added problem constraint is similar to that which arises during problem solving when the **PROBLEM-SOLVER** selects an alternative from a choice set that violates a constraint, and is resolved in a similar way. The procedure **CONFLICT-ASSUMPTS** is invoked with the consequent constraint term of the newly added constraint as its argument. The set of *assumptions* returned by this procedure is then used by the **PROBLEM-SOLVER** to resolve the conflict.

5.4. Explanation

There are three types of knowledge that form the basis for variable assignments: variable (choice set) ordering, value (alternative) ordering, and constraints. The first expresses the relative "importance" of each choice set, that is, the relative importance of making the more preferred selections in them. Value ordering is expressed via preference (or utility) functions defined over the alternatives within choice sets. Several heuristic approaches to variable and value ordering for reducing backtracking have been discussed in the literature [Dechter and Pearl, 1988]. Finally, constraints force the problem solver to explore only the feasible solutions.

Because of the preference functions, given a choice set X_i consisting of an ordered set of alternatives $\{X_{i,1}, X_{i,2}, \ldots X_{i,n}\}$, the question *"why $X_{i,k}$?"* implicitly states *"why not any of the alternatives preceding $X_{i,k}$?"* Since the system must have actually attempted all of these, it must be the case that they led to constraint violations when considered in conjunction with some other selections. If the variable ordering used for retraction is the reverse of that used for search (i.e., chronological backtracking), then it must be the case that selections preceding $X_{i,k}$ were not possible in conjunction with attempted selections in choice sets preceding X_i. Since backtracking is usually not chronological, however, all that can be said is the alternatives preceding $X_{i,k}$ were not feasible with certain other selections; all such attempts are recorded as nogoods.

Finally, the third basis for variable assignments is contained in the justification structure of the constraint term nodes, and is similar, conceptually, to the notion of data dependencies in truth maintenance systems. Specifically, if a node corresponding to a consequent term (say, *"hardware speed should be greater than 5 MIPS"*) has its *t-justif* or *f-justif* satisfied, it means that the current selections in choice sets associated with that term (in this case *hardware*) satisfy the constraint expressed by that term; further the problem solver was constrained into making this selection because the antecedent terms that make up the justification for the current node also hold. Each of these antecedent terms, in turn, has either a similar support, or it is justified by a selection made according to a utility function. Thus, ultimately it is the utility functions that form the bases on which all selections are justified.

6. Relationship to Other Work

Our work is related to a number of constraint satisfaction approaches, notably, relaxation methods, truth maintenance, and integer programming.

6.1. Relationship to Relaxation Methods

Relaxation methods focus on eliminating bad combinations of assignments prior to search by transforming a given constraint network into a "more explicit" one (Montanari and Rossi, 1990). This reduces backtracking in subsequent problem solving. It is interesting to note that the application of a *k*-consistency algorithm results in nogoods of size upto *k*. Smaller nogoods are more powerful in pruning search.

In our model, nogoods are examined by the function TMS-NOGOOD-VIOLATION to ensure that the problem solver aviods these untenable combinations of assignments. A second use of nogoods, as described in the previous section, is that they are useful for explanation.

6.2. Relationship to Truth Maintenance Systems

The relationship of our model with truth maintenance systems can be examined in terms of the structure and semantics of the nodes in the dependency network, and the labelings of the nodes.

Table 1 indicates the relationship between the four valued logic used to label nodes in our constraint network and the INs and OUTs of a Doyle-style TMS. A truth value of *true* for a proposition corresponds to it being IN and its negation being OUT. Similarly, a *false* corresponds to the proposition being OUT and its negation being IN. A value of *unknown* indicates that the proposition and its negation are both unknown. Finally, a value of *t/f*, indicating a contradiction, indicates that the proposition and its negation are both IN.

Truth value of term p \ Term	p	~p
TRUE	IN	OUT
FALSE	OUT	IN
UNKNOWN	OUT	OUT
T/F	IN	IN

Table 1

A non-monotonic justification such as *"unless* $x \rightarrow y$", which states that unless x is true y is true, is expressable in our language as a disjunction of false and unknown, that is, *"(false x) OR (unknown x)* $\rightarrow$ *y"*. In effect, a non-monotonic justification is converted into a Boolean expression representing a term-node which is handled in the standard way by our constraint-driven reasoner.

In terms of the semantics of the nodes of the dependency network, the t-justif and f-justif part of constraint term nodes are similar to support-list justifications of TMSs. A fundamental difference, however, is that our structure models a constraint expression (a term) and not a problem solver datum. In contrast, dependency nodes in truth maintenance systems represent assertions (each being a problem solver AND a TMS datum with different meanings in the two) whose justification structure is dynamic. In fact, in order to maintain *consistency* and *well-foundedness* -- two fundamental properties that the data must satisfy -- a TMS essentially manipulates the justification structures which in turn determine node labelings. In contrast, our constraint reasoner basically performs label propagation with static justification structures, abdicating all decision-making responsibility to the problem solver. This leads to considerable simplicity in our status assignment algorithms and a more natural division of responsibility between the problem solver and the constraint reasoner, avoiding the rather ad-hoc constraint satisfaction methods employed by the various TMSs.

