September 1969

REF-ARF: A SYSTEM FOR SOLVING PROBLEMS STATED AS PROCEDURES

by

Richard E. Fikes*
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Submitted for publication in the AI Journal.

Artificial Intelligence Group
Technical Note 14

*The research reported in this paper was conducted while the author was at Carnegie-Mellon University. Dr. Fikes is now a Research Mathematician at Stanford Research Institute.
TABLE OF CONTENTS

INTRODUCTION

CHAPTER I
Input Language for a General Problem Solver  2
   The Task of Designing an Input Language  2
   The REF Input Language  5
   Stating Problems in REF  9
   Expansions to REF  15

CHAPTER II
Interpreting a REF Problem Statement  17
   The Translation Problem  17
   The ARF Interpreter  20
      Use of Variables and Constraints  20
      Size of the Interpreter's Search Space  24
   Summary  30

CHAPTER III
Solving ARF's Constraint Satisfaction Problems  32
   Designing a Constraint Satisfier  32
      Value Assignment Methods  32
      Constraint Manipulation Methods  37
      Combining Constraint Manipulation and Value Assignment  40
      The Resulting Design  41
   ARF's Expression Manipulation Methods  42
      Expression Simplification  43
      Entering a Value into a Context  46
      Constraint Manipulation Methods  47
      Deleting an Element from the Range of a Variable  52
      An Example of ARF's Expression Manipulation  53
   Conclusions  56
CHAPTER IV

Guiding the Interpreter's Search 58

ARF's Search Strategy 58
  Recognition of Equivalent Contexts 59
  Deletion of Contexts containing Contradictions 62
  Selecting the "Closest" Object to the Goal 63
  A Proposal for Extending ARF's Search Strategy 69

CHAPTER V

A Sampling of Problems 75

Boolean Constraint Satisfaction Problems 76
  A Sorting Problem 76
  A Magic Square Problem 77
  The Picnic Problem 77
  A Packing Problem 79
  The Instant Insanity Problem 82
  The Couples Tennis Problem 83
  The Eight Queens Problem 87
  The Confusion of Patents Problem 88

Process Constraint Satisfaction Problems 91
  The John and Mike Tennis Problem 91
  A Crypt Addition Problem 93
  A Sequence Formation Problem 93
  A Hypothesis Formation Problem 95

Heuristic Search Problems 98
  The Monkey Problem 98
  A Waterjug Problem 99
  A Programming Problem 99
  The Missionaries and Cannibals Problem 101

CHAPTER VI

Summary 106

REFERENCES

ACKNOWLEDGEMENTS

APPENDIX

ARF's Output for the Magic Square Problem
INTRODUCTION

We wish to describe here our efforts to pursue the development of computer programs capable of displaying intelligent problem-solving behavior. Our pursuit has proceeded in an experimental mode in that we have written and debugged a program which embodies our ideas and have used our experience with the program to motivate and develop new ideas. This paper is both a description of the program and a discussion of those ideas and issues from the project which seem important to us at this time.*

If we are given a particular problem to solve, then the value of a problem-solving program to us is related to the ease with which we can state the problem to the program and the effectiveness of the program for finding a solution. In general, the value of a problem solver can be measured in terms of the range of problems which can be easily stated in its input language and the range of problems for which its problem-solving methods are effective. Hence, to build a good problem-solving program we must be concerned both with the generality of its input language and with the effectiveness and generality of its problem-solving methods. We will begin by considering the generality of our problem solver's input language (which we call REF), and then proceed to a discussion of the methods contained in the problem-solving program (which we call ARF).

*This project includes the research for my Ph.D. thesis [6] and one year of research after completion of the thesis. In [5] we gave a description of the REF language and an introduction to the ARF problem solver; ARF's methods and behavior were not described. This paper is a detailed report on both REF and ARF, as was the thesis. Sections of this paper which describe work done after completion of the thesis include "ARF's Expression Manipulation Methods," "Guiding the Interpreter's Search," and "A Sampling of Problems."
I. INPUT LANGUAGE FOR A GENERAL PROBLEM SOLVER

THE TASK OF DESIGNING AN INPUT LANGUAGE

The designer of an input language for a general problem-solving program faces two conflicting demands. First, the user of the language must be able to state easily and in a natural manner as diverse a class of problems as possible. Hence the language should have sufficient descriptive power for natural representation of the elements in a complex problem situation. Second, a computer program must be effective at solving the range of problems statable in the input language. Hence the language should be precisely defined and its structure be simple enough to allow algorithmic interpretation of problem statements.

Some research efforts have focused on the user's demands for generality and naturalness in the input language by attempting to provide a close approximation to English for stating problems (e.g., [1] and [3]). Although promising results have emerged from such efforts, it is currently the case that either the subset of English which is allowable falls far short of providing the freedom of expression we are accustomed to when using English and/or that the range of problems which the program can effectively translate is unacceptably restrictive.

Others have focused on the demands of the program designer, i.e., effective problem-solving methods be applicable. An example of such an approach is the use of predicate calculus theorem-proving programs as problem solvers (e.g., [9]). These programs require that a problem be formulated as a theorem to be proved in the predicate calculus. This formulation task is often difficult and cumbersome, but predicate calculus is a formally defined language with a simple structure and therefore lends itself to the application of mechanical theorem-proving techniques. Although impressive progress has been achieved in the development of predicate calculus theorem-proving
programs, the restricted domain of problems which can be easily expressed in the input language severely limits their value.

Another example of a research effort which chose a problem-statement language adapted more to the problem-solving methods of the program than to the user's demands is the work on the General Problem Solver (GPS) [4]. The program's methods assume that a problem exists in a heuristic search paradigm, i.e., as an initial object, a final or goal object, and a set of operators which transform a given object into a new object. The user is required to formulate his problem in the heuristic search paradigm and state it in that form. As is the case with the predicate calculus problem solvers, this translation task is often difficult and cumbersome. The GPS language does, however, allow easier descriptions of complex objects, predicates, and functions than does the predicate calculus so that the user has a richer language in his employ for stating problems. In addition, GPS allows the user to make statements concerning properties of the functions and objects contained in a problem statement (i.e., GPS allows the definition of "differences" between objects and specification of a "difference table" which indicates to the program which differences are affected by which operators). Although with respect to the predicate calculus problem solvers GPS provides a decrease in the difficulty of stating problems, the complexity and informal character of the GPS input language causes an accompanying increase in the complexity of the methods and overall structure of the GPS program.

For our problem solver we have attempted to form an input language which provides a more equitable compromise of the user's demands and the program designer's demands than does any of the existing problem solvers. We have adopted a formal language which can be algorithmically interpreted so that effective problem-solving methods can be applied, and yet we have attempted
to provide a maximum of representational power so that a diverse class of problems may be stated in a natural manner.

We have created our language for stating problems by extending a language designed for stating solution procedures to problems (i.e., a programming language). One still writes procedures in the extended language, but these procedures may define problems by indicating a selection from a space of potential solutions and then a verification that the selection satisfies the requirements of the problem for a solution (i.e., select \( x \in X \) such that \( P(x) \) is true). The extensions involve the addition of a `select` function and a `condition` statement to the programming language.

The `select` function provides the facility for indicating that a selection is to be made from a space of potential solutions. It requires arguments which define a set and the value of the function is an element of that set. For example, if the base programming language is ALGOL, then the statement '\( B \leftarrow \text{select}(0, 9) \)' could mean that \( B \) is to be assigned as a value an integer in the range 0 to 9.

The `condition` statement is used to state a Boolean expression which must have `true` as a value. By using the `condition` statement one can verify that a selection is a solution to the problem being stated. For example, consider the problem of finding two integers in the range 0 to 9 whose sum is 15. Using ALGOL as the base language and the integer-valued `select` function mentioned above, the problem could be stated as follows:

```algon
begin;
integer B,C;
B \leftarrow \text{select}(0, 9);
C \leftarrow \text{select}(0, 9);
\text{condition} B+C=15;
end;
```
The procedure (or problem statement) says, in effect, select integer values in the range 0 to 9 for each of B and C such that their sum is 15.

We will refer to such a problem statement as a nondeterministic procedure. This terminology is derived from a paper on nondeterministic algorithms by Floyd [8] and the concept of a nondeterministic automaton as introduced by Rabin and Scott [11]. The use of the term nondeterministic is not meant to impute a probabilistic nature to the language, but rather that one may include freedom of choice or multiple paths in procedures written in the language. Any processor which executes a nondeterministic procedure must make choices (or selections) so that all condition statements are satisfied and so that the procedure terminates.

By extending a programming language in this manner we can form a problem statement language which allows the user to employ the data structures of the programming language to represent objects or to model situations and to employ the control structure of the programming language to represent functions and predicates of those objects.

We will explore the representational power and scope of nondeterministic input languages by describing REF, the particular input language created for our problem-solving program, and then considering how it can be used for stating various types of problems.

THE REF INPUT LANGUAGE

The syntax of REF is defined in Fig. 1.1 in Backus Normal Form [—]. The semantics of REF is described in the following paragraphs.
Each identifier in a REF problem statement names a describable vector; i.e., if B is an identifier then 'B[3]' refers to the third element of the vector B and 'COLOR of B' refers to the value of the attribute COLOR of the identifier B. Vector elements and attribute values may be either integers or identifiers. Vector element numbers may range over the positive integers. The conventional notion of associating a single scalar value with an identifier (e.g. ALGOL's X := 3) has no meaning in REF since each identifier names a data structure (the describable vector) rather than a single value. To provide such a facility the value of the attribute SCALAR.VALUE is treated as a special case syntactically and can be denoted by enclosing the identifier in angular brackets, e.g. <B>.

Integer valued expressions may be formed with the operators + and -. Boolean valued expressions may be formed with the relations =, <, =, >, and V. Expressions are evaluated using an operator hierarchy in the standard left to right manner with one exception. When more than one of function appears sequentially in an expression, as in 'A of B of C', evaluating is from right to left to preserve the intuitive meaning of the expression, i.e., 'A of (B of C)'. The following example illustrates the evaluation of a REF expression.

Assume that <A> is B, <I> is 1, B[1] is C, <D> is E, <E> is F, F of G is H, and C of H is 3. Then evaluation of the expression <A>[<I>] of <D>> of G proceeds as follows:

```
<A>[<I>] of <D>> of G
B[1] of <E> of G
  C of F of G
    C of H
```
The **set** statement is the basic assignment statement in REB. The left side expression in a **set** statement indicates the vector element or identifier-attribute pair which is to be assigned a value; e.g., to set the scalar value of identifier $B$ to be 3 one writes '$set <B> to 3;'. The **set vector** statement is a multiple assignment statement which sets the values of the first $n$ elements of the vector indicated by the statement's left side expression; e.g., to specify that $M$ is to be the vector $(2,3,5,7,11)$, one would write '$set vector M to 2,3,5,7,11;'.

The **for** statement is used to define a loop in a problem statement. The loop consists of all the statements following the **for** statement up to and including the statement having the label indicated in the **for** statement. The first time through the loop the scalar value of the identifier indicated in the **for** statement is assigned the value 1, and each subsequent time through the loop this value is incremented by one. The value of the expression following the right arrow in the **for** statement defines the number of times the loop is to be interpreted.

The **if** statement is used to indicate a conditional transfer. If the value of the Boolean expression in the statement is true then the interpreter is to transfer to the statement indicated by the label; otherwise, interpretation continues sequentially.

The computed **goto** statement is used to indicate an n-way branch. The value of the integer expression indicates the statement to which control is to be transferred as follows: if the value of the integer expression is $j$ then control is transferred to the statement whose label is the $j$th member of the parenthesized label list. For example, if $<I>$ is 3 then the statement 'goto (L1,L2,L3,L4)<I>;:' will cause interpretation to continue at statement L3.
excl is a Boolean valued function which is used to indicate that a set of values must be distinct; i.e., excl(A1,A2,...,An) is true if and only if Ai ≠ Aj for all i and j such that i ≠ j.

Interpretation of a REF procedure halts when the end statement is reached. The end statement may be labeled and transferred to from a goto or an if statement.

In a REF procedure one may indicate that the problem solver is to select an integer from some range by using the select function in an assignment statement or a computed goto statement. The function takes two integer arguments that define the interval range from which an integer is to be selected. For example, to specify that the problem solver is to assign an integer to <B> from the range 0,1,...,9 one would write 'set <B> to select (0,9);'.

The condition statement is used to request verification of the selected solution. The statement indicates to the interpreter that the selections it has made must satisfy the statement's Boolean expression (i.e., give it the value 'TRUE').

This completes the description of the REF language. The language is a simple one lacking real valued expressions, numeric operators such as multiplication and exponentiation, block structure, and subroutines. These deficiencies limit REF, but there is sufficient power in the language to allow natural representation of an interesting range of problems.
STATING PROBLEMS IN REF

In this section we will explore the characteristics of REF problem statements by considering the representation of four different forms of problems.

First, consider the class of constraint satisfaction problems where in the statement of the problem "select \( x \in X \) such that \( P(x) \)" \( X \) is a finite set and \( P(x) \) is a conjunction of Boolean expressions. We will call this the class of Boolean constraint satisfaction problems. Members of the class include problems of solving sets of simultaneous equations, finding feasible solutions to resource allocation problems, sorting lists of numbers, and various puzzles.

The REF select function and condition statement provide natural mechanisms for stating a problem from this class. After representing the objects or quantities of a problem as REF data structures one writes a REF procedure consisting first of a sequence of assignment statements to indicate the selection of an \( x \) from the solution space \( X \) and then a sequence of condition statements to indicate the constraints which the selected \( x \) must satisfy.

A typical example of a Boolean constraint satisfaction problem is the magic square puzzle stated in English in Figure I.2a. For this problem \( X \) can be considered the set of all 3-by-3 matrices containing the integers 1 through 9 and \( P(x) \) the eight summation equations and the requirement that each integer in the solution matrix be distinct. Figure I.2b shows a REF statement of the problem using this formulation. The REF statement uses the vector \( M \) to represent the magic square matrix, a set vector statement to select the solution, and a sequence of condition statements containing the constraints to verify the selections as a solution.
For many constraint satisfaction problems it is not convenient to express the verifying predicate \( P(x) \) as a simple conjunction of Boolean expressions. Since REF provides the full power of a programming language, one can write an arbitrary process in a REF problem statement to verify that a selected \( x \) is a legitimate solution. This availability of a process writing capability is the basis for REF's distinctive representational power.

We will call constraint satisfaction problems which use a process to verify a solution, _process constraint satisfaction problems_. A broad range of problems can be stated naturally in this form. For example, the problem of defining a perceptron [13] which will discriminate between two classes of objects can be described as a problem of selecting a set of parameters (i.e., the weights) for a process which applies each of the objects to the perceptron and tests to see if the correct discrimination is made.

The crypt-addition problem stated in English in Figure I.3a can be expressed naturally as a process constraint satisfaction problem. Although it is possible to state the verifying conditions for this problem as a single equation (i.e., \( 1000*S + 100*E + \ldots = \ldots + 10*E + Y \)), the typical statement of such problems suggests the column-by-column addition process and, in fact, most humans will exploit the properties of that process when solving the problems.

We present in Figure I.3b a REF procedure which states the crypt-addition problem as a process constraint satisfaction problem. In the procedure the vectors \( A1 \) and \( A2 \) represent the addends, the SUM vector
represents the sum, and the L vector lists each of the letters to be assigned values in the problem. The loop terminating at L1 selects values for each of the letters. The condition statements following L1 verify that each selected value is unique and that there are no leading zeros in the solution. The loop ending at L2 represents the addition process and indicates that the solution must satisfy the conditions imposed by that process. Each time through the loop a column is added, beginning with the rightmost one. The final condition statement indicates that the value of M must equal the carry into the leftmost column.

Now consider a third class of problems consisting of those which can be naturally stated in the heuristic search paradigm introduced by Newell and Ernst [19]. These problems may be stated by describing an initial object, a desired final object, and a set of operators for producing new objects from given objects. Newell and Ernst argue the generality of this paradigm and show that theorem proving, integration, sentence parsing, letter series completion, and various puzzles are members of the class.

