A PROBLEM-SOLVER/TMS ARCHITECTURE FOR GENERAL CONSTRAINT SATISFACTION PROBLEMS

by

Albert Croker

and

Vasant Dhar

Information Systems Department
Leonard N. Stern School of Business
New York University
90 Trinity Place
New York, NY 10006

December 1988

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

Working Paper Series

CRIS #191
GBA #89-1

Table of Contents

Abstract

Constraints, in various forms, are ubiquitous to design problems. In this paper, we provide a formal characterization of a generalized constraint satisfaction problem (CSP) that can be used to model many types of design/planning problems, and the architecture of an imlemented reasoning system for solving this problem. The architecture includes a truth maintenance system (TMS) which is specifically designed to reason about the relationships expressed in the constraints as a problem solution evolves. The CSP consists of two types of data. The first type of datum corresponds to assignments that are handled by the problem solver, and the second type corresponds to constraint terms handled by the TMS. The dependency network, representing the relationhips among constraint terms, is static and generally quite small, depending on the number of constraint terms. Also, justifications are never manipulated (only evaluated). This results in an architecture that makes efficient use of both space and time. The need for efficient TMSs, even though these might deal only with certain classes of problems, is underscored by the fact that general purpose TMSs have often been found to be highly inefficient for solving large problems. We also show how certain instances of the generalized CSP can be formulated as an integer programming problem, special cases of which can be solved efficiently using mathematical (integer) programming techniques.

1. Introduction

Design problems arise in a variety of domains. Solving such problems generally requires a hierarchical 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 such as designing a fugue (Reitman, 1965), a house (Alexander, 1964), an engineered artifact (Simon, 1973), or a business plan (Dhar and Pople, 1987). Domain expertise is involved in deciding how best to decompose the problem, in generating alternatives, recognizing constraints among them, and in resolving conflicts among the parts in a way that least impairs the quality of the overall design. Abstractly, the design problem can be viewed as a process of *constraint satisfaction.*

In this paper, we present an architecture of a reasoning system for a certain class of constraint satisfaction problems. This class of problems is characterized by a decomposition of the problem into discrete sets of competing alternatives called choice sets. The alternatives are defined in terms of attributes that characterize the choice set. In addition, constraints defined in terms of choice set attributes restrict the space of design solutions. Our reasoning system consists of two components: a problem solver that contains domain knowledge, and a truth maintenance system (TMS) 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 TMS and a problem solver, considerable simplicity in the TMS architecture and efficiency in its status assignment algorithms is achieved. We provide a precise characterization of the overall reasoning process by describing the algorithms corresponding to the problem solver and the TMS. We also contrast our reasoning system with those that might be designed using other TMSs. Specifically, we show some of the advantages of our architecture that result from exploiting the structural characteristics of the problem.

We should point out that our reasoning system does not model the entire process of problem decomposition and constraint definition. We assume that the choice sets and constraints have been defined for the problem solver. In effect, we are modeling only the constraint satisfaction process component involved in solving design problems.

2. Constraint Satisfaction Problems

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.

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, and, therefore, each processor in the set has an associated value for this attribute.

An *assignment* for **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 from the set of assignments for **X**. That is, $C_j: \times_{i=1}^{n} X_i \rightarrow \{T,F\}$. An assignment X for **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 **X** is called a *satisficing assignment* for the CSP characterized by the set of choice sets **X** and constraint **C** if $\forall\ C_i \in \mathbf{C}$, $C_i(X) = T$.

We specify constraints in the form

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

where each constraint term t_i is a Boolean-valued expression over a set of constants and variables. We call constraints expressed in this form *dependency constraints*. Each constraint term 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

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

We thus call each term that occurs to the right of the arrow in a constraint an *antecedent* term, and the term to the left the consequent term. 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. A constraint that has no antecedent terms is called a *premise* constraint. The consequent of a premise constraint must always hold.

With this notation, 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.

The above problem description is similar to that of Dechter and Pearl (1988), Mackworth (1977), and Nudel (1983), with some important differences. First, we consider n-ary instead of binary constraints. Secondly, the form of these constraints and the choice sets play a central role in how responsibilities are divided between the problem solver and TMS components of the reasoning system.

3. System Architecture

The system that we describe here incorporates a TMS, and is designed to be used for solving problems that can be formulated as CSPs. The TMS has had a significant impact on the resulting architecture. In this section we discuss this architecture, and the TMS that we have developed and used in this system. The system that we describe is one component of a larger system that we have developed for modeling and analyzing situations in support of decision making, which we view as a form of design.