It should be noted that the constraint terms can be constraints themselves, expressing relationships among sets of selections across choice sets. For example, a constraint involving a term of the form

hardware.cost < (software.cost + operating-system.cost)

specifies a relationship that holds for certain combinations of hardware, software and operating systems (as the problem solver makes selections, the CDR determines whether such relationships, and hence the constraints they make up, hold). Since the terms can be arbitrarily nested Boolean expressions, higher order constraints are easily expressed. In general, the number of constraints of the form above is small compared to the size of the search space. Thus the dependency network maintained by our reasoner is small, resulting in an efficient constraint-driven module.

6.3. Operations Research Approaches

Constraint satisfaction problems have been dealt with extensively in the Operations Research literature where an additional requirement of optimality is expressed via an objective function. If the constraints and objective function are linear, and the variables are continuous valued, the problem is easily solved using linear programming (LP) algorithms such as the Simplex algorithm (Dantzig, 1963) or Karmarkar's new algorithm (Karmarkar, 1984). Solving a discrete valued problem is more difficult. It involves an iterative process where each iteration begins by first solving its LP relaxation (that is, ignoring integrality). The set of feasible solutions of the LP relaxation form a polytope which is generally a superset of the polytope representing integer solutions. Therefore additional constraints (sometimes called "cuts") are introduced into the formulation to move toward the integer solutions. This is accomplished in the second step by using either the branch and bound or the "cutting planes" technique (Gomory, 1958; Chvatal, 1973). Grotschel and Padberg (1982) have reported remarkable success in applying specialized branch and bound and cutting planes algorithms in solving the traveling salesman problem. In addition, Crowder et al. (1983) have described several constraint pre-processing and cutting plane generation strategies for general 0-1 problems that result in a dramatic reduction in the work done by the branch and bound step.

The constraints involved in these discrete problems, linear constraints, are special cases of those in the constraint satisfaction problem described in this paper. Thus, certain special cases of our problem can be solved efficiently using these methods. In the remainder of this section we describe these special cases and how they can be transformed for solution using discrete optimization methods. We also describe how our TMS can be coupled with an optimization module to provide a useful decision support functionality.

Since choice sets contain discrete sets of alternatives each of which may or may not be selected, each alternative can be characterized in terms of a *0-1* variable. Constraints can then be expressed in terms of algebraic relationships among Boolean variables. Each such constraint can in turn be expressed as a clause. For example, the constraint $\neg s_1, s_2 \rightarrow s_3$ is equivalent to *"s_1 or not-s_2 or s_3"* where each s_i is a propositional variable. In this way, the problem can be expressed conveniently in conjunctive normal form.

Each clause can be expressed as an inequality. For example the above clause can be expressed as

$$s_1+(1-s_2)+s_3 \geq 1$$

In general, as has been noted independently by Hinton (1979) and Hooker (1988), a clause can be expressed in the form:

$$c_1 s_1 + \ldots + c_n s_n \geq 1-n(c)$$

where c is a row vector and s is a column vector, and $n(c)$ is the number of negative elements in the vector c. Each c_i is 1,0, or -1, indicating whether s_i appears, does not appear, or $\neg s_i$ appears in the clause, respectively. The above notation is due to Hooker (1988).

If the constraint set consists entirely of premise constraints, the problem can be formulated as a general *0-1* integer programming problem. If all terms in the constraints are linear, we have a linear *0-1* formulation. For example, the premise *"software cost is less than hardware cost"*, where *software* and *hardware* are choice sets and cost is an attribute of both sets, expresses a linear constraint. In contrast, a premise constraint such as *"the ratio of hardware to software costs should be less than half the air-conditioning equipment cost"* is a non-linear (quadratic) constraint.

Non-linear cases can be solved by transforming the problem into linear form. It has been shown (Watters, 1967) that any polynomial *0-1* program can be transformed into a linear *0-1* program by replacing every product of *0-1* variables by a new *0-1* variable and introducing additional constraints (see Hansen,1979). It has been recognized, however, that the number of new variables and new constraints so introduced may be very large even for small non-linear *0-1* problems (Hansen, 1979), making them difficult to solve.

If the set *C* includes non-premise dependency constraints involving terms of the form described above, the problem can still be reduced to a *0-1* form, although the number of *0-1* constraints required to express a dependency constraint can be large, depending on the number of terms in it and the sizes of choice sets referenced by the terms. Essentially, each term of a dependency constraint requires enumerating the set consisting of combinations of selections (from the choice sets referenced in the term) that satisfy the term expression. Specifically, a constraint term involving n choice sets each with an average of k selections can result in a set of size on the order of k^n. Expressing the constraint as a whole requires generating the cartesian product of the sets corresponding to the constraint terms. Expressing dependency constraints using *0-1* variables could therefore result in a large number of constraints. As with the case above involving only premise constraints, the formulation becomes even more difficult if the constraints turn out to be non-linear, as does the effort required to solve the problem.