One can state such problems in REF by writing a procedure which has the general form shown in the flow chart of Figure I.4. The procedure begins by creating a data structure which represents the initial object of the problem. Following this initialization phase is a loop which can be exited only when the desired final object has been realized. Each time through this loop an operator is selected, tested for applicability, and applied. Operator application is followed by a test for the final object which results in either a continuation in the loop or a transfer to end.
A distinguishing characteristic of the REF procedures which state these heuristic search problems is that their termination is not guaranteed. That is, in the procedures we considered earlier there were only a finite number of possible calls on the select function and each possible combination of values for those calls led either to a condition statement whose Boolean expression was false or to termination at end. In contrast, REF procedures which describe heuristic search problems may cause cycling to occur in a loop with new calls on the select function being made during each cycle.

A typical example from this class of problems is the water jug problem [17] stated in English in Figure I.5a. For this problem an object can be described by an ordered pair of positive integers representing the number of gallons of water in each of the two jugs. Hence, (0,0) describes the initial object and any object whose description has a second element of 2 is a final object. There are six possible operators as follows:

1. Fill the first jug from the water source.
2. Fill the second jug from the water source.
3. Empty the first jug into the water sink.
4. Empty the second jug into the water sink.
5. Pour as much water as possible from the second jug into the first jug.
6. Pour as much water as possible from the first jug into the second jug.

Any of the six operators are applicable to any object.
We present in Figure I.5b a statement of the problem in REF. In that statement A represents the 8 gallon jug and B represents the 5 gallon jug. This REF procedure has the form discussed above with the initialization phase being lines 1-3, the operator selection at statement L10, the branch on the selected operator at line 5, the operator applications in lines 6-23, and the test for the final object at line 24.

For problems of this class which have a large number of operators and have complex applicability tests, one can often use the data structures and control structures of REF to form a model of the problem situation. For example, consider the monkey problem [16] stated in English in Figure I.6a and stated in REF in Figure I.6b. We have modeled the situation in the REF procedure by assuming three locations on the floor: X1, X2, and UNDER.BANANAS. These locations are sufficient to express the general situation where the monkey is not at the box and neither the monkey nor the box is under the bananas. We also assume two vertical locations: ON.FLOOR and ON.BOX. Only the monkey can change vertical positions and these two locations are sufficient to express the changes which can occur. The positions of the monkey and of the box are indicated in the procedure by a two element vector, the first element giving the horizontal position and the second element giving the vertical position. The initial object (or state) of the problem is defined in the procedure by placing the monkey at X1 on the floor and the box at X2 on the floor. We recognize nine operators in the problem as follows:
1.-3. The monkey walks to a location on the floor.

4.-6. The monkey moves the box to a location on the floor.

7. The monkey climbs on the box.

8. The monkey steps down from the box.

9. The monkey gets the bananas.

The desired final state is the one which results from application of the ninth operator, get the bananas.

This problem could be stated in REF using the form discussed above and indicated in Figure I.4. Such a REF statement would have a loop containing a nine-way branch and would have applicability tests preceding the application of each operator. For example, the applicability test for the GET BANANAS operator would be the following two condition statements:

```
condition MONKEY[2]=ON.BOXL;
condition MONKEY[1]=UNDER.BANANAS;
```

Alternatively, one can construct the REF procedure as a model so that at each step only those operators which are applicable can be selected.

This is the strategy adopted in writing the procedure of Figure I.6b. For example, if the monkey climbs on the box and is not under the bananas then he must step down. If he climbs on the box and is under the bananas, then he has the choice of getting the bananas or stepping down. This situation is represented in the REF procedure in lines 10-13.

We have seen in this section how problems are stated as nondeterministic procedures in REF. With respect to our discussion of the demands placed on the input language of a general problem solving program we see that although
REF does not provide the power and informality of English, the REF expression operators, data structures, and process writing facilities do provide a richer and more flexible language for describing problem situations than does either the predicate calculus or the GPS heuristic search input language. We have not yet discussed the methods used by ARF for solving problems stated in REF, but we shall show in our description of the program that the more complex input language does not prevent the successful application of effective problem solving methods.

EXPANSIONS TO REF

We observed earlier that REF has many limitations which restrict its representational power. In this section we will suggest some of the ways in which REF might be expanded to reduce those limitations.

Some of the more obvious features lacking in REF are multiplication and division operators and callable parameterized procedures (or subroutines). Besides the usual programming conveniences of using parameterized procedures, one could write procedures which state entire classes of problems. The input parameters to such procedures would define a particular problem in the class. For example, one could write a parameterized procedure for the class of crypt-addition problems (such as SEND+MORE=MONEY) which would accept as input two addend vectors and a sum vector. The use of such procedures to define problem classes sets up the possibility for the design of learning mechanisms in the problem solver since the program could use its past experience with a procedure in solving a problem defined by a new set of parameters to the procedure.
REF's representational power would be greatly enhanced by the addition of sets and set operations. One would want to include operations for adding and deleting set elements, forming conjunctions and disjunctions of sets, sequencing through the elements of a set in a for loop, etc.

The inclusion of sets in REF motivates an expansion of the select function. One would like to have a form of the select function which takes a single argument, the name of a set, and produces as a value an element of the set. Other desirable extensions of select would be to allow selection from an infinite range of the integers and the ability to use as an argument to select an expression whose value depends on previous select function calls.

There is no facility in REF for stating optimization problems. One might provide such a facility by including another form of the condition statement, namely a maximize or minimize statement. An expression whose value was to be optimized would be included in such a statement and it would be a numerical valued expression containing quantities which were previously set by calls on the select function. Such a statement would cause the problem solver to consider the entire set of selections satisfying the other constraints of a problem and determine one which optimized the expression of the maximize or minimize statement. Figure I.7 shows how the addition to REF of a maximize statement, a multiplication operator, and a select function capable of selecting a value from the infinite set nonnegative.integers would allow the natural statement of an integer programming problem. Figure I.7a shows the problem as stated in a textbook [11] and Figure I.7b shows the same problem stated in the extended REF.
These suggestions serve to illustrate the range of possibilities for the REF language. As we consider each expansion to the language we must ask whether we can include in ARF capabilities for handling problem statements which use the expansion. In fact, we have often found it profitable to proceed in our research with REF-ARF by focusing on the issue of what expansions to REF would increase its usefulness to the user, and then letting the suggested expansions motivate the new capabilities to be incorporated into the problem solver. For example, suggestions for expansions to REF given above motivate considering the addition to the program of capabilities for handling unordered sets, learning from repeated processing of parameterized procedures, and dealing with optimization problems.

II. INTERPRETING A REF PROBLEM STATEMENT

THE TRANSLATION PROBLEM

The first task in the design of a problem solver which uses REF as its input language is to devise ways of extracting the necessary information from a REF procedure to translate it into a form which will enable the application of effective problem solving methods. The task of designing automatic means for extracting semantic information from a procedure written in a programming language has been studied by McCarthy [15], Floyd [7] [8], King [13], and others. Most of these studies have been directed toward the development of automatic techniques for proving that a program does what the writer claims it does. Floyd's work with nondeterministic algorithms [8], however, is directly related to the task of translating REF problem statements
into a solvable form. He shows that a procedure written in a nondeterministic programming language can be automatically translated into a procedure in the base programming language which finds acceptable values for the select function calls by using a backtracking algorithm [10].

The backtracking algorithm conducts a depth-first search as follows. During normal execution of the procedure a call on the select function produces as a value the first integer in the range specified by the parameters to the call (e.g., the call 'select(0,9)' would produce the value 0). Execution continues until a condition statement is encountered whose Boolean expression is false or until end is reached. In the case where end is reached, the values assigned to the select function calls constitute a solution. In the case where a condition statement's Boolean expression is false, the algorithm backtracks to the last call of the select function and attempts to increment the value of that call by one. If the new value is in the range specified for the call, then execution again continues normally. If the new value does not belong to the range of the call, then a second backtracking step occurs to the next-to-last select function call. A similar attempt is made to increment the value of that call. The algorithm continues in this manner until a solution has been found or until the value of the first select function call cannot be further incremented.

The question arises as to whether we can program other translation schemes for REF problem statements to allow the application of methods more powerful than backtracking. Consider desirable properties for a translation of each of the example problems introduced above in Part I.

One would like a translation of the magic square problem (shown in Figure I.2) which would allow application of techniques for solving
simultaneous equations. Such techniques could be used to "eliminate variables" by expressing some of the matrix elements as functions of other matrix elements (e.g. $M[1]$ might be expressed as $15 - M[2] - M[3]$). It would also be desirable for the problem solver to determine in its translation that the order of the statements is irrelevant and that they can be considered by the program in any order.

For the crypt-addition problem (shown in Figure I.3) one would like the problem solver to translate the problem so that deductive methods similar to those used by humans when solving such problems could be applied. For example, one may conclude from the two leftmost columns that '10*M + O' is either equal to 'S + M' or to '1 + S + M' and therefore that $M$ is 1 and $S$ is either 8 or 9. A basic feature of a translation which allows the application of such deductive methods is that the problem solver not be constrained to consider the columns in order from right to left. Hence, we wish the problem solver to be able to free itself from the control sequence implied by the statements in the procedure.

For the waterjug problem (shown in Figure I.5) and the monkey problem (shown in Figure I.6) it would be desirable for the translation to allow the application of heuristic search methods. One method particularly relevant for these problems recognizes when two states in the search are semantically equivalent and prunes one of the two search tree branches containing the equivalent states. One would also like the problem solver to apply some form of means-ends analysis to help direct its search. For example, in the monkey problem such analysis is needed to generate the subgoal "move the box under the bananas."
The problem solver we have written contains a translation scheme which has many of the desirable features mentioned above. In the remainder of this part of the paper we will describe and discuss this scheme in detail.

**THE ARF INTERPRETER**

**The Use of Variables and Constraints**

ARF translates a problem stated in REF by using an interpreter which acts much like a standard programming language interpreter. The context in which the interpreter is operating at any given time is represented by a single data structure. This context structure contains a pointer to the next statement to be interpreted and contains the vector and the attribute-value pairs of each identifier which has been encountered during the interpretation.

The ARF interpreter differs from a standard interpreter in the way it deals with condition statements and calls of the select function. When it encounters a statement containing a select function call, it does not assign an integer value to the call. Instead it creates a variable to represent the selection and treats the name of that variable as the value of the call. (We denote the name given to the variable created by the $i$th call of the select function by $S(i)$.) When a variable is defined by the interpreter it is entered into the context along with a list containing each of the values in its range.

The use of variables in this way implies that when the interpreter evaluates a REF expression, the value may be another expression containing variables rather than an integer, identifier, true, or false. The interpreter can store these expressions in the context as vector elements or as values of attributes. For example, after interpretation of the
set vector statement in the magic square problem each element of the
vector M is a variable name rather than an integer, and the interpreter's
context structure has the following form:

Data Structure

M

Vector: S(1), S(2), S(3), S(4), S(5), S(6),
       S(7), S(8), S(9)

Variables

S(1)
Range:  1, 2, 3, 4, 5, 6, 7, 8, 9

S(2)
Range:  1, 2, 3, 4, 5, 6, 7, 8, 9
       ...
       ...
S(9)
Range:  1, 2, 3, 4, 5, 6, 7, 8, 9

When the interpreter encounters a condition statement, it evaluates
the Boolean expression. If the evaluation produces true, then no other
action is taken. If the evaluation produces false, then the interpreter
indicates that no solution exists and halts. If the evaluation produces
an expression containing variable names, then that expression is entered
into the context as a constraint on the values of those variables. Hence,
after interpreting the condition statements of the magic square problem,
the context has the following form:

Data Structure

M

Vector: S(1), S(2), S(3), S(4), S(5), S(6),
       S(7), S(8), S(9)

Variables

S(1)
Range:  1, 2, 3, 4, 5, 6, 7, 8, 9

S(2)
Range:  1, 2, 3, 4, 5, 6, 7, 8, 9

21
\[ S(9) \]

Range: 1, 2, 3, 4, 5, 6, 7, 8, 9

Constraints
\[
\begin{align*}
S(1) + S(2) + S(3) &= 15 \\
S(4) + S(5) + S(6) &= 15 \\
&\quad \cdots \\
S(3) + S(5) + S(7) &= 15 \\
\text{excl}(S(1), S(2), S(3), S(4), S(5), S(6), S(7), S(8), S(9))
\end{align*}
\]

When the interpreter reaches end, the context structure represents the completed translation of a problem. The problem is in the form of a Boolean constraint satisfaction problem in that there are a set of variables to be assigned values, a range of possible values for each variable, and a set of constraints which the variable values must satisfy. With the problem stated in this form, constraint manipulation methods can be applied which attempt to eliminate variables, reduce the range of variables, and derive contradictions. ARF contains a set of such methods which accept as input a problem stated in the form of a context structure produced by the interpreter. We will discuss in the next part how these methods are applied to find acceptable values for a context's variables. **Case Analysis and the Interpreter's Search Space**

The ARF interpreter is not always able to translate a problem stated in REF into a single constraint satisfaction problem represented as a context structure. The difficulty can occur during the interpretation of an if statement, a computed goto statement, or an assignment statement as described below.

In the interpretation of an if statement, evaluation of the Boolean expression in the statement may produce another expression rather than
true or false. When this situation arises, provision must be made for considering both the case where the branch is taken and the case where it is not. The interpreter can perform the required case analysis by saving a copy of the current context and either adding the Boolean expression to the current context as a constraint and taking the branch or adding the negation of the Boolean expression to the current context as a constraint and ignoring the branch. The saved context copy can be used at some later point in the processing for consideration of the second case.

During the interpretation of a computed goto statement evaluation of the integer expression in the statement may produce another expression, rather than an integer. In this situation each possible branch must be considered. As with the if statement, the interpreter can save a copy of the current context as it considers a branch and then return to consider another of the branches later in the processing. When the interpreter considers one of the branches as a computed goto statement, it adds to the context a constraint equating the statement's integer expression to the integer value which it must have for the branch to be taken.

In some instances it is necessary to produce multiple cases during the interpretation of a set or set vector statement. We will consider here only the interpretation of a set statement since interpretation of a set vector statement is defined in ARF as a sequence of set statement interpretations.

The interpretation task can be summarized as follows. The value defined by the set statement's right side expression is to be entered as the value of the data slot (i.e., the vector element or identifier-attribute pair) defined by the statement's left side expression. If the data slot being assigned a value already has a value, then that value will be
replaced by the new value. All references to the old value in the context and in the statements right side expression must be found and replaced by the old value before the new value is entered.

Two situations can occur during this process which require case analysis. The first is that the interpreter may not be able to determine the data slot indicated by the statement's left side. For example, consider interpretation of the statement 'set B[<I>] to 3' with the following input context:

Context C0

Data Structure

B
Vector: null, 7

I
<I>: S(1)

Variables

S(1)
Range: 1, 2, 3, 4, 5

Constraints: none

The interpreter faces the issue of whether or not to replace the value 7 in the B vector with the value 3. Since the value of S(1) has not yet been determined, the interpreter must consider both the case where it is 2 and the 7 is replaced by 3 and the case where it is not 2 and no replacement occurs. Hence, the following two contexts are output at the completion of the interpretation:

Context C1

Data Structure

B
Vector: null, 3

I
<I>: 2

Context C2

Data Structure

B
Vector: null, 7

I
<I>: S(1)
The second situation which can cause case analysis occurs when the interpreter cannot determine if a reference to a data slot in the context or in the statement's right side refers to the same data slot as the statement's left side. For example, consider interpretation of the statement 'set B[1] to 6' with the input context C2 from the example above. The interpreter cannot determine if B[S(1)] in the context's constraint is a reference to B[1]. Hence, both the case where it is and the case where it isn't must be considered so that two contexts result from the interpretation as follows:

Context C2.1

Data Structure
B
Vector: 6, 7
I
<i>: 1
Variables
S(1)
Value: 1
Constraints: none

Context C2.2

Data Structure
B
Vector: 6, 7
I
<i>: S(1)
Variables
S(1)
Value: 2
Range: 1, 3, 4, 5
Constraints
B[S(1)] = 3

Note that in either of these two situations the case analysis can be n-ary rather than binary. For example, if the B vector had k elements in context C0, then k + 1 cases could be produced during the interpretation of the statement 'set B[i] to 3'. The maximum number of cases which
could be produced by this interpretation would be limited to five by
the range of $S(1)$.

When the interpreter does case analysis during the processing of a
REF procedure, it effectively translates the original problem into
multiple constraint satisfaction problems. The definition of any one
of these subproblems is completed whenever the interpreter moves a
context to end. The constraint satisfaction problem solving methods can
then either find a solution to the problem represented by the context or
determine that no solution exists. A solution to any of these problems
is a solution to the original problem.