The inclusion of a TMS in a problem solving system suggests a partitioning of that problem solving system into two well-defined subsystems: a problem solver, and the TMS. This partitioning permits an unambiguous assignment of specific responsibilities to each component. The architecture of the CSP solving system that we describe here is consistent with this approach, and is composed of a problem solver subcomponent and a TMS subcomponent. We describe each of these subcomponents below.

3.1. 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 becomes more limited in the future actions that it may take as a result of the problem's characterizing set of constraints. 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 by retracting some currently selected alternatives, and substituting other alternatives from the same choice sets. This process allows a set of beliefs to grow non-monotonically.

Corresponding to each choice set X_i the problem solver maintains a *selection* variable $\mathbf{X_i}$ that is used to designate the alternative that it has selected from that choice set. This compound variable consists of one component, designated $\mathbf{X_i}.A_{i,j}$, for each attribute $A_{i,j}$ over which the associated choice set is defined. During problem solving this variable specifies the alternative that has been selected from the associated choice set through its components, each of which denotes an attribute value of that alternative.

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 $\mathbf{X_i}$, we represent this initial state as $\mathbf{X_i}$ = **UNKNOWN**. When the problem solver selects an alternative from a choice set it sets each of the attribute components of the associated 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 we allow a user to bias the search by specifying a preference for the order in which choice sets, and alternatives withing choice sets, will be considered. For the purposes of this paper this order is not relevant and will not be discussed further.

Once the problem solver has selected an alternative form a choice set it must then determine a new set of relationships (i.e., limitations) that must be 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 TMS.

3.2. The Truth Maintenance System

The TMS subcomponent is designed to be separate from, but interact closely with the problem solver. With respect to control, the TMS is subordinate to the problem solver. Specifically, with each new belief communicated to it by the problem solver, the TMS 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 TMS in carrying out its task is a *constraint term node*. With one exception the TMS 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. (The term defined by 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 beliefs of the problem solver, the relationship expressed by t_i must hold, and, equivalently, that expressed by $\neg t_i$ 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 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 sub components: 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 TMS 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 also consists of two components: 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 the 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.

Alternatively, 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. The system notifies the user of such inconsistencies.

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.

4. Implementation

In this section we describe the data structures and algorithms used in implementing the overall problem solving system. These descriptions are specified in such a way as to adhere to the principle that the TMS module be distinct from the problem solver module in the overall reasoning system in the overall reasoning system (McAllester, 1982). Each of the algorithms, presented here in the form of a function or a procedure, comprise one or the other of the two modules.

The descriptions of the data structures and algorithms that we provide here 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 on structured and object-oriented language conventions.

4.1. 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 alternative 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 indicating 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**. An index 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 TMS 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 TMS 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 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 TMS throughout the course of the problem solving task.

[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**.

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 TMS 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.

4.2. Control Regime

The overall reasoning process exhibited by the interaction of the problem solver and TMS can be characterized as a "heuristic-deduction" cycle. The problem solver, using some heuristic, selects some alternative from a choice set. Based on this selection and the set of problem constraints the TMS 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 TMS, control passes back to the problem solver and the cycle resumes. If any violations are detected, the TMS 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 TMS-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.   TMS-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 TMS-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 **TMS-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 TMS knows will lead to an inconsistency. (Each untenable combination of selections, called a *nogood*, when first detected by the TMS is added to a list. This list of nogoods is accessed by **TMS-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 TMS of this selection through the invocation of the TMS module **TMS-PROPAGATE**. This module, which is described below, controls the constraint propagation function of the TMS. 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 **TMS-PROPAGATE** provides, through its second argument, information about the combinations of selected alternatives that led to the contradictions so that the problem solver can take appropriate action to alleviate the problem before continuing the task of selection alternatives.

The TMS is organized as a set of modules, each performing a specific aspect of the overall TMS function, and serving as an entry point to the TMS from the problem solver. One of these modules, **TMS-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 TMS-PROPAGATE (cs, conflict-sets)

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

The procedure **TMS-PROPAGATE** serves as an overall control module for the propagation function of the TMS. This module invokes the procedure **TMS-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 TMS 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 **TMS-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 TMS-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 TMS-JT-PROPAGATE (x, conflict-sets)
         else
L12.        for each const-term-node x in f-conseq
L13.           do TMS-JT-PROPAGATE (x, conflict-sets)
```

Step L1 of **TMS-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 other than *unknown*, and by by Step L1 must be 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, **TMS-LT-PROPAGATE** attempts to propagate the newly derived c-term-label value forward by invoking **TMS-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 t-conseq component if the c-term-label value is *false*.

The function **CONFLICT-ASSUMPTS**, when presented with a c-term-node from 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 TMS module **TMS-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 the 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_A 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 **TMS-LT-PROPAGATE** and **TMS-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 TMS-JT-PROPAGATE shown below is used by the TMS 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 **TMS-JT-PROPAGATE** using each of the nodes in either the t-conseq or the f-conseq component.

Procedure TMS-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 TMS-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 TMS-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 TMS-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 TMS-JT-PROPAGATE (x, conflict-sets)
```

The function **SATISFIED** used by **TMS-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 **TMS-PROPAGATE**, the TMS provides the problem solver with two other entry modules: **TMS-RETRACT** and **TMS-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 **TMS-PROPAGATE**, and will not be further elaborated on here.