OR techniques have two additional drawbacks. 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 is difficult). 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. These issues have been discussed at length by Dhar and Ranganathan (1989) in the context of a course scheduling type of constraint satisfaction problem.

The limitations of OR techniques can be overcome to a some extent by coupling an optimizer to a constraint reasoning module such as our CDR. The architecture that we have implemented can be coupled with an optimization package to achieve a functionality that allows for the repercussions of changes to be assessed incrementally. Specifically, if an initial optimal solution is found, the choices that make up this solution can be communicated to the problem solver and the CDR. Conducting a what-if analysis is then straightforward since the CDR can compute the impacts of changing decisions. A change can either "go through" (not require making changes in other parts of the solution), or result in violated constraints, identified by the CDR. In the latter case the CDR computes alternative fixes (represented by the AND/OR graph in the previous section) to be evaluated by the problem solver or/and the decision maker.

7. Concluding Remarks

We have provided precise descriptions of the class of problems modeled by our architecture and the algorithms corresponding to the problem solver and the constraint-directed reasoner. We also described how (and why) the tasks in problem solving are distributed between them.

Our objectives have been to design an architecture that has the expressive power to represent a general class of constraint satisfaction problems, to solve such problems efficiently, and for the solution to be incrementally modifiable. In addition, the modeling primitives are powerful and simple enough to enable a user to describe a problem as naturally as possible. Our architecture has been motivated in large part out of frustration in trying to achieve these objectives simultaneously with existing tools.

A common drawback of most AI tools that we have witnessed is that the knowledge engineer or user has difficulty in fitting the problem into the primitives provided by the tool. For example, we have found that in systems that use TMSs, it is often unclear how the problem solver should be designed so that the interactions (and responsibilities) between it and the TMS are demarcated correctly. In practice, we have found that the importance and difficulty of such decisions is often underestimated, and that it is often necessary for knowledge engineers (or users) to familiarize themselves with the inner workings of the tool to make good design decisions. In contrast, we have observed that users of our system are able to quickly specify declaratively the various knowledge components of their constraint satisfaction problem once the choice sets and their attributes have been specified (although these tend to get modified as the constraints are expressed). The problem solver and the CDR are completely transparent to the user, an important consideration in designing complex reasoning systems for real-world applications.

REFERENCES

Alexander, C., *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, Mass 1964.

Chvatal, V., Edmonds Polytopes and a Hierarchy of Combinatorial Problems, *Discrete Mathematics*, 4, 1973.

Croker, A., Dhar, V., and McAllester, D., Dependency Directed Backtracking for Generalized Satisficing Assignment Problems, to appear in *Management Science*. Available as Technical Report 190, Department of Information Systems, NYU, 1988.

Crowder, H., Johnson,E., and Padberg, M., Solving Large-Scale Zero-One Linear Programming Problems, *Operations Research*, vol 31, no. 5, September-October 1983.

Dantzig, G., *Linear Programming and Extensions*, Princeton University Press, 1963.

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

Dechter, R. Methodolgy for CSPs, Workshop on Constraint Processing, IJCAI, Detroit, MI, August 1989.

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

Dhar, V., and Ranganathan, P., Experiments with an Integer Programming Formulation of an Expert System, MCC Technical Report ACA-AI-022-89, Austin, Texas, February 1989.

Doyle, J., A Truth Maintenance System, *Artificial Intelligence*, June, 1979.

Freuder, E.C., Synthesizing Constraint Expressions, *Communications of the ACM*, 21,11, November, 1978.

Gomory, R.E., Outline of an Algorithm for Integer Solutions to Linear Programs, in R.L.Graves and P.Wolfe, eds., Recent Advances in Mathematical Programming, McGraw-Hill, 1963.

Goodwin, J.W., A Process Theory of Non-Monotonic Inference, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 1985.

Grotschel, M., and Padberg, M., The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley 1982.

Hansen, P., Methods of Nonlinear 0-1 Programming, *Annals of Discrete Mathematics* 5, 1979, pp. 53-70.

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

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

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

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

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

Montanari, U., Networks of Constraints: Fundamental Properties and Application to Picture Processing, *Information Science*, 7, 1974.

Montanari, U., Rossi, F., Constraint Relaxation May be Perfect, *Artificial Intelligence*, forthcoming

Nudel, B., Consistent Labeling Problems and Their Algorithms: Expected-Complexities and Theory-Based Heuristics, *Artificial Intelligence*, **21**, 1983, pp. 135-178.

Petrie, C., Russinoff, D., and Steiner, D., Proteus 2: System Description, MCC Technical Report AI-136-87, May 1987.

Reinfrank, M., Lecture Notes on Reason Maintenance Systems, Technical Report INF2 ARM-5-88, Siemens AG, Munich, West Germany, 1988.

Reitman, W. R., *Cognition and Thought*, Wiley, New York, 1965.

Simon, H., The Structure of Ill-Structured Problems, *Artificial Intelligence*, **4**,3, September 1973.

Watters, L.J., Reduction of Integer Polynomial Programming Problems to Zero-One Linear Programming Problems, *Operations Research*, **15**, 1967, pp.1171-1174.

Table of Contents