The crypt-addition problem stated in REF in Figure I.2b is one for
which case analysis is required. Each time the L3 loop is interpreted a
binary branch is required at the if statement. Hence, there are poten-
tially 16 cases, or subproblems, to be considered. These cases represent
the 16 possible combinations of carry values produced during the addition
process. Figure II.1 shows the context at end for the case where all the
carry values are 0.

For some problems the case analysis required is so great that the
major difficulty in finding a solution is finding a context which can be
interpreted to end. To solve such problems ARF needs methods for guid-
ing the interpreter as to which case to pursue first when case analysis
occurs, and once a case is chosen, how long to pursue the case before
returning to consider another one.

Consider the nature of this interpreter guidance task for the monkey
problem as stated in Figure I.6b. In particular, consider the formation
of the subproblem which contains the four step solution to the problem.
As interpretation begins the first variable is created at statement WALK
and the first case analysis is required at the if statement of line 7. One of the contexts produced at line 7 represents the case where the monkey has walked to X2 and the other represents the case where he has walked somewhere else. The context where the monkey is at X2 will contain the constraint 'X[S(1)] = X2'. If interpretation proceeds with this case, then the second variable is created at statement L1 and three cases are produced. The context which branches to MOVE.BOX will contain the constraint 'S(2) = 3'. If interpretation proceeds with this context, then at statement MOVE.BOX the third variable is created and at statement Ll the fourth variable is created and three new cases are produced. The context which branches to CLIMB will contain the constraint 'S(4) = 2'. If interpretation proceeds with this case, then two cases are produced at the if statement of line 10. The context which does not branch to STEP.DOWN will contain the constraint 'X[S(3)] = UNDER.BANANAS'. If interpretation proceeds with this case, then the fifth variable is created and two cases are produced at the computed goto statement of line 11. The context which branches to GET.BANANAS will contain the constraint 'S(5) = 1'. If interpretation proceeds with this case, then end is reached and a solution attempt can be made on the problem represented by the case. The context has the following form at end:

Data Structure

X
  Vector: X1, X2, UNDER.BANANAS

Y
  Vector: ON.FLOOR, ON.BOX

Monkey
  Vector: X[S(3)], ON.BOX

Box
  Vector: X[S(3)], ON.FLOOR
<table>
<thead>
<tr>
<th>Variables</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(1)</td>
<td>X[S(1)] = X2</td>
</tr>
<tr>
<td>Range: 1, 2, 3</td>
<td>S(2) = 3</td>
</tr>
<tr>
<td>S(2)</td>
<td>X[S(3)] = UNDER, BANANAS</td>
</tr>
<tr>
<td>Range: 1, 2, 3</td>
<td>S(4) = 2</td>
</tr>
<tr>
<td>S(3)</td>
<td>S(5) = 1</td>
</tr>
<tr>
<td>Range: 1, 2, 3</td>
<td></td>
</tr>
<tr>
<td>S(4)</td>
<td></td>
</tr>
<tr>
<td>Range: 1, 2, 3</td>
<td></td>
</tr>
<tr>
<td>S(5)</td>
<td></td>
</tr>
<tr>
<td>Range: 1, 2</td>
<td></td>
</tr>
</tbody>
</table>

ARF's constraint satisfaction methods have sufficient power to solve the above problem with only a modest amount of effort. Hence, ARF's difficulty in finding a solution to the monkey problem is deciding which case to interpret at each branch statement.

In general, the task of guiding the interpretation when case analysis occurs can be formulated as a search problem to which heuristic search methods can be applied. The objects of this search problem are the context structures and the operators are the statements in the REF procedure. The initial object is an empty context at begin and a final object is a context at end whose constraints have been satisfied. The interpreter is used to apply the operators to the objects. Although there is never more than one operator applicable to a given object, the search tree grows exponentially because case analysis can cause an operator to produce more than one object as output. The operators defined by each type of REF statement are given in Figure II.2.

With this formulation we may conclude that in general ARF translates a problem stated in REF into a heuristic search problem. The space in which this search problem is defined has the characteristic that each of its objects contains a Boolean constraint satisfaction problem. The
problem solver must have both heuristic search methods for conducting the search and constraint satisfaction methods for solving the problems associated with the objects in the space.

**Size of the Interpreter's Search Space**

The difficulty of the search problem for a given REF procedure depends on the amount of case analysis the interpreter must do during application of the operators. The amount of case analysis is determined by two factors: the number of branch statements interpreted which depend on the value of a `select` function call to determine the path taken, and the number of statements interpreted which contain expressions of the form '$\alpha[\beta]$' and '$\alpha$ of $\beta$' that depend on the value of a `select` function call for evaluation. For example, the `if` statement in the procedure defining the crypt-addition problem causes case analysis to occur because evaluation of the branching condition depends on the values of earlier `select` function calls. Also, in the example used in the discussion of set statement interpretation, it was the expression $B[<1>]$ which required the formation of new cases.

In most instances when ARF is given a Boolean constraint satisfaction problem stated in REF, no case analysis is needed and the interpreter's search tree has only one terminal node. This was the case for the magic square problem discussed above. For such problems there is essentially no search problem (for the interpreter) and the primary problem solving burden falls on the constraint satisfaction methods.

For any given process constraint satisfaction problem we may compare the size of the interpreter's space with the size of the space in which Floyd's backtracking algorithm searches. For the crypt-addition problem there are $10^8$ cases for the backtracking algorithm to consider while there
are only 16 cases in the subproblem space. This reduction in search allows the constraint satisfaction methods to do most of the work necessary to solve this problem.

We can make a similar comparison of search space sizes for a heuristic search problem stated in REF. In this case we can compare the size of the original problem's search space and the size of the ARF interpreter's search space. For a problem like the waterjug problem stated in Figure I.5, the interpreter must create six cases each time it interprets the computed goto statement L10. This case proliferation essentially nullifies the use of variables and causes the subproblem space to be the same size as the original problem's search space. However, for the statement of the monkey problem given in Figure I.6 the use of variables produces a significant reduction in search space size. Figure II.3a shows three levels of the search tree formed in the search space of the original problem. The nodes occur in this tree each time a set statement is encountered which contains a select function call, and the branches emanating from each node represent the possible values of the select function call. Figure II.3b shows three levels of the search tree formed in the subproblem space. The nodes occur in this tree each time a branch statement requires the creation of new cases, and the branches emanating from each node represent the cases created. The figures illustrate that the subproblem search tree is the smaller. The solution node is at level five for these trees and at that level there are 83 nodes in ARF's subproblem tree and 242 nodes in the original problem's tree.

SUMMARY

We have discussed in this chapter the problem of translating a REF problem statement into a form to which effective problem solving methods
can be applied. We began by listing some specific desirable traits of such a translation and then described the translation scheme contained in ARF. The translation which ARF performs allows the application of deductive constraint manipulation methods and heuristic search methods as mentioned in the list of desirable traits. It also provides freedom from the control sequence implied in the REF procedure by delaying the assignment of values to calls on the select function through the use of variables. We will see in the later discussions that although the use of variables provides the key which makes effective problem solving possible in ARF, the interpreter is still tied to the control sequence of the procedure and additional freeing devices are needed if the program's power is to be significantly increased.

There are two basic types of problems which ARF must solve: the constraint satisfaction problems contained in the context structures and the heuristic search problem of guiding the interpreter in the subproblem space. In Part III we will discuss the solving of constraint satisfaction problems and the methods applied to the context structures by ARF. In Part IV we will consider heuristic search methods applicable to the subproblem space and describe the methods employed by ARF.
III. SOLVING ARF'S CONSTRAINT SATISFACTION PROBLEMS

Consider again the form of the constraint satisfaction problems created during interpretation of a REF procedure. Each problem is represented by a context which contains a list of variables, a list of constraints, and a data structure. Associated with each variable is a list of integers which defines the range of values assignable to the variable. The constraints are in the form of Boolean expressions whose truth value can be determined if values are assigned to each variable. The data structure in a context contains the vector elements and attribute-value pairs that have been defined during interpretation. The values in this data structure serve the same function in the statement of the constraint satisfaction problem as do the constraints. That is, a value of 3 for the second element of the vector B is equivalent to the constraint 'B[2]=3'. Such constraints are represented in the data structure rather than as part of the constraint list in order to save processing time and storage space during both creation and solution of the problem.

DESIGNING A CONSTRAINT SATISFIER

We will proceed by presenting a progression of designs for methods to solve problems of this form and discuss the relative merits of each method. We will then show how these methods have been combined to form the framework for ARF's constraint satisfier.

Value Assignment Methods

First of all, observe that it is possible to enumerate all possible sets of values for the variables since each problem contains a finite number of variables each having a finite set of possible values. Hence, one could organize a problem solver to generate systematically each possible
set of variable values and test if the set constitutes a solution by evaluating each of the constraints. For example, consider the subproblem of the crypt-addition problem in which all the carry values are 0. The context which represents this subproblem is the one shown in Figure II.1. Let an 8-tuple represent a set of values for this problem's eight variables. Then a problem solver might consider all possible sets by generating them in lexicographic order as follows:

\[
\begin{align*}
(0,0,0,0,0,0,0,0) \\
(0,0,0,0,0,0,0,1) \\
(0,0,0,0,0,0,0,2) \\
\vdots \\
(9,9,9,9,9,9,9,8) \\
(9,9,9,9,9,9,9,9)
\end{align*}
\]

There are \(10^8\) such sets for this problem, and since the problem has no solution they must all be generated.

A much more efficient way of considering all possible solutions is to use a backtracking method. Backtracking achieves its effectiveness by requiring that the problem solver be able to test a partially generated solution. That is, for the problems we are considering instead of generating an entire set of values and then testing if the set is a solution, the backtracking method generates each value of a set separately and tests the partially defined set after each value is generated. The test can be implemented as a reevaluation of each of the constraints with the values in the partially generated set assigned to the appropriate variables. If a constraint reduces to 'FALSE' during the reevaluation, then the problem solver may discard the partial set and remove from consideration all possible completions of the discarded set.
Consider the use of this form of backtracking on the crypt-addition subproblem discussed above. Assume that the partial sets are being created by first generating a value for S(1), then a value for S(2), etc., and that the sets are being generated in lexicographic order as described above. As generation proceeds for the subproblem under consideration, the partial set (0,0,,,) will cause the constraint 'excl(S(1),S(2),S(3),S(4),S(5),S(6),S(7),S(8))' to become false. This partial set can be discarded and generation can proceed with (0,1,,). The same constraint disqualifies (0,1,0,,), (0,1,1,,), (0,1,2,0,,), etc. As generation proceeds the next constraint to disqualify partial sets is 'S(2)+S(5)=S(3)'. The first partial set containing five values which it and the excl constraint do not disqualify is (0,1,3,4,2,). The generation continues in this fashion until all partial sets have been discarded and the conclusion is reached that no solution exists for the subproblem. This backtrack search method reduces the number of sets which must be generated from \(10^6\) to approximately 58,000 \((\sim 6\times 10^4)\).*

Note that when a partial set having two elements is discarded, such as (9,0,,,), \(10^6\) complete sets are excluded from consideration as solutions; whereas when a partial set having five elements is discarded, such as (0,1,2,3,4,), only \(10^3\) complete sets are excluded. This observation motivates the design of schemes that will enable the problem solver to discard partial sets having as few elements as possible. One

* Floyd's backtracking algorithm discussed above in the section entitled "Size of the Interpreter's Search Space" does not achieve this reduction in the number of sets generated for the crypt-addition problem because it generates all eight elements of each set before testing whether the set is a solution.
strategy for such a scheme is to allow the problem solver to decide for each problem the order in which elements are assigned values in the sets being generated. That is, the problem solver described above always generates a value for S(1) first, then a value for S(2), etc.; but this need not be the case. For example, the problem solver might choose to begin by generating values for S(7) rather than for S(1). Since the subproblem contains the contradictory constraints $\neg(S(7)=0)$ and $S(7)=0$, it would be necessary to generate only ten partial sets to determine that the problem has no solution, i.e., \((\ldots 0,)\), \((\ldots, 1,)\), \(\ldots\), \((\ldots, 9,)\).

One way for the problem solver to determine a generation order is to examine the content and form of a problem's constraints. For example, ARF uses a method which orders the generation based on an attempt to make as many constraints reducible to 'TRUE' or 'FALSE' as soon in the generation as possible with preference given to those constraints which most restrict the variable values. This method decides which variable to assign a value next by selecting those constraints which have the least number of unvalued variables. These are the constraints which are reducible to 'TRUE' or 'FALSE' with the least number of additional generations. It then attempts to select constraints from this set which most restrict the values of the variables occurring in them. This judgment is based on the form of the constraints. That is, there is defined an operator ordering as follows: $\equiv$, $<$, $\text{excl}$, $\neg\text{excl}$, $\sim$, and $\vee$; constraints are ordered by their main operators using this ordering so that equations are assumed to be most restrictive and disjunctions least restrictive. The variables occurring in the most restrictive set of constraints are the candidates for selection. The algorithm chooses the most restricted candidate by determining the range size of each variable in the set and selecting one with the smallest range.
Figure III.1 shows the ordering determined by this algorithm for the crypt-addition subproblem we have been considering. The first set of constraints selected are those containing one variable, and the equation 'S(7)=0' is chosen as the most restrictive from that set. Hence, the first variable given a value in each solution set is S(7). The next set of constraints selected are those containing one variable other than S(7), and the constraint 'S(4)+S(7)<10' is chosen as the most restrictive from that set. Hence, the second variable given a value in each solution set is S(4). The algorithm continues as indicated in the figure until all eight variables are included in the ordering. The ordering defined in this way is used for the generation of each candidate solution set. For this example there is never more than one constraint in the set of most restrictive constraints and never more than one unvalued variable in the most restrictive constraint, so it is not necessary for the algorithm to consider the variables' range sizes.

As we observed earlier the choice of S(7) to be generated first is the most desirable for this problem. To see further the power of the ordering determined by the algorithm assume that the problem does not have the constraint '¬(S(7)=0)'. The same ordering would have been determined. During the generation, acceptable values for S(7) and S(4) could be found, but the zero value of S(7) would prevent any value for S(6) from satisfying both the excl constraint and the constraint 'S(4)+S(7)=S(6)'. Hence, only ninety-two partial sets would be generated, each having three or less elements.
Constraint Manipulation Methods

If we consider ways to further improve the power of this solution procedure for constraint satisfaction problems, we might observe that there is much information in the constraints that is not being utilized. For example, in the subproblem we have been considering, all ten possible values for $S(7)$ will be generated even though the constraint 'S(7)=0' clearly indicates what the value of $S(7)$ must be. Similarly, if values have been generated for $S(7)$ and $S(4)$, then the constraint 'S(4)+S(7)=S(6)' excludes all but one value for $S(6)$.

One may include in a problem solver routines which perform algebraic manipulations on constraints with the goal of reducing the amount of generation required during the backtracking search or, as in this case, eliminating the need for any search at all. These routines perform three principle functions: elimination of variables from constraints, elimination of elements from variable ranges, and deduction of inconsistencies among the constraints. Consider how such routines could perform these functions in the crypt-addition subproblem we have been using as an example.

Equality constraints such as 'S(7)=0' or 'S(1)+S(3)=S(8)' can be used to eliminate a variable by substituting all occurrences of the variable by an equivalent expression. The equivalent expression can be stored in the context structure as the value of the variable and all occurrences of the variable in vector elements, attribute values, values of other variables, and constraints can be replaced by the equivalent expression. Hence, in the example subproblem all occurrences of $S(7)$ would be replaced by 0 and all occurrences of $S(8)$ would be replaced by 'S(1)+S(3)'.
Constraints such as '$\neg (S(7)=0)$' and '$S(4)<9$' can be used to eliminate elements from the range of a variable. After a constraint has been used in this way it may be deleted from the context if it is true for all remaining range values. This is the case for both '$\neg (S(7)=0)$' and '$S(4)<9$'. If a constraint causes the range of a variable to be reduced to one element, then that element can be set as the value of the variable in the context.

Inconsistencies may be deduced from the constraints in several ways: a constraint may reduce to 'FALSE' following the elimination of a variable and reapplication of simplification routines, a constraint may cause removal of the entire range of a variable, or combinations of constraints such as '$S(7)=0$' and '$\neg (S(7)=0)$' may be inconsistent.