The module **TMS-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 TMS modules, **TMS-NOGOOD-VIOLATION** does not access any of the c-term-nodes that make up the dependency net.

5. Relationship to Other Work

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 for and 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 first part 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. In the second part we describe how the general problem can be modeled using other AI approaches, in particular, other truth maintenance systems.

5.1. Special Cases: Integer Linear Programming

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 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 as the following inequality:

$$c_1x_1+...+c_nx_n \geq 1-n(c)$$

where c is a row vector and x 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 x_i appears, does not appear, or $\neg x_i$ 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. For example, in a business plan purchased long term capital assets might not be changeable whereas other decisions on how to deploy manpower and other resources might be relatively flexible. In such problems, once a plan is in place, decision makers make incremental decisions in the context of an evolving set of constraints.

For the types of problems described above, the limitations of OR techniques can be overcome to a large extent by coupling an optimizer to a TMS. The problem solver/TMS architecture we have implemented can be coupled easily 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 TMS.

Conducting a what-if analysis is then straightforward since the TMS can compute the impacts of changing decisions. A change can either "go through" (not require making changes in other parts of the plan/design), or result in violated constraints that are identified by the TMS. In the latter case the TMS 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.

5.2. Artificial Intelligence Methods

It is also 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, 1987), or the ATMS (de Kleer, 1986). Using these systems requires a recasting of the problem and a carefully designed problem solver.

We encountered two types of problems with Doyle's TMS. Firstly, we find the rationale for non-monotonicity inappropriate for modeling choice problems where an alternative is selected in preference to its competitors on the basis of a comparison by the problem solver as opposed to a lack of belief in its competitors. Perhaps more importantly, this scheme leads to an embedding of control information in the justification structure to be a liability, resulting in an inappropriate separation of control responsibility between the problem solver and the TMS.

In the remainder of this section, we comment on some of the differences between our reasoning system and those that might result by using two other well known TMSs, namely, RUP -- a justification-based TMS, and ATMS -- an assumption-based TMS.

5.3. 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 designs 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 c.label becomes false whereas c.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).

5.4. 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,..., C_n}, specifies oneof 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 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.

6. Conclusions

The reasoning architecture we have described differs in several ways from others involving truth maintenance systems. Fundamentallly, the problem solver and the TMS do not operate on the same data. The datum manipulated by the problem solver is the choice-set selection, with one selection made per cycle. The TMS performs constraint propagation but does not make selections; the data it manipulates are the constraint term nodes (i.e. the constraints). In effect, the decision making responsibility is that of the problem solver whereas the constraints that guide decision making are part of the TMS.

Note that constraint terms can be very general, exressing 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 TMS determines whether such relationships (and hence the constraints they make up) hold.

The architecture exploits the structure of the problem in several ways. First, the number of constraints of the form above is small compared to the size of the search space; thus the dependency network maintained by the TMS is small. Second, since the dependency network directly represents the constraints which are often completely specifiable at the start, it is compiled before problem solving begins and remains static. A secondary source of efficiency is that the node labels contain embedded queries to the problem solver and by evaluating the label whenever necessary, the TMS is able to determine whether a datum holds in some solution state maintained by the problem solver. In effect, the labels function as demons (like RUP noticers) which "fire" when necessary (when the node is involved in constraint propagation).

The architecture also offers considerable advantages from a knowledge engineering standpoint. A major obstacle for knowledge engineers and users of complex reasoning systems is one of determining how to construct the problem solver so that the responsibilities between it and the TMS are divided appropriately and so that the system is exhaustive. In practice with reasoning systems that employ a TMS, we have found that the importance and difficulty of these tasks (in particular the first) is underestimated, and often requires the user to be familiar with the inner workings of the reasoning system. In contrast, we have observed that users are able to specify declaratively the various knowledge components of their constraint satisfaction problem very rapidly 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 TMS are completely transparent to the user. In effect, the complexity of the reasoning system is largely hidden from the user. We feel that this is an important consideration that must be addressed if complex reasoning systems are to be used in 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.

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.

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

de Kleer, J., Problem Solving With the ATMS, *Artificial Intelligence*, vol 28, no.2, March 1986. (1986b)

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.

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

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.

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.

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.