By appealing directly to the constraints in these ways the search for possible solutions may be reduced or eliminated. The effectiveness of these constraint manipulation methods is related to how and when they are applied, just as the effectiveness of the backtracking algorithm is related to the order in which the values are generated. A simple way of organizing the problem solver is to apply all the constraint manipulation methods to each constraint in the order in which they occur in the problem. In our example subproblem this would mean consideration of the excl constraint first, then '$\neg (S(7)=0)$', then '$\neg (S(4)=0)$', etc. In this problem the most obvious inconsistency occurs between the constraints '$S(7)=0$' and '$\neg (S(7)=0)$', but if the problem solver does not consider altering the order in which it processes constraints then this inconsistency will not be recognized until all twelve constraints have been processed.

This implies that the effectiveness of the problem solver can again be increased by designing an algorithm for selecting the order in which
constraints are processed. This ordering should be designed so that those constraints will be considered first for which there is a high probability that a variable can be eliminated, variable range elements can be deleted, or an inconsistency can be determined. The nature of the ordering algorithm should depend on the constraint manipulation methods which are available. A simple scheme which is suitable for use in ARF groups all constraints containing no variables or having occurrences of exactly one variable first, conjunctions next, equations next, '<' constraints next, excl constraints next, and all others following. The ordering within each of these groups (except the first) is based on the number of variables occurring in the constraint, those having the fewest variables being considered first.

For our example subproblem this ordering would dictate that constraints \(~(S(7)=0)\)', \(~(S(4)=0)\)' and 'S(7)=0' be considered first. Hence, the subproblem could be eliminated after consideration of at most three constraints with no backtracking required. To further illustrate the merit of this ordering assume that the problem does not contain the constraint \(~(S(7)=0)\)'. If the problem solver used 'S(7)=0' to replace all occurrences of S(7) in the constraints by 0, then the constraint 'S(4)+S(7)<10' would become 'S(4)<10', constraint 'S(4)+S(7)=S(6)' would become 'S(4)=S(6)' , and the excl constraint would become excl(S(1),S(2),S(3),S(4),S(5),S(6),0,S(8))'. The ordering would dictate that 'S(4)<10' be considered next since it has only one variable. The ordering would then select 'S(4)=S(6)' , since it is the equation which contains the fewest variables. If the problem solver uses this constraint to eliminate either S(4) or S(6), the excl constraint will have two identical operands and can therefore be determined to be false. Hence, the subproblem could be eliminated after consideration of five of the eleven constraints, again without resorting to backtracking.
Combining Constraint Manipulation and Value Assignment

Up to this point we have been considering a problem solver organized so that it would first apply constraint manipulation methods to the constraints of a problem and then afterwards, if necessary, would conduct a backtracking search to find a solution. One can further increase the problem solver's power by intermixing the constraint manipulation and value assignment methods. For example, when a variable is assigned a value during the backtracking, new constraints are produced by substituting the new variable value in the old constraints. The constraint manipulation methods may be able to use these new constraints to make deductions which will further reduce the generation required to complete the current partial set. Hence, they can profitably be applied to each new constraint as the search progresses.

If constraint manipulation continues during the backtracking in this manner, then the issue of what order the variables are to be assigned values by the backtracking routine can be raised anew. That is, the ordering routine may have dictated that \( S(i) \) be assigned a value first and \( S(j) \) be assigned a value second. But the assignment of a value to \( S(i) \) creates a new set of constraints and \( S(j) \) may no longer be the most desirable next variable for value assignment. This implies the ordering routine could be simplified so that it outputs only the next variable to be assigned a value, rather than an ordering of all the variables in the problem.

Hence, we can incorporate another degree of flexibility in the solving process; namely, after each value assignment during the backtracking search the constraint manipulation routines can be applied to the new constraints and then the decision made as to which variable is to be assigned a value next.
The Resulting Design

This completes our progression through the design of a problem solver for solving constraint satisfaction problems. The suggested design is not complete in that we have not discussed particular methods for constraint manipulation, expression evaluation and simplification, etc. What we have is a top level design or strategy for attacking the problems.

The flow chart in Figure III.2 indicates a problem solver which incorporates the design we have discussed. When a problem is input to the problem solver all of its constraints are on the unprocessed list; also, whenever a new constraint is produced during the application of the constraint manipulation methods or the assignment of a value to a variable, it is put on the unprocessed list.

The problem solver begins by processing each of the unprocessed constraints. The order in which the constraints are processed is determined by a selection algorithm which each time attempts to select the most constraining unprocessed constraint. If, when all the constraints have been processed, the problem is still not solved, then a step in the backtracking search is taken. Problem copies are kept in a stack so that backtracking can occur when necessary. At each step in the search the variable to be assigned a value is selected by an algorithm which attempts to determine the most constrained variable. Each time a value assignment is successfully made the new constraints produced during the assignment are processed as before and another step in the search is taken. This alternation between value assignment and constraint processing continues until a solution is found or until all possibilities have been considered.

This design which we have described is the one used in ARF's constraint satisfier. As the interpreter moves a context through a REF procedure it
creates constraints and places them on the unprocessed list. When a context is interpreted to end the constraint satisfier is called to find a solution and it proceeds as described in the flow chart of Figure III.2.

At any time during the interpretation of a context ARF's executive has the option of processing the context's constraints. In the current ARF this constraint processing is done whenever case analysis is necessary during interpretation. The goal of such processing is to derive a contradiction, when possible, so that the context can be eliminated from the interpreter's search tree before the new cases are formed. The issue of when constraints are to be processed involves trading off the advantages to the constraint satisfier of waiting until a context has been interpreted to end so that the constraints can be processed in an optimal order against the advantages to the heuristic search executive of processing each constraint when it is formed so that contexts containing contradictions can be pruned from the search tree as soon as possible.

ARF'S EXPRESSION MANIPULATION METHODS

In this section we will describe ARF's expression and constraint manipulation routines. These routines assume that an expression is represented internally as a list structure containing an operator and a list of operands. The operators currently defined are inte, symb, =, <, ~, +, -, of, elem, sele, excl, \&, and \lor. An inte expression represents an integer constant and has as its operand the cell which contains the integer. A symb expression represents a REF identifier and has as its operand the location of a structure which contains the name of the identifier. The operands of each of the other expression types are themselves expressions. An of expression has two operands, \( x \) and \( y \), and
indicates the value of the attribute x of identifier y. An elem expression has two operands, x and i, and indicates the value in the ith element of the vector x. A sele expression has two interpretations: when it appears in a REF statement it represents a call on the select function and has two operands; when it occurs in an expression in a context structure it represents an interpreter created variable and has a single integer operand which identifies the variable. ' + ' expressions may have any number of operands, while ' - ' expressions are restricted to one operand.

We may illustrate the internal form of an expression by using a prefix notation in which an expression is denoted by its operator followed by its list of operands separated by commas and enclosed in parentheses. Using this notation the internal form of the expression 'M[1] - B of C + 4 = 15' is as follows:

\[
= (+ (\text{elem}(\text{symb}(M), \text{inte}(1))), -(\text{of}(\text{symb}(B), \text{symb}(C))), \text{inte}(4)), \text{inte}(15))
\]

Expression Simplification

There is a single routine in ARF which is called whenever a REF expression is to be simplified or evaluated in a context. This routine acts as an executive for a set of other routines called actions which actually do the manipulations and substitutions during the simplification. These action routines are organized in lists indexed by expression operators; that is, there is a list of actions for equations, a list of actions for sums, a list for negations, etc. This action list organization facilitates easy modification and amplification of the simplifier as ARF and REF evolve. The simplification executive uses a simple "bottom up" algorithm which recursively simplifies each of the subexpressions of an expression before
evaluating the expression itself. Its flow of control is shown in Figure III.3. Each action routine exits + or - to indicate if simplification of the expression input to the action is complete. The executive responds to a - exit by ignoring the remaining actions on the list.

The action routines are designed to minimize the number of sub-expressions in an expression and to put each expression into a standard form. The definition of each standard form is given in Figure III.4. The use of standard forms simplifies the writing of action routines since each subexpression of the expression to which an action routine is being applied may be assumed to be in standard form. In addition, all the routines in ARF which process simplified expressions can assume that expressions are in standard form.

Some of the actions are of particular interest and will now be discussed. The restrictions on the standard forms for '－' and '～' expressions are satisfied by actions which carry out the following transformations:

\[
\begin{align*}
\sim(\sim(a)) & \rightarrow a & \neg(a) & \rightarrow a \\
\sim(a \lt b) & \rightarrow b \lt a+1 & \neg(a+b+\ldots+x) & \rightarrow \neg a \land b \land \ldots \land x \\
\sim(a \& b \& \ldots \& x) & \rightarrow \neg a \lor \neg b \lor \ldots \lor x & \neg(inte(j)) & \rightarrow inte(-j) \\
\sim(a \lor b \lor \ldots \lor x) & \rightarrow \sim(a) \land b \land \ldots \land x \\
\sim(symb(TRUE)) & \rightarrow symb(FALSE) \\
\sim(symb(FALSE)) & \rightarrow symb(TRUE) \\
\end{align*}
\]

Note that the restrictions on the '～' standard form free us from having to write constraint processing routines for constraints of the form

'\sim(a \lt b)', '\sim(a \& b \& \ldots \& x)', and '\sim(a \lor b \lor \ldots \lor x)'.

44
There are actions for expressions of the form 'S(i) = j', 'S(i) < j', and 'j < S(i)' which examine the range of the variable S(i) to determine if the expression will have the same value for all possible values of S(i). For example, if the first element in the range of S(i) is greater than the integer j then 'S(i) < j' can be replaced by 'FALSE' and 'j < S(i)' can be replaced by 'TRUE'.

Another of the simplification actions attempts to determine if an expression of the form 'a+b+c+...<j' (or 'j< a+b+c+...') always has the same truth value by attempting to evaluate the expression 'min(a)+min(b)+min(c)+...<j' (or 'j<max(a)+max(b)+max(c)'). If each of the addends a, b, c, ... are integers or have the form S(i) or -S(i), then the minimum and maximum values are known and the evaluation will produce 'TRUE' or 'FALSE'. If 'FALSE' is produced then the original expression 'a+b+c+...<j' (or 'j< a+b+c+...') can be replaced by 'FALSE'. If 'TRUE' is produced, then the expression 'j-1<max(a)+max(b)+max(c)+...' (or 'min(a)+min(b)+min(c)+...<j+1') is formed and evaluated; if that evaluation produces 'FALSE', then the original expression may be replaced by 'TRUE'.

The last action applied to '=' , '<', and 'v' expressions performs a generate-and-test procedure to determine if the expression has a constant value for all possible values of the variables occurring in it. The procedure generates each possible combination of values for the expression's variables and evaluates the expression assuming those values. If the evaluation produces the same result for each combination of variable values, then the original expression can be replaced by that result. For example, if the expression 'A[S(1)] + B[S(2)] < 4' is true for all possible value combinations of S(1) and S(2), then the expression can be replaced by 'TRUE'. This action is a very powerful tool for finding contradictions.
(by reducing constraints to 'FALSE') and for eliminating irrelevant con-
straints (by reducing them to 'TRUE'), but it can require large amounts
of processor time and must be used cautiously. We have responded to
this danger by including in the action a test which allows the action to
be applied only if the number of possible combinations of variable values
in the expression is 10 or less.

Enter a Value into a Context

We will now describe the routine which is called whenever a variable,
vector element, or identifier-attribute pair is to be assigned an expression
as a value in a context. (NOTE: the expression may be an inte or symb
expression and therefore denote a constant). This routine begins by ap-
propriately entering the value expression into the context. If a value
already exists for the vector element, variable, or identifier-attribute
pair, then the routine creates a new constraint which equates the old and
new values. If a value does not already exist, then the context is searched
to find and replace all references to the vector element, variable, or
identifier-attribute pair by the new value. Any new constraints produced
during this substitution procedure are placed on the context's unprocessed
constraints list.

In the case where a variable is being assigned a value, the new value
must be in the variable's range. If the value being assigned is an integer
then a test is made to determine if the integer occurs in the range. In
the case that the value is not an integer, new constraints are formed as
follows: If the range of the variable is i1,i2,...,ik and the value ex-
pression is x, then the constraints 'i1-1<x' and 'x<i1+k' are formed. Also,
if necessary, the constraint 'excl(x,n1,n2,...,nm)' is formed where the nj
are all the integers not in the variable's range such that $1 \leq j < k$.

These constraints are sufficient to assure that $x$ denotes a value in the
variable's range.

**Constraint Manipulation Methods**

Whenever a constraint is to be processed, an executive routine is
called which executes a series of constraint manipulation action routines
to perform the processing. These action routines attempt to determine
values for vector elements, attribute-identifier pairs, and variables;
delete elements from the ranges of variables; and find contradictions.
The action routines are organized in lists indexed by expression operators
just as the expression simplification actions are. Hence, there is the
same flexibility for adding, deleting, and changing actions in the con-
straint processor as in the expression simplifier. The flow of control
in the constraint processing executive is shown in Figure III.5. Each
action exits indicating CONTINUE, STOP.OK, or STOP.FAIL. A CONTINUE exit
indicates continuation to the new action. A STOP.OK exit indicates that
the new constraint has been eliminated and that no more actions are to be executed. A STOP.FAIL exit indicates that a contradiction was found and
that the executive should exit indicating failure. Any new constraints
created by the actions are entered into the context as unprocessed con-
straints by the executive.

Figure III.6 identifies, describes briefly, and indicates the results
of each constraint manipulation action routine in the current ARF; the
figure also gives the action lists for each constraint main operator. In
the remainder of this section we will expand on the figure's action
routine descriptions and provide an example of ARF's constraint processing behavior.
Actions 1 through 5 test if the constraint being processed is in a particular form. If it is, then the action makes all the relevant deductions possible from the constraint and eliminates it.

Action 6 is applied during the processing of a '=' constraint and it attempts to use the new constraint to determine a value for a vector element, identifier-attribute pair, or variable. If such a value can be determined, then the value assignment routine is called to establish the value in the context. For example, the action will use the constraint 'S(1)+S(3)=S(8)' to assign the value 'S(1)+S(3)' to S(8) and thereby eliminate the variable S(8) from all constraints and value expressions in the context.

The action proceeds by searching the constraint for a subexpression x which satisfies each of the following conditions:

1. The constraint is in one of the following forms:
   \[ x=y, \ y=x, \ -x=y, \ y=-x, \ldots+x+,\ldots=y, \ y=\ldots+x+\ldots, \ldots+x+\ldots=y, \text{ or } y=\ldots+x+\ldots \] where y is any expression.

2. The expression x is in one of the following forms:
   \[ S(i), \ y[j] \] where y is a symb expression and j is an inte expression, or 'y of z' where y and z are symb expressions.

3. The expression x occurs only once in the constraint.

4. If x is of the form S(i), then the constraint contains a subexpression other than x of the form S(j).

If an appropriate subexpression x is found, then the constraint is manipulated so that it has the form x=z. The z expression is then entered in the context as the value of the vector element, attribute-identifier
pair, or variable defined by x. The purpose of condition 3 is to prevent the value expression z from containing x. For example, this condition would prevent constraint 'S(1)=A[S(1)]+1' from being used to determine 'A[S(1)]+1' as a value for S(1). Condition 4 insures that variable values are expressed in terms of other variables rather than vector elements or attribute values. For example, the action will use the constraint 'A[1]=S(2)' to set a value for A[1] rather than for S(2).

Action 7 attempts to eliminate elements from the ranges of those variables which appear in the constraint being processed by generating each possible combination of values for those variables and testing if the new constraint and any other processed constraint which contains only those variables are true for each combination. The rule used to eliminate range elements is the following: if S(i) occurs in the new constraint, j is an element in the range of S(i), and there is no combination of variable values that contains j as the value for S(i) for which the constraints are all true, then j can be eliminated from the range of S(i). The generate-and-test procedure is applied only if the number of possible value combinations for the variables is not greater than 20. This restriction prevents the action from requiring an excessive amount of processor time by generating through large spaces of value sets.

In addition to deleting range elements, this action will eliminate the new constraint when it contains only one variable and the generate-and-test procedure is carried out. In that case the constraint is true for each remaining possible value for the variable and can therefore have no further effect on the context. The following example illustrates the use of this action. If S(1) and S(2) each have 0, 1, 2 in their ranges, the context contains the processed constraints '0<S(1)+S(2)' and
'S(1)+S(2)<3', and the new constraint '-1<S(2)-S(1)' is being processed, then this action will generate combinations of values for S(1) and S(2) and thereby determine that 2 can be deleted from the range of S(1) and that 0 can be deleted from the range of S(2).

Action 8 attempts to make deductions when an excl constraint is being processed which has inte or symb expressions (i.e., constants) as operands. Assume the constraint has the form 'excl(...,c1,c2,...,cn,...)' where c1,c2,...,cn are inte or symb expressions. If the constraint has an operand of the form S(i), then each cj which is an inte expression is deleted from the range of S(i) by a call on the range value deletion routine; if such a deletion produces a contradiction, then the action exits STOP.FAIL. For each operand 'x' of the constraint which is not a constant or a variable, the constraint 'excl(c1,c2,...,cn,x)' is formed and input to action 7, the generate-and-test action; if execution of this action produces a contradiction, then action 8 exits STOP.FAIL. If the generate-and-test action were called with the original excl constraint, it could be effective only if the number of possible combinations of values for the variables in the constraint's operands was not greater than twenty. By forming excl constraints which have only one nonconstant operand, the generate-and-test action can be effective if the number of possible combinations of values for the variables in that single nonconstant operand is not greater than twenty. After each nonconstant operand is considered in these ways (i.e., either as 'S(i)' or as 'x'), the original excl constraint is reevaluated to determine if it reduces to 'TRUE' of 'FALSE'. Exit from the action is CONTINUE if the constraint does not reduce to 'TRUE' or 'FALSE' and either STOP.OK or STOP.FAIL if it does.
Action 9 is applied during the processing of a '<' constraint and it attempts to form additional constraints by replacing occurrences of $S(i)$ or $-S(i)$ in the new constraint by their maximum or minimum values. If the new constraint is of the form '$a+b+c+d+...<j$' (or '$j<a+b+c+d+...$'), then the following series of additional constraints is formed:

'$a+\min(b)+\min(c)+\min(d)+...<j$' (or '$j<\max(a)+\max(b)+\max(c)+\max(d)+...$'),

'$\min(a)+b+\min(c)+\min(d)+...<j$' (or '$j<\max(a)+\max(b)+\max(c)+\max(d)+...$'),

'$\min(a)+\min(b)+c+\min(d)+...<j$' (or '$j<\max(a)+\max(b)+\max(c)+\max(d)+...$'), ...

If $x$ is of the form $S(i)$ or $-S(i)$, then '$\min(x)' (or '$\max(x)$') is determined from the range of $S(i)$; otherwise, '$\min(x)' (or '$\max(x)$') is assumed to be $x$ itself. Consider the following example of this action's behavior. If the constraint '$S(1)+S(2)<5$' is being processed in a context where $S(1)$ has a range $1,2,3,...,10$ and $S(2)$ has a range $0,1,2,...,9$, then this action will produce the constraints '$S(1)<5$' and '$S(2)<4$'.

If none of the addends $a,b,c,d,...$ in the new constraint are of the form $S(i)$ or $-S(i)$, then the action is unable to produce any additional constraints which differ from the original new constraint. In that case the action attempts to achieve results by transferring control to action 7, the generate-and-test action.

Actions 10 through 16 compare the constraint being processed with each constraint on the processed constraints list in an attempt to make deductions. For action 10, if '$x_1=c_1$' is the constraint being processed, then the processed constraints list is searched for occurrences of $x_1$ to be replaced by $c_1$; in any case, if a constraint of the form '$x_2=c_2$' is found on the processed constraints list, then all occurrences of $x_2$ in the new constraint will be replaced by $c_2$. For actions 11 through 16,
their descriptions in Figure III.6 are given in terms of constraint pairs formed by the new constraint and a constraint on the processed constraints list.

Deleting an Element from the Range of a Variable

There is a routine which is called whenever it has been determined that an element can be deleted from the range of a variable. This routine deletes the element and then attempts to make deductions implied by the deletion. It first tests if there is only one element remaining in the variable's range. When that is the case, the value assignment routine is called to set the remaining range element as the value of the variable; the routine then exits + or - to indicate the success or failure of the assignment.

When the variable's range has more than one remaining element, a test is made to determine if the deleted element was the first or last element in the range. If it was, then the minimum or maximum value of the variable has been changed and action 9 for processing '<' constraints (described in the previous section) may be able to create new constraints. For example, if the routine has deleted the first element from the range of S(i), then 'min(S(i))' has a new value and the action routine can be productively reapplied to processed constraints such as '...+S(i)+...<j'.

A 'min' list and a 'max' list of variable names are attached to a '<' constraint when it is processed. A variable name is on the 'min' (or 'max') if the minimum (or maximum) value of that variable was used to create a new constraint by the action 9 routine. Hence, when the range element deletion routine determines that it has deleted the first (or last) element from a variable's range, it searches the processed constraints list for a '<' constraint whose 'min' (or 'max') list contains the variable. Action 9
is reapplied to each constraint found during this search and any new constraints formed by the action are placed on the context's unprocessed constraints list.

An Example of ARP's Expression Manipulation

To illustrate the behavior of the routines described in the preceding sections, consider again the crypt-addition subproblem in which the carry value is always zero (see Figure II.1). As interpretation begins, the first constraints are formed by the condition statements at lines 8-10 of the REF procedure (see Figure I.3b). The constraints are simplified and placed on the unprocessed constraints list as 'excl(S(1), S(2), S(3), S(4), S(5), S(6), S(7), S(8))' and '~(S(7)=0)\&\&(S(4)=0)'. The first time the L2 loop is interpreted the branching condition at line 14 is simplified as follows:

\[
\langle A_1[<I>] \rangle + \langle A_2[<I>] \rangle + <CARRY> < 10
\]

\[
S(1) + S(3) + 0 < 10
\]

\[
0 + S(1) + S(3) < 10
\]

\[
S(1) + S(3) < 10
\]

Since the truth value of the branching condition cannot be determined, the interpreter must do case analysis. Before creating the two cases, the interpreter attempts to eliminate the context by processing its unprocessed constraints.

Constraint '\(\sim(S(7)=0)\&\sim(S(4)=0)\)' is selected to be processed first. Action routine 5 erases the constraint being processed and creates additional constraints '\(\sim(S(7)=0)\)' and '\(\sim(S(4)=0)\)'. These new constraints are placed on the unprocessed list. Constraint '\(\sim(S(7)=0)\)' is selected to be processed next. Action routine 4 calls the range element deletion routine to delete 0 from the range of S(7) and erases the constraint being processed.
Since 0 is the first element in the range of S(7), the range element
deletion routine searches the processed constraint list for a '<' con-
straint with S(7) on its 'min' list; no '<' constraints are found.
Constraint '~(S(4)=0)' is selected next and the processing erases it and
deletes 0 from the range of S(4) in a similar manner. Constraint
'\texttt{excl}(S(1),S(2),S(3),S(4),S(5),S(6),S(7),S(8))' is the remaining un-
processed constraint and the only effect of its processing is to move
it to the processed constraints list.

Since no contradiction was found during the constraint processing,
the interpreter creates two cases (i.e., contexts). For the case we
are considering the branching condition is assumed to be true. Hence,
processing of the constraint 'S(1)+S(3)<10' is initiated. Action routine
9 creates two new constraints 'S(1)<10' and 'S(3)<10'. Both of these
new constraints reduce to 'TRUE' during simplification because all of
the range elements for S(1) and S(3) are less than 10; the two 'TRUE'
constraints are eliminated when they are processed. The constraint
'S(1)+S(3)<10' is put on the processed constraints list and a 'min' list
is formed for it containing S(1) and S(3).

Interpretation continues for our case at the condition statement L3.
The statement's Boolean expression is evaluated as follows:

$$\langle A1[<i>]\rangle + \langle A2[<i>]\rangle + \langle CARRY\rangle = \langle SUM[<i>]\rangle$$

$$S(1) + S(3) + 0 = S(8)$$

$$0 + S(1) + S(3) = S(8)$$

$$S(1) + S(3) = S(8)$$

$$-S(8) + S(1) + S(3) = 0$$

The evaluated expression is put on the unprocessed constraints list.
Interpretation continues to statement L2 and back through the loop a second time. At the line 14 if statement, case analysis is again necessary and the interpreter processes the context's unprocessed constraints. Action routine 6 erases constraint '-S(8) + S(1) + S(3) = 0' after determining the value 'S(1) + S(3)' for S(8). The value assignment routine enters this value for S(8) and creates the additional constraints '-1 < S(1) + S(3)' and 'S(1) + S(3) < 10'. During the simplification of constraint '-1 < S(1) + S(3)', the expression '-1 < max(S(1)) + max(S(3))' is formed and simplified to 'TRUE'; this result causes the expression 'min(S(1)) + min(S(3)) < -1 + 1' to be formed and simplified to 'FALSE'; this result allows the conclusion that the original constraint, '-1 < S(1) + S(3)', simplifies to 'TRUE' and can be eliminated. Constraint 'S(1) + S(3) < 10' remains unchanged during simplification and is put on the unprocessed constraints list. The value assignment routine also replaces S(8) by 'S(1) + S(3)' in the excl constraint and moves that constraint to the unprocessed constraints list. When constraint 'S(1) + S(3) < 10' is selected for processing it is found to already exist on the processed constraints list and no processing occurs. The only effect of processing the new excl constraint is to move it to the processed constraints list. We conclude our example by showing in Figure III.6 the context as it exists at this point in the interpretation.

Appendix I provides a second example of ARF's expression manipulation capabilities. ARF's output for the magic square problem (see Figure I.2) is presented in the appendix and the semantic trace in the output shows the role played by the expression manipulation routines in solving the problem.
CONCLUSIONS

ARF achieves most of its problem solving power from its ability to efficiently solve the constraint satisfaction problems defined by the interpreter's context structures. The most powerful methods available to the program for solving these constraint satisfaction problems are the constraint processing action routines; hence, these routines form the heart of ARF's problem solving power. This power is derived from the fact that the deductions made by these routines make many-fold reductions in the space of possible solutions for a context and thereby save the program from having to conduct large expensive searches to find solutions.

No claims are made concerning the completeness of the set of simplification action routines and constraint manipulation action routines described above. Indeed, this set of routines has continually changed and grown during ARF's lifetime. What we can claim is that our experience has shown this to be a "good" set of action routines in that it allows the program to achieve a relatively consistent level of deductive ability in the most frequently occurring problem situations. The decision as to whether any given routine should be included as an action is a heuristic one based on judgments concerning the processing time required to determine if the action is applicable, the frequency of the action's applicability (i.e., its generality), the processing time required to apply the action, and the expected reduction in processing time achieved by an application of the action.

The backtrack search strategy used to find acceptable variable values once a context has been interpreted to end has proved to be adequate for most of the problems submitted to ARF, but it is clear that
a more flexible search strategy will be needed for larger problems (see the discussion in part V below of the packing problem). Such a strategy might have available the following operator:

If j is an element in the range of S(i) in context C, then form a new context C' by copying context C and assigning j as the value of S(i) in context C'; also, remove j from the range of S(i) in context C.

This operator would allow the search executive at each step in the search to assign any range value to any variable in any context.

One might also include operators which do case analysis rather than value assignment during the search; for example, consider the following operator:

If i1, ..., ij, ik, ..., im is the range of S(n) in context C, then form context C' by copying context C and adding the constraint 'S(n) < ik' to the copy; also, add the constraint 'ij < S(n)' to context C.

Burstall [2] has used an operator of this form to bisect the remaining solution space at each step. Such a strategy relies heavily on the ability of the constraint processing methods to eliminate cases by deducing contradictions.
IV. GUIDING THE INTERPRETER'S SEARCH

We observed earlier in Part II that ARF is organized to treat a REF procedure as a heuristic search problem in a space whose objects are the context structures used by the ARF interpreter and whose operators are the statements in the REF procedure. The search begins by applying the operator associated with the begin statement to an empty context and ends successfully when the operator associated with the end statement can be applied to produce an object with no unsatisfied constraints. In this chapter we will consider the task of designing a search strategy for problems in this space. Our discussion will include descriptions of ARF's search methods and proposals for extensions of those methods.

ARF'S SEARCH STRATEGY

Since there is never more than one operator applicable to an object at a given time in this space, the only task of a search strategy is to select at each step the next object to which an operator will be applied. Figure IV.1 shows the basic structure of ARF's executive routine for conducting this search. It maintains a list of current objects (i.e., a list of those objects at the terminal nodes of the current search tree) from which an evaluation routine selects the next object to be interpreted. Each interpretation step produces either a solution or a list of objects (perhaps empty) to be added to the current objects list. Interpretation steps continue in this manner until a solution is found or until there are no current objects to be interpreted.
The important element in this executive's search strategy is the evaluation routine which selects an object at each step. ARF uses three standard heuristic search techniques in its evaluation routine: recognition and deletion of contexts which are equivalent to contexts produced earlier in the search, elimination of contexts which cannot lead to a solution (by the derivation of contradictions), and selection for continued interpretation at each step in the search the context which is "closest" to being a goal context at end. In this section we will describe these three methods and show how they affect ARF's behavior.

Recognition of Equivalent Contexts

One important way of improving a problem solver's ability to search through a space is to give it the capability of answering the question "Have I considered this point in the space before?". For ARF to have this capability it must do more than just keep a record of where it has been, because two contexts need not be identical to represent equivalent states (in fact, identical contexts are rarely produced). For example, consider the two subproblems of the monkey problem shown in Figure IV.2. These non-identical subproblems represent equivalent states in the sense that the box is in the same position and the possible positions of the monkey are the same in both subproblems. One would like ARF to recognize this equivalence and delete context 1.1.1.2.

Assuming that ARF saves a copy of each context produced during interpretation, consider what are the necessary and sufficient conditions for a new context produced by the interpreter to be eliminated as redundant.
To state these conditions define an admissible value assignment for a context to be an assignment of a value to each variable of the context such that each variable's value is an element of its range and the assignment satisfies the context's constraints. Then, a new context X produced by the interpreter is redundant with respect to the contexts previously produced by the interpreter if and only if for every admissible value assignment for context X there exists a previously produced context Y and an admissible value assignment for context Y such that the assignments make the data structures of contexts X and Y identical. (Note: A context's next applicable operator is considered part of its data structure).

The use of a general test for determining when a new context is redundant would in most cases require more processor time to apply than it would yield in search reduction. An alternative course of action is to use a test which looks only for a set of sufficient conditions for eliminating a context. Hopefully such a test would detect obvious looping situations and would be inexpensive to apply.

We have taken this alternative in ARF. When a new context is produced it is successively compared with each previously produced context. For each pairwise comparison, success of the test indicates that the new context is redundant and can be eliminated; failure of the test indicates that redundancy cannot be determined. If either of the contexts has constraints or non-integer variable values, then the test fails. The test succeeds only if a mapping F can be found which satisfies the following conditions:
1. \( F \) maps the variables of the old context into the variables of the new context (\( F \) need not be one-to-one).

2. If \( S(i) \) is a variable in the old context and \( j \) is in the range of \( F(S(i)) \), then \( j \) is also in the range of \( S(i) \).

3. If every occurrence of each variable \( S(i) \) in the data structure of the old context is replaced by \( F(S(i)) \), then every value expression in the data structure of the old context is identical to the corresponding value expression in the data structure of the new context.

The existence of such an \( F \) assures that for all admissible value assignments for the new context there exists an admissible value assignment for the old context such that the assignments will make the data structures of the two contexts identical.

This test is weak in that it excludes from consideration contexts having constraints, but it is sufficient to recognize the looping in the monkey problem and the waterjug problem. It is often the case that for problems which require the production of many subproblems the constraints in each subproblem are simple enough to be eliminated during processing and the looping test can be effectively applied.

There remains the issue of which and how many contexts to save during interpretation. That is, there are space bounds on the number of contexts the program can store and there are processing time considerations concerned with application of the looping test. We have dealt with these issues by first observing that in most cases two contexts first become
equivalent at a labeled REF statement after having reached that statement along different control paths. For example, in our REF statement of the monkey problem all of the context equivalences first occur at the end of the control paths which converge at statements WALK and LL. Based on this observation we save and test contexts in ARF only at those REF statements which are labeled and can be reached along more than one control path. This policy reduces the total number of contexts which must be saved and the number of times the looping test is applied.

We further restrict the amount of memory required for loop testing by limiting the total number of saved contexts to twenty. When interpretation produces more than twenty contexts to be saved, the save list is evenly "thinned out" as follows: production of the \( i \)th context to be saved \((i > 20)\) causes the \(((i \mod 20)+1)\)st element of the save list to be deleted and the context being saved to be entered as the new 20th element. This deletion algorithm attempts to preserve completeness by keeping contexts from all phases of the interpretation on the list; for example, after thirty-nine contexts have been produced to be saved, the list has the following contexts on it: 1st, 3rd, 5th, ..., 37th, 39th.

**Deletion of Contexts Containing Contradictions**

As we observed earlier in Part III, solving of the constraint satisfaction problem within a context is more efficient if constraint processing is delayed until the context is interpreted to end; on the other hand solving of the search problem in the interpreter's search space is more efficient if the constraint processing methods are used to recognize contradictory constraints as soon as they are created. A constraint processing strategy is desired which balances these conflicting factors. The strategy adopted for ARF is to process a context's constraints during
interpretation only when that context is to be interpreted through an if, set, or computed goto statement which will require case analysis. Hence, no attempt is made to derive a contradiction in a context until it is known that interpretation of that context is going to create new branches in the subproblem search tree. This strategy allows constraint processing to be delayed until the end statement is reached for problems such as the magic square problem, but allows contradictory contexts to be eliminated from the subproblem search tree for problems such as the SEND+MORE=MONEY crypt-addition problem.

If a more flexible constraint processing strategy were desired, the processing of a constraint could be considered to be an operator which the search executive could apply as a step in its search. The executive could then consider constraint processing at each search step and select each constraint to be processed. Although this mechanism would increase the decision making burden of the search executive, it would give the executive control over more of the program's activities and therefore provide the possibility for more sophisticated search strategies.

Selecting the "Closest" Object to the Goal

The two search methods we have discussed above serve to eliminate objects from the search tree. We will now describe how ARF's evaluation routine selects an object for further interpretation from among those which have not been eliminated. To form the basis for our selection routine we have defined a distance measure in the search space so that the routine can select the object which is the least distance from the desired final
object. The strategy is to select the object nearest the goal and move it through the space until it either reaches the goal or moves away from the goal far enough so that some other object is nearer. The desirability of this search strategy depends on the accuracy of the distance measure and the amount of processing required to compute the distance between an object and the goal. The ideal measure is both accurate and easy to compute.

The measure in ARF is derived from the control structure of the REF procedure being interpreted. Prior to interpretation, a structural analysis of the procedure is made to determine the shortest control path from each statement in the procedure to \texttt{end} and the number of statements on each of those paths. This statement count along the shortest path to \texttt{end} is attached to each statement in the procedure and is used as the distance measure for objects in the subproblem space. That is, each context has associated with it the next statement through which it is to be interpreted; the shortest path statement count for that statement is taken as the measure of how near a context is to reaching \texttt{end} and thereby becoming a goal object.

This distance measure has the desirable attribute of being easy to compute, since the analysis of the procedure's control structure does not require a large amount of processing time and is done only once. The measure's accuracy is less than ideal in that it is based on necessary but not sufficient conditions for reaching \texttt{end}; that is, it indicates the minimum number of statement interpretations necessary to move a context to \texttt{end}, but does not provide any indication of whether that minimum can be achieved by the context. Nevertheless, the measure is useful and is sufficient to give ARF a sense of direction in its search.
Consider the use of this measure for the REF statement of the monkey problem given earlier in Figure 1.6. Figure IV.3 shows the control structure for this procedure and the parenthesized numbers in the figure are the distance measures associated with each statement. Note that standard goto statements are not assigned a measure nor are they considered in the statement count for a path.

As interpretation proceeds for this procedure the first case analysis occurs at the if statement following statement WALK. The executive must decide which of the two contexts produced to continue interpreting. The distance measures indicate that the context at statement WALK is 6 statements from end and the context at statement L1 is 4 statements from end. Hence, the context at L1 is selected for continued interpretation. Interpretation through statement L1 causes three contexts to be produced: one at WALK 6 statements from end, one at MOVE, BOX 6 statements from end, and one at CLIMB 3 statements from end. The context at statement CLIMB is selected and interpretation continues to the if statement at line 10 of the procedure. Since the monkey is not under the bananas, the interpreter must take the branch to STEP, DOWN and continue back through L1. The three contexts produced by the interpretation through L1 are determined to be equivalent to the three contexts produced the first time L1 was interpreted and they are erased.

The interpreter now has three contexts to choose from for further interpretation: a context at WALK coming from the line 7 if statement and containing one variable, a context at WALK coming from L1 and containing two variables, and a context at MOVE, BOX coming from L1 and containing two variables. Since each of these contexts is six statements from end, the executive must either use some other criterion for selecting one of them
or make an arbitrary selection. An additional criterion is used, and it is that the context with the fewest variables is selected. This criterion causes ARF to seek the simplest or shortest solution to a problem and in this case it causes selection of the context at WALK coming from the line 7 if statement. The interpreter moves this context through statement WALK and through the line 7 if statement. The two contexts produced at the if statement are determined to be equivalent to previous contexts and are erased. The executive now has two contexts to select from both six statements from end and both having two variables. It arbitrarily selects the context at WALK and after interpretation through WALK and the line 7 if statement the resulting contexts are found to be equivalent and erased.

The remaining context is now selected and interpreted through the two statements at MOVE, BOX and through L1. Of the three contexts produced at L1, the one branching to CLIMB is the least distance from end and is selected. Interpretation of statement CLIMB and the line 10 if statement produces two contexts: one at STEP, DOWN and one at the line 11 computed goto statement. The context at the computed goto statement is selected since it is only one statement from end. Interpretation of the computed goto statement produces two contexts and the executive's selection of the one at end causes termination of the search with a solution.

Figure IV. 4 shows the search tree produced during the interpretation we have described for the monkey problem. The search strategy recognizes that the monkey must climb to get the bananas and directs interpretation along the CLIMB path whenever a context is available to do so.
For the REF statement of the SEND+MORE=MONEY crypt-addition problem we introduced in Part I (see Figure I.3), ARF's search strategy results in a modified form of depth-first search during interpretation. Figure IV.5 shows the control structure and distance measures for this procedure. The first selection in the search is made between context 1.1 and context 1.2 after the case analysis at the line 13 if statement in the first pass through the L2 loop. Both of these contexts produced by that case analysis are distance 5 from end and have 8 variables; hence, an arbitrary selection is made of the context on the path of L3 (context 1.1). Interpretation continues with context 1.1 until the L2 loop is entered the second time at the line 12 set statement. The distance of context 1.1 from end at that statement is 7. Since context 1.2 is at distance 5, it is selected for interpretation and its interpretation continues until both contexts are at the line 12 set statement.

The context selection algorithm is organized so that if two or more contexts are at the same distance from end and have the same number of variables, then the most recently interpreted context is chosen. This convention gives the search a depth-first nature for a problem such as SEND+MORE=MONEY since interpretation will continue with the same context as long as it is at least as close to end as any other context and no new variables are being created. Interpretation proceeds with context 1.2 through the line 13 if statement where contexts 1.2.1 and 1.2.2 are produced at distance 5 from end. Context selection and interpretation continues with these two contexts just as it did for contexts 1.1 and 1.2 at the first case analysis; that is, context 1.2.1 is interpreted to the line 12 set statement and then context 1.2.2 is interpreted back to the line 13 if statement where contexts 1.2.2.1 and 1.2.2.2 are formed. Figure IV.6 shows the complete search tree for this problem and indicates graphically the depth-first nature of the search.
For the waterjug problem (see Figure I.5), ARF's search strategy produces essentially breadth-first search. Figure IV.7 shows the control structure and distance measures for this procedure. The breadth-first behavior occurs because of the rule which selects from among the contexts closest to end the context with the least number of variables. Since each interpretation of statement L10 produces a new variable, this selection criterion tends to cause each of the six contexts produced at statement L10 to be interpreted through statement L7 before any one of them is interpreted through L10 again. ARF is able to find a solution because most of the contexts in the search tree are determined to be redundant and are eliminated. Figure IV.8 shows the search tree produced in finding the solution.

We may conclude from the crypt-addition and waterjug examples, that, in general, ARF conducts primarily a breadth-first search when interpretation is continually producing new variables and that it conducts primarily a depth-first search when no new variables are being created during interpretation.
A PROPOSAL FOR EXTENDING ARF'S SEARCH STRATEGY

Although the number of statements to end distance measure provides ARF with some sense of direction in its search, stronger heuristics are needed for many problems. For example, ARF is able to solve the water-jug problem primarily because the search tree is quite small after redundant contexts are removed, rather than because of any insights into how the search should proceed. We wish to indicate in this section a collection of ideas for giving ARF a means-ends analysis capability which could provide the desired increase in search expertise.

We begin by observing that a context can be interpreted either forward or backward along a control path. That is, we can construct an inverse interpreter which moves contexts through a procedure from end toward begin rather than from begin toward end. For example, inverse interpretation of a set statement means starting with a context in which the assignment indicated by the set statement is assumed to have been made and modifying the context so that it is in a state equivalent to the state it was in before the assignment was made. An inverse interpreter would use contexts, variables, constraints, and case analysis in a manner similar to the standard interpreter.

Consider the inverse interpretation of each form of REF statement. Inverse interpretation of an if or computed goto statement involves adding the constraint to the context which must have been true for the standard interpreter to have taken the path on which the inverse interpreter is approaching the statement. For example, if in the REF statement of the waterjug problem introduced in Part I (see Figure 1.5), the inverse interpreter
moves a context from statement L9 through the if statement at L5, then it will add to the context the constraint '8 < <A> + <B>'.

When the inverse interpreter completes interpretation of a labeled statement which can follow more than one statement during forward interpretation (i.e., a join point), it saves a copy of the context and considers each path which the forward interpreter could have taken to reach the labeled statement as a separate case. This case analysis creates a subproblem space to be searched as it does during forward interpretation. Note that the inverse interpreter need not add a branching condition to each new subproblem as does the standard interpreter.

When the inverse interpreter encounters a set statement it can deduce that all references in the context to the vector element or attribute-identifier pair indicated by the statement's left side can be replaced by the statement's right side expression. In addition, if the left side vector element or attribute-identifier pair has a value in the context, then a new constraint can be formed equating that value to the right side expression and the value can be removed from the context. For example, inverse interpretation of the statement 'set <B> to <A> + <B>' with a context containing the constraint '¬(<B>=2)' would cause the constraint to be changed to '¬(<A>+<B>=2)'. Inverse interpretation of a set statement can cause case analysis to occur just as it does during forward interpretation.

One could use an inverse interpreter in any of several ways. For example, it could replace the standard interpreter so that all interpretation was from end toward begin. This would be equivalent to a problem solver (or theorem prover) which searched backward from the goal (or theorem) toward the initial object (or axiom set). The inverse interpreter could also be used in conjunction with the standard interpreter to conduct a
two-way search; the standard interpreter would grow a search tree from \texttt{begin}, the inverse interpreter would grow a search tree from \texttt{end}, and the search executive would attempt to guide the interpreters so that the two trees joined.

The most productive use for the inverse interpreter seems to be as a tool in the application of means-ends analysis to guide the standard interpreter. This analysis might take the following form. When a context is selected by the executive for interpretation, its interpretation would continue until it was eliminated by a contradiction, reached the \texttt{end} statement, or was forced to take a branch at an \texttt{if} or computed \texttt{goto} statement such that the context would have to return to that \texttt{if} or computed \texttt{goto} statement before it could reach \texttt{end}. Examples of the undesirable branch at an \texttt{if} or computed \texttt{goto} statement would be the branch to \texttt{L10} at the \texttt{L7 if} statement in the waterjug procedure and the branch to \texttt{WALK} at the \texttt{L1} computed \texttt{goto} statement in the monkey procedure. When such an undesirable branch occurs we wish to interpret the context through statements which will alter the context so that when it returns to the \texttt{if} or computed \texttt{goto} statement a branch leading toward \texttt{end} can be taken. Once such a statement(s) is located, the search executive can define a subgoal to interpret the context through that statement(s). In the attempt to achieve this subgoal the shortest path distance measure can be computed and used to guide the search. When the subgoal is achieved, then the context can be interpreted past the troublesome \texttt{if} or computed \texttt{goto} statement toward \texttt{end}. Each such blockade along the path to \texttt{end} can be treated in the same way.

The inverse interpreter is used in this process to locate the statement(s) which becomes the subgoal as follows. An empty context is formed
and placed on a branch of the if or computed goto statement which leads
toward end. The inverse interpreter is called to move this context
through the if or computed goto statement. This interpretation adds a
constraint to the context which if true would allow a context to pass
through the if or computed goto statement on the path toward end. Inverse
interpretation is continued until a set statement is encountered which
alters a vector element or attribute value occurring in that constraint.
This is the statement which becomes the subgoal for the standard interpreter.

The context produced by the inverse interpreter after passing through
the set statement defines a set of conditions on a context produced by the
standard interpreter at the set statement; that is, if the context produced
by the standard interpreter is to be able to take a branch toward end at
the if or computed goto statement, then each constraint in the inverse
interpreter's context must be true in the standard interpreter's context
and each vector element or attribute-identifier pair which has a value in
the inverse interpreter's context must have an equivalent value in the
standard interpreter's context. Only when a context satisfies these condi-
tions does it achieve the subgoal.

Consider how this process would work for the waterjug and monkey
procedures. In the waterjug procedure the L7 if statement forces the
interpreter to branch to L10 rather than to end. If we formed an empty
context at end and moved it through the L7 if statement, it would contain
the constraint '<B>=2'. The set statements which alter <B> occur one line
before statement L10, at L2, at L4, at L9, and one line after statement
L6. If inverse interpretation continued beyond the L7 if statement, then
case analysis would occur and each of these set statements would be reached
by some context. Inverse interpretation of the set statement preceding statement L10 would change the constraint to '0=2', which would simplify to 'FALSE'; hence, this context would be eliminated. A similar contradiction would be found at the L2 and L4 set statements. Inverse interpretation of the L9 set statement would change the constraint to '<A> + <B> - 8 = 2', which would reduce to '<A> + <B> = 10'. This constraint would form a condition on a context produced by the standard interpreter so that the subgoal could be stated as follows: reach statement L9 with a context in which <A> + <B> = 10. Similarly, the set statement following L6 would produce the subgoal: reach the statement following L6 with <A> + <B> = 2.

When the executive tries to satisfy these subgoals there will be no difficulty in interpreting contexts to the two set statements, but none of these contexts will satisfy the conditions imposed by the inverse interpreter's contexts. In this case new subgoals can be formed by continuing interpretation of the inverse interpreter's contexts until <A> or <B> are altered again.

For this problem the process of producing subgoals essentially provides a two-way search, since the inverse interpretation begins at end and the standard interpreter's inability to satisfy the conditions at the subgoals causes the inverse interpretation to be continued.

For the monkey problem procedure the subgoal generating statement is the if statement following statement CLIMB. Inverse interpretation produces a subgoal at statement WALK with a context which contains the constraint 'BOX[1]=UNDER.BANANAS' and a subgoal at statement MOVE.BOX with a context containing no binding constraints. Formation of this latter subgoal is precisely the inductive step needed to solve this problem. Given that subgoal, satisfying it and then completing the solution is trivial.
Although we have not given a complete specification of this means-ends analysis procedure, we have outlined its basic form and indicated its usefulness in two example problems. We feel that strategies based on this type of analysis hold the most promise for increasing the effectiveness of ARP's searching abilities.
V. A SAMPLING OF PROBLEMS

In this chapter we present a sampling of the problems which have been stated in REF and discuss ARF's attempts to solve them. A brief account of ARF's history is appropriate at this point. ARF is an 8,000 line program written in the IPL-V programming language [18] and is run on an IBM 360/67 computer [12]. The initial version of the program was completed in the spring of 1968 and the system has evolved since then to its present state. During the 1969 spring semester graduate students in an artificial intelligence class were assigned the task of stating in REF a problem of their own choosing and of running ARF with that problem statement as input. This endeavor gave the students experience with and insight into a running problem solving program, and provided us with a sizable collection of problems stated in REF and solved by ARF. We have chosen for discussion a sample of 16 problems from the approximately 60 problems which have been run by the class and by the author. ARF found a solution to each of the sample problems.

To summarize ARF's behavior on the sample, we present a set of statistics in Table V.1. The problem classes introduced in Part I above (i.e., Boolean constraint satisfaction problems, process constraint satisfaction problems, and heuristic search problems) are used to divide the problems in the table into three groupings. The problems within each group are ordered with respect to the total processing time required by ARF to find a solution. Table entries 1 and 2 for each problem provide a global indication of the length of the problem statement and the difficulty ARF had in finding a solution.

Entries 3-16 in the table provide information related to the interpreter's search in the subproblem space. For these entries we consider
each instance of case analysis during interpretation to define a non-
terminal node in the interpreter's search tree; terminal nodes are formed
by contexts which are found to contain a contradiction, are found to be
equivalent to an earlier produced context, or contain a solution.
Entry 14, average selectivity produced by distance measure, shows the
average number of contexts which were closest to end each time a selection
was made and therefore indicates how effective the distance measure was
at selecting a single context for interpretation. The other selection
entry (no. 15) shows the average number of contexts which were closest
to end and also had the least number of variables. This entry indicates
the effectiveness of the combined distance measure and variable count
criteria at selecting a single context.

Entries 17-23 provide information related to the backtrack search
which is conducted when a context reaches end. The final three entries
provide information related to the size of the solution context and the
space from which the solution was selected.

BOOLEAN CONSTRAINT SATISFACTION PROBLEMS

For each of these problems ARF was able to interpret a single
context to end without doing any case analysis. ARF is most effective
on problems such as these where no case analysis is necessary because
its constraint satisfaction methods are considerably more powerful than
the heuristic search methods it uses to guide the interpreter.

An English statement and a REF statement of each of the sample
Boolean constraint satisfaction problems follows.

A Sorting Problem

English statement: Sort in ascending numerical order the N distinct
integers in the vector INPUT and place them in the vector OUTPUT.
In this particular case \( N \) is 5 and vector \( INPUT \) is 3, 7, 9, 6, 2.

**REF statement:**

```
begin;
    set vector INPUT to 3, 7, 9, 6, 2;
    set \(<N>\) to 5;
    for I <- \(<N>\) goto L1;
L1:       set OUTPUT[<I>] to INPUT[select(1, \(<N>\))];
              for I <- \(<N>\) + 1 goto L2;
L2:       condition OUTPUT[<I>] < OUTPUT[<I> + 1];
end;
```

If REF is allowed parameterized procedures (or subroutines), then this REF procedure could have the vector \( INPUT \) and the integer \( N \) as input parameters and thereby become the statement of an entire class of sorting problems.

**A Magic Square Problem**

**English statement:** Assign one of the first nine positive integers to each element of a 3 \( \times \) 3 matrix such that no integer appears more than once in the matrix and the sum of the integers in each row, column, and diagonal is 15.

The REF statement of this problem was presented above in Figure I.2.

The details of ARF's behavior on this problem are given in Appendix I.

**The Picnic Problem**

**English statement:** Al, Bill, and Chris planned a big picnic. Each boy spent 9 dollars. Each bought sandwiches, ice cream, and soda pop. For each of these items the boys spent jointly 9 dollars, although each boy split his money differently and no boy paid the same amount of money for two different items. The greatest single expense was what Al paid for ice cream; Bill spent twice as much
for sandwiches as for ice cream. How much did Chris pay for soda pop. (All amounts are in round dollars.) [11]

REF statement:

begin;
set vector A to select(1,9), select(1,9), select(1,9),
select(1,9), select(1,9), select(1,9), select(1,9),
select(1,9), select(1,9);

condition excl(A[4], A[5], A[6]);
condition excl(A[7], A[8], A[9]);
for I = 9 doto L1;
if <I> = 2 then L1;

L1:


end;

In the REF procedure the nine purchase prices are represented by a nine element vector as follows:
<table>
<thead>
<tr>
<th>SANDWICHES</th>
<th>ICE CREAM</th>
<th>SODA POP</th>
</tr>
</thead>
</table>

The set vector statement indicating that each of the purchase prices is to be selected from the interval 1 to 9 represents a departure from the original statement of the problem where only the value of A[9] is required for a solution and no upper bound for the purchase prices is explicitly stated. A more faithful translation of the problem would state that a value for A[9] is to be selected from the positive integers and that each of the other elements of the A vector contains an unknown positive integer; but the only facility in REF for defining unknown quantities is the select function and it requires that a finite range be given.

Note in Table V.1 that the constraint processing methods reduced the solution space from $3.9 \times 10^8$ to 24 before the backtrack search began. This reduction made the problem solvable in a reasonable amount of processing time.

A Packing Problem

**English statement:** Pack three rectangles with dimensions $1 \times 9$, $2 \times 7$, and $2 \times 3$ into a rectangular area with dimensions $4 \times 9$.

**REF statement:**

```
begin;
    set <PIECES> to 3;
    set vector ORIENTATIONS to LENGTH,WIDTH:
    set LENGTH of BLOCK1 to L1;
    set WIDTH of BLOCK1 to W1;
    set LENGTH of BLOCK2 to L2;
    set WIDTH of BLOCK2 to W2;
```
set LENGTH of BLOCK3 to L3;
set WIDTH of BLOCK3 to W3;
set vector BLOCKS to BLOCK1,BLOCK2,BLOCK3;
for I -> <PIECES> do to STARTUP;
set NEAR of (LENGTH of BLOCKS[<I>]) to select(0,30);

STARTUP: set NEAR of (WIDTH of BLOCKS[<I>]) to select(0,30);

condition FAR of (LENGTH of BLOCK1) + -NEAR of (LENGTH of BLOCK1) = 9;
condition FAR of (WIDTH of BLOCK1) + -NEAR of (WIDTH of BLOCK1) = 1;
condition FAR of (LENGTH of BLOCK2) + -NEAR of (LENGTH of BLOCK2) = 7;
condition FAR of (WIDTH of BLOCK2) + -NEAR of (WIDTH of BLOCK2) = 2;
condition FAR of (LENGTH of BLOCK3) + -NEAR of (LENGTH of BLOCK3) = 3
condition FAR of (WIDTH of BLOCK3) + -NEAR of (WIDTH of BLOCK3) = 2;
set <TOP> to 4;
set <BOTTOM> to 0;
set <LEFT> to 0;
set <RIGHT> to 9;
for NEWBLOCK -> <PIECES> do to PLACED;
set <THIS> to BLOCKS[<NEWBLOCK>];
set HORIZONTAL of <THIS> to ORIENTATIONS[select(1,2)];
set VERTICAL of <THIS> to ORIENTATIONS[select(1,2)];
condition excl(HORIZONTAL of <THIS>, VERTICAL of <THIS>);
condition <LEFT> + -1 < NEAR of ((HORIZONTAL of <THIS>) of <THIS>);
condition FAR of ((HORIZONTAL of <THIS>) of <THIS>) < <RIGHT> + 1;
condition <BOTTOM> + -1 < NEAR of ((VERTICAL of <THIS>) of <THIS>);
condition FAR of ((VERTICAL of <THIS>) of <THIS>) < <TOP> + 1;
for J -> <NEWBLOCK> + -1 do to PLACED;
set <OTHER> to BLOCKS[<J>];
condition ((FAR of (HORIZONTAL of <THIS>) of <THIS>) < (NEAR of (HORIZONTAL of <OTHER>) of <OTHER>) + 1) \lor ((FAR of (HORIZONTAL of <OTHER>) of <OTHER>) < (NEAR of (HORIZONTAL of <THIS>) of <THIS>) + 1) \lor ((FAR of (VERTICAL of <THIS>) of <THIS>) < (NEAR of (VERTICAL of <OTHER>) of <OTHER>) + 1) \lor ((FAR of (VERTICAL of <OTHER>) of <OTHER>) < (NEAR of (VERTICAL of <THIS>) of <THIS>) + 1)

PLACED;

LAST: end;

In the REF procedure given above, each block is placed in the 4 x 9 space by selecting 'NEAR of LENGTH' and 'NEAR of WIDTH' locations for each block. The selected locations are integers which, when the orientations of the blocks are selected, will represent points on the horizontal or vertical axis of the 4 x 9 space. These location selections are made from a range of 0 to 30 so that this procedure can be used with any space whose dimensions do not exceed 30 x 30. After selecting the "near" locations the "far" locations for each block are defined in terms of the near locations and the block dimensions. The 4 x 9 dimensions of the space are then entered as <TOP> and <RIGHT>. The for loop with index NEWBLOCK is then used to select an orientation for each block and to insure that the blocks' locations are in the 4 x 9 space. The for loop with index J completes the problem statement by insuring that no two blocks overlap in the space.

This REF procedure was written by W. Mann and represents one of several REF representations which he has written for this class of problems.

One significant issue which has arisen from Mann's work with these problems is that of the recognition of symmetries. The number of steps in ARF's backtrack search appears to grow exponentially with the number
of blocks to be packed into the area, and one obvious cause of this
unfortunate growth is that ARF does not take advantage of the symmetries
in the problem to reduce its search. If ARF is to handle successfully
larger problems of this type, then mechanisms are needed to either allow
the user to state symmetries in REF or to enable ARF to derive symmetries
in the problem situation.

The Instant Insanity Problem

English statement: Four cubes are given. Each side of each cube
is colored either red, white, blue, or yellow. Line up the cubes
so that each color appears on each of the top, bottom, front, and
back of the line. [20]

REF statement:*

begin;

        set vector CUBE to CUBE1,CUBE2,CUBE3,CUBE4;
        set vector CUBE1 to RED,BLUE,WHITE,RED,YELLOW,YELLOW;
        set vector CUBE2 to RED,WHITE,BLUE,WHITE,YELLOW,WHITE;
        set vector CUBE3 to YELLOW,WHITE,BLUE,RED,BLUE,WHITE;
        set vector CUBE4 to RED,YELLOW,BLUE,WHITE,YELLOW,BLUE;
        set vector OPP to 3,4,1,2,6,5;
        set vector M to select(1,6),select(1,6),select(1,6),select(1,6),
                        select(1,6),select(1,6),select(1,6),select(1,6);
        for I = 1 to 4 do to I1;
        set <I2> to <I> + <I>;
        set <I4> to <I2> + <I2>;
        condition ~(M[<I2> + -1] = M[<I2>]);
        condition ~(M[<I2>] = OPP[M[<I2> + -1]]);
        set TEST[<I4> + -3] to CUBE[<I>][M[<I2> + -1]];

*This is a slight modification of a REF procedure written by D. Solow.

82
set TEST[<i4> + -2] to CUBE[<j>][M[<i2>]];

set TEST[<i4> + -1] to CUBE[<j>][OPP[M[<i2> + -1]]];

L1: set TEST[<i4>] to CUBE[<j>][OPP[M[<i2>]]];

for J -> 4 doto L2;

L2: condition excl(TEST[<j>], TEST[<j> +4], TEST[<j> +8], TEST[<j> + 12]);

end;

The REF procedure makes two selections for each cube, i.e., M[2*j - 1]
and M[2*j] for each cube j. These selected values could be considered the
top and front for each cube. After the selections are made, the four
relevant sides of each cube are entered into the TEST vector and the
problem's excl conditions are stated.

Note in Table V.1 that although the constraint processing methods
were not able to reduce the solution space size before the backtrack search
began, their deductions allowed a solution to be found in a space with $10^6$
elements in only 28 search steps.

The Couples Tennis Problem

English Statement: Six couples took part in a tennis match. Their
names were Howard, Kress, McLean, Randolph, Lewis, and Rust. The
first names of their wives were Margaret, Susan, Laura, Diana, Grace
and Virginia. Each of the ladies hailed from a different city;
Fort Worth Texas, Wichita Kansas, Mt. Vernon New York, Boston Mass.,
Dayton Ohio, Kansas City Mo.; and finally, each of the women had a
different hair color, namely black, brown, gray, red, auburn, and
blond.

The following pairs played doubles: Howard and Kress against
Grace and Susan, McLean and Randolph against Laura and Susan. The
ladies with black and brown hair played first against Howard and
McLean, then against Randolph and Kress. The following singles were
played: Grace against McLean, Randolph and Lewis, the gray haired lady against Margaret, Diana and Virginia, the lady from Kansas City against Margaret, Laura and Diana, Margaret against the ladies with auburn and blond hair, the lady from Wichita against Howard and McLean, Kress against Laura and Virginia, the lady from Mt. Vernon against the ladies with red and black hair, McLean against Diana and Virginia, and the girl from Boston against the lady with gray hair.

Finally, Rust played against the lady with black hair, the auburn girl against Diana, Lewis against the girl from Kansas City, the lady from Mt. Vernon against Laura, the one from Kansas City against the auburn one, the woman from Wichita against Virginia, Randolph against the girl from Mt. Vernon, and the lady from Fort Worth against the redhead.

There is only one other fact we ought to know to be able to find the last names, home towns, and hair colors of all six wives, and that is the fact that no married couple ever took part in the same game.

REF Statement:

```plaintext
begin;
    set HOWARD of HUSBAND to select(1,6);
    set KRESS of HUSBAND to select(1,6);
    set MCLEAN of HUSBAND to select(1,6);
    set RANDY of HUSBAND to select(1,6);
    set LEWIS of HUSBAND to select(1,6);
    set RUST of HUSBAND to select(1,6);
```

†This REF procedure was written by L. Snyder and R. Teitelbaum
set BLACK of HAIR to select(1,6);
set BROWN of HAIR to select(1,6);
set GREY of HAIR to select(1,6);
set BLOND of HAIR to select(1,6);
set RED of HAIR to select(1,6);
set AUBURN of HAIR to select(1,6);
set FWT of HOME to select(1,6);
set KMO of HOME to select(1,6);
set WK of HOME to select(1,6);
set MVNY of HOME to select(1,6);
set BM of HOME to select(1,6);
set DAT of HOME to select(1,6);

set vector L to Howard, Howard, Kress, Kress, McLean, McLean,
    Randy, Randy, Howard, Howard, McLean, McLean, Randy, Randy,
    Kress, Kress, McLean, Marge, Diana, Marge, Laura, Marge, Marge,
    Howard, McLean, Kress, Kress, MVNY, MVNY, McLean, McLean, BM,
    RUST, DIANA, LEWIS, LARUA, KMO, VIRGIN, RANDY, FWT, RANDY, LEWIS;

set vector R to grace, susan, grace, susan, laura, susan, laura,
    susan, black, brown, black, brown, black, brown, black, brown,
    grace, grey, virgin, KMC, DIANA, AUBURN, BLOND, WK, WK, LAURA,
    VIRGIN, RED, BLACK, DIANA, VIRGIN, GREY, BLACK, AUBURN, KMO,
    MVNY, AUBURN, WK, MVNY, RED, GRACE, GRACE:

set MARGE of NAME to 1;
set SUSAN of NAME to 2;
set LAURA of NAME to 3;
set DIANA of NAME to 4;
set GRACE of NAME to 5;
set VIRGIN of NAME to 6;
condition excl(HOWARD of HUSBAND, KRESS of HUSBAND, RUST of HUSBAND, LEWIS of HUSBAND, MCLEAN of HUSBAND, RANDY of HUSBAND);

condition excl(BLACK of HAIR, RED of HAIR, BLOND of HAIR, BROWN of HAIR, AUBURN of HAIR, GREY of HAIR);

condition excl(FWT of HOME, KMO of HOME, BM of HOME, MVNY of HOME, DAT of HOME, WK of HOME);

for I - 42 doto LAST;

set <T> to L[<I>];

set <U> to R[<I>];

if <T> = HOWARD \lor <T> = KRESS \lor <T> = RUST \lor <T> = LEWIS \lor <T> = RANDY \lor <T> = MCLEAN then MAN;

if <T> = MARGE \lor <T> = VIRGIN \lor <T> = GRACE \lor <T> = DIANA \lor <T> = LAURA \lor <T> = SUSAN then WOMAN;

CITY.HAIR:condition ~ ( <T> of HOME = <U> of HAIR);

goto LAST;

MAN:if <U> = MARGE \lor <U> = VIRGIN \lor <U> = SUSAN \lor <U> = LAURA \lor <U> = GRACE \lor <U> = DIANA then MAN.WOMAN;

if <U> = BLACK \lor <U> = BROWN \lor <U> = BLOND \lor <U> = RED \lor <U> = GREY \lor <U> = AUBURN then MAN.HAIR;

MAN.CITY:condition ~ ( <T> of HUSBAND = <U> of HOME);

goto LAST;

MAN.WOMAN:condition ~ ( <T> of HUSBAND = <U> of NAME);

goto LAST;

MAN.HAIR:condition ~ ( <T> of HUSBAND = <U> of HAIR);

goto LAST;

WOMAN:if <U> = RED \lor <U> = BLOND \lor <U> = BROWN \lor <U> = BLACK \lor <U> = GREY \lor <U> = AUBURN then WOMAN.HAIR;
WOMAN.CITY:condition \sim (\text{T} \text{ of NAME} = \text{U} \text{ of HOME});

goto \text{LAST};

WOMAN.HAIR:condition \sim (\text{T} \text{ of NAME} = \text{U} \text{ of HAIR});

\text{LAST};

\text{end;}

The REF procedure for this problem assumes six ladies identified with the integers 1 through 6. One of the six ladies is selected for each husband, hair color, and hometown, and each lady is assigned a name. The \text{LAST for} loop is then used to form the inequality constraints for each of the tennis matches. In the loop a doubles match is treated as four singles matches; e.g., Howard and Kress playing Grace and Susan in doubles is equivalent (for this problem) to Howard playing Grace, Howard playing Susan, Kress playing Grace, and Kress playing Susan, all in singles. The \text{for} loop was used instead of stating the 42 conditions as a matter of convenience to ease the chore of writing the procedure.

This puzzle is the "largest" problem in the sample in several ways; for example, the longest REF procedure, the most statements on the solution interpretation path, the most constraints in the solution context at completion of interpretation, etc. Most of ARF's processing time is spent interpreting through the \text{LAST loop} 42 times where the bulk of the constraints are formed.

\textbf{The Eight Queens Problem}

\textbf{English statement}: Place eight pieces on a chess board such that each row, column, and diagonal contains no more than one piece. [\text{-}]

\textbf{REF statement}:

\text{begin;}

\text{set vector R to select(1,8), select(1,8), select(1,8), select(1,8),}
\text{ select(1,8), select(1,8), select(1,8), select(1,8);}
The formulation of this problem used in the REF procedure is derived from Floyd [8] and is not intended to be an intuitive statement of the problem. The selections are made in the procedure by selecting a column number for each of the 8 rows on the board.

At each step in ARP's backtrack search, processing of the excl constraints deleted each excluded element from the ranges of the remaining unvalued variables. Also, at each step in the search the most constrained variable (i.e., the one with the smallest remaining range) was assigned a value. These features of ARP's constraint satisfaction methods enabled a solution to be found in only 44 search steps.

The Confusion of Patents Problem

English statement: A certain patent attorney was astonished when he received the simultaneous allowance of five patents, for five separate clients, each of whom lived in a different city.

His astonishment turned to chagrin, however, when he learned what had happened to the patents. They had been received in his office on the same day, but due to an error of a new clerk were sent out in wrong envelopes. Each client received a patent—-but not his own.

The inventor of the steam-shovel received the mouse-trap patent, while the inventor of the latter found in his mail the papers which should have gone to Mr. Green. Mr. Blue received the patent for the
rumble-seat awning. Mr. Black's patent was sent to Chicago; the patent which should have gone there was sent to Boston.

Mr. Brown had the patent intended for New York, Mr. White had Mr. Brown's patent. The non-refillable bottle patent was sent to Los Angeles; the inventor of the bottle received the patent of the Cleveland client, while in Cleveland the surprised client received a patent for an anti-snore device.

Who should have received what where?

REF Statement:

begin;

set vector INVENTION to STEAM.SHOVEL,MOUSE.TRAP,RUMBLESEAT,AWNING,NONREFILLABLE.BOTTLE,ANTISNORE.DEVICE;

set vector PATENT.SENT to INVENTION[select(1,5)],INVENTION[select(1,5)],INVENTION[select(1,5)],INVENTION[select(1,5)];

set vector TEMPl to CHICAGO,BOSTON,NEW.YORK,LOS.ANGELES,CLEVELAND;
set vector CITY to TEMPl[select(1,5)], TEMPl[select(1,5)], TEMPl[select(1,5)], TEMPl[select(1,5)];

set vector TEMP2 to GREEN,BLUE,BLACK,BROWN,WHITE;

set vector INVENTOR to TEMP2[select(1,5)], TEMP2[select(1,5)], TEMP2[select(1,5)], TEMP2[select(1,5)];

condition excl(PATENT.SENT[1],PATENT.SENT[2],PATENT.SENT[3],PATENT.SENT[4],PATENT.SENT[5],CITY[1],CITY[2],CITY[3],CITY[4],CITY[5],INVENTOR[1],INVENTOR[2],INVENTOR[3],INVENTOR[4],INVENTOR[5]);

for I = 5 downto 1;

set CITY of INVENTION[<I>] to CITY[<I>];

set INVENTOR of INVENTION[<I>] to INVENTOR[<I>];
\begin{verbatim}
set PATENT.SENT of CITY[<>] to PATENT.SENT[<>];
set PATENT.SENT of INVENTOR[<>] to PATENT.SENT[<>];

ll: condition ~ (PATENT.SENT[<>] = INVENTION[<>]);

condition PATENT.SENT of INVENTOR of STEAM.SHOVEL = MOUSE.TRAP;
condition INVENTOR of PATENT.SENT of INVENTOR of MOUSE.TRAP = GREEN;
condition PATENT.SENT of BLUE = RUMBLESEAT.AWNING;
condition INVENTOR of PATENT.SENT of CHICAGO = BLACK;
condition CITY of PATENT.SENT of BOSTON = CHICAGO;
condition CITY of PATENT.SENT of BROWN = NEW.YORK;
condition INVENTOR of PATENT.SENT of WHITE = BROWN;
condition PATENT.SENT of LOS.ANGELES = NONREFILLABLE.BOTTLE;
condition CITY of PATENT.SENT of INVENTOR of NONREFILLABLE.BOTTLE = CLEVELAND;

condition PATENT.SENT of CLEVELAND = ANTISNORE.DEVICE;

end;
\end{verbatim}

The selections are made in this procedure so that for \( i = 1, \ldots, 5 \) the \( i \)th selected inventor invented the \( i \)th element of the INVENTOR vector in the \( i \)th selected city and he received the \( i \)th selected patent. The \textbf{for} loop associates with each invention the city in which it was invented and its inventor, with each city the invention whose patent was sent to the city, and with each inventor the invention whose patent was sent to the inventor. These associations do not express all the inter-relationships among the problem elements, but they are sufficient to allow statement of the problem's constraints.

Note in the statistics table that ARF spends 5 minutes and 34 seconds interpreting only 54 statements. The slowness of the interpreter is caused by the number of cases that are generated during the interpretation.
of the statements 'set PATENT.SENT of CITY[<I>] to PATENT.SENT[<I>]' and 'set PATENT.SENT of INVENTOR[<I>] to PATENT.SENT[<I>]'. Since CITY [<I>] and INVENTOR[<I>] are selected quantities, the interpreter must generate cases for each occurrence of 'PATENT.SENT of x' (for any x) in the context. The excl constraint from line 12 of the procedure creates a contradiction in all but one of these cases each time (see table entry 7, number of contexts eliminated by constraint processing during interpretation), but a large amount of processing time is consumed in the case consideration.

PROCESS CONSTRAINT SATISFACTION PROBLEMS

An English statement and a REF statement of each of the sample process constraint satisfaction problems follows.

The John and Mike Tennis Problem

**English statement:** John plays Mike in tennis. John defeats Mike 6 games to 3. On five occasions the person who served lost the game. Who served first?

**REF statement:**

begin;

  set vector PARITY to ODD,EVEN;

  set vector ODD to 1,3,5,7,9;

  set vector EVEN to 2,4,6,8;

  set <ODD> to 5;

  set <EVEN> to 4;

  set vector WIN to select(0,1),select(0,1),select(0,1),select(0,1),
                  select(0,1),select(0,1),select(0,1),select(0,1),1;


*This is a modification of a REF procedure written by A. M. Farley.
set <MIKE.SERVE> to PARITY[select(1,2)];
set <JOHN.SERVE> to PARITY[select(1,2)];
condition ~ (<MIKE.SERVE> = <JOHN.SERVE>);
set <LOST> to 0;
for I = <JOHN.SERVE> goto L1;
L1: set <LOST> to <LOST> + 1 + -WIN(<JOHN.SERVE>[<I>]);
   for I = <MIKE.SERVE> goto L2;
L2: set <LOST> to <LOST> + WIN(<MIKE.SERVE>[<I>]);
condition <LOST> = 5;
End;

The procedure first selects a winner for each game by setting WIN[i] to 1 if John won the ith game or by setting WIN[i] to 0 if Mike won the ith game. Note that since John won the set he necessarily won the ninth game. After selecting the games which each player served, a sum is computed of the number of games in which the server lost. The procedure ends by requiring this sum to be 5.

The most frequent difficulty encountered by users of ARF is that the REF procedure forces ARF to do so much case analysis that the program is essentially enumerating each possible solution. For example, consider the following alternative statement of the L1 loop for this problem:

for I = <JOHN.SERVE> goto L1;
if WIN(<JOHN.SERVE>[<I>]) = 1 then L1;
set <LOST> to <LOST> + 1;
L1:;

The ARF interpreter would do a binary case analysis each time this loop is interpreted. This case analysis would cause ARF to create 56 cases, i.e., one for each possible combination of wins and losses which satisfy the other constraints of the problem. Such a case explosion greatly increases the
amount of processing time required for ARF to find a solution.

A Crypt Addition Problem

**English statement:** Assign a decimal digit to each of the letters in the words SEND, MORE, and MONEY such that when the letters are replaced by the corresponding digits the following summation is true:

\[
\begin{align*}
\text{SEND} \\
+ & \text{MORE} \\
\hline
\text{MONEY}
\end{align*}
\]

No digit may be assigned to more than one letter, and leading zeros are not allowed in the numbers formed by the addends and the sum.

The **REF statement** of this problem is presented in Figure I.3 and a discussion of ARF's behavior in solving the problem is given in the section of Part III entitled "An Example of ARF's Expression Manipulation" and in the section of Part IV entitled "Selecting the Closest Object to the Goal."

A Sequence Formation Problem

**English statement:** For a given pair \((m, n)\) of positive integers, find a sequence \(a_1, a_2, \ldots, a_n\) of integers \(a_i\) such that \(0 \leq a_i \leq n\), \(a_i = 0\), and \(a_n \leq m\); also, for all \(i\) and \(j\) such that \(j \leq n\) and \(i < j\),

\[
D(\{a_i, a_j\}, \{a_{i+1}, \ldots, a_{j-1}, n\}) \leq |a_i - a_j|, \text{ where } D(S, T) \text{ denotes the distance between sets } S \text{ and } T \text{ and is defined to be } \min_{s \in S, t \in T} |s - t|.
\]

For the procedure given below, \(m = 1\) and \(n = 4\).

**REF statement:**

```
begin;
  set <M> to 1;
  set <N> to 4;
  for I = <N> doto I1;
```

*This problem was introduced to me by D. Loveland.*
L1: set A[<I>] to select(0, <N>);
    condition A[1] = 0;
    condition A[<N>] < <N>+1;
    for j ← <N> goto L2;
    for I ← <J>−1 goto L2;
    set vector B to A[<I>], A[<J>];
    set <DIS> to <N>;
    for K ← 2 goto L3;
    if ~ (<N>−B[<K>] < <DIS>) then L4;
    set <DIS> to <N>−B[<K>];
L4: for L ← <J>−<I>−1 goto L3;
    set <DIF> to A[<I>+<J>]−B[<K>];
    if ~ (<DIF> < 0) then L5;
    set <DIF> to −<DIF>;
L5: if ~ (<DIF> < <DIS>) then L3;
    set <DIS> to <DIF>;
L3::
    if ~ (<BOUND> < 0) then L2;
    set <BOUND> to −<BOUND>;
L2: condition <DIS> < <BOUND>+1;
end;

Determination of the existence of these sequences for various values of m and n is of mathematical interest and the problem is well suited for ARF. Although the case shown above (m = 1, n = 4) is nontrivial, the slowness of ARF's interpreter prevents the program from solving any unknown cases in a reasonable amount of processing time.
A Hypothesis Formation Problem

English statement: The four-tuples E1,...,E8 are defined as follows:

E1: 0, 1, 0, Y  
E2: 0, 1, 1, N  
E3: 1, 1, 0, Y  
E4: 1, 1, 1, Y  
E5: 0, 0, 0, N  
E6: 0, 0, 1, N  
E7: 1, 0, 0, Y  
E8: 1, 0, 1, Y

Find sets H(1), H(2),..., H(M) (1 ≤ M ≤ 5) such that:

1) Each Ei is an element of the set H(J) if and only if its fourth member is Y.

2) Each H(J) is a subset of the set whose elements are the triples (0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),..., (1, 1, 1). Any H(J) may be defined by a triple (Z1, Z2, Z3), where each Z is 0, 1, or 2, and for each Zk equal to 0 or 1 a triple (X1, X2, X3) is an element of H(J) if and only if Xk = Zk.

3) Each Ei is an element of H(J) if and only if the triple formed by its first three members is an element of H(J).

Solution:

H = (2, 1, 0) U (1, 2, 2).

REF Statement:

begin;
  set vector H to H1,H2,H3,H4,H5;
  set vector E to E1,E2,E3,E4,E5,E6,E7,E8;
  set vector E1 to 0,1,0,Y;
  set vector E2 to 0,1,1,N;
  set vector E3 to 1,1,0,Y;
  set vector E4 to 1,1,1,Y;
  set vector E5 to 0,0,0,N;

95
set vector E6 to 0,0,1,N;
set vector E7 to 1,0,0,Y;
set vector E8 to 1,0,1,Y;
set <M> to select(1,5);
for I -> <M> goto L1;
L1: set vector H[<I>] to select(0,2), select(0,2), select(0,2);
   for I -> 8 goto L2;
   for J -> <M> goto L3;
   for K -> 3 goto L4;
   if ~ (H[<J>][<K>] = 2) \& ~ (H[<J>][<K>] = E[<I>][<K>]) then L3;
L4:;
   condition E[<I>][4] = Y;
   goto L2;
L3:;
   condition E[<I>][4] = N;
L2:;
end;

This problem is a formalization of a typical hypothesis or concept
formation task used in psychological experiments [14]. Such experiments
involve a set of exemplars (usually objects or drawings) which have been
divided into two subsets by the experimenter using some rule (i.e., the
hypothesis or concept). The exemplars are presented one at a time to a
human subject who is asked to state which of the subsets the exemplar
belongs in. After each presentation the experimenter informs the subject
whether he was right or wrong. The subject's goal is to discover a rule
which he can use to correctly place the exemplars into the two sets.
The exemplars usually have an easily recognizable set of attributes which the subject may use in forming a rule. For example, each exemplar might be a card with a circle drawn on it. The attributes relevant for forming the sets could be size of circle (large or small), color of circle (red or blue), and location of circle (on left or right side of the card). These three attributes each having two possible values allow the formation of 8 exemplars and 255 hypotheses. The problem we are considering has the same combinatorial characteristics as this example. In our problem each exemplar is a quadruple whose first three elements are either 0 or 1 and whose fourth element is either Y or N. The first three elements represent the binary values of three attributes and the fourth element denotes which subset the exemplar is in.

In our REF statement of the problem we specify the manner in which the hypothesis is to be stated. The hypothesis is to define the Y set as a disjunction of sets which have certain common attribute values. All exemplars not in the Y set are assumed to be in the N set. We allow a maximum of five disjuncts in the definition of set Y.

To solve this problem ARF conducts basically a depth-first search in the interpretation space performing case analysis at the beginning of the L1 loop and at the if statement preceding statement L4. Since there are two disjuncts in the solution and ARF tries to find the simplest solution to a problem, it considers all possible solutions with one disjunct before considering the necessary second disjunct. Although ARF's search for this problem is lengthy, it is intelligent in that the simplest solutions (i.e., those with one disjunct) are tried first and each time the if
statement preceding L4 is interpreted, processing of the branching condition causes retention of the most general hypothesis which correctly categorizes the exemplars considered to that point.

HEURISTIC SEARCH PROBLEMS

The weaknesses in ARF's search strategies during interpretation show most clearly in heuristic search problems. If the REF procedure for the problem has the form shown in Figure I.4 (i.e., an initialization phase and a loop in which a select function is used in a computed goto statement to select an operator each time through the loop), then the distance measure provides little direction to the search and the search becomes basically breadth-first. The most effective tool ARF has for these problems is the ability to recognize and delete redundant contexts. For some problems this pruning of the search tree occurs frequently enough to allow ARF to find a solution in a reasonable amount of time.

An English statement and a REF statement of each of the sample heuristic search problems follows.

The Monkey Problem

English statement: In a room is a monkey, a box, and some bananas hanging from the ceiling. The monkey wants to eat the bananas, but he cannot reach them unless he is standing on the box when it is sitting under the bananas. How can the monkey get the bananas?

The REF statement of this problem is presented in Figure I.6b and a discussion of ARF's behavior in solving the problem is given in the section of Part IV entitled "Selecting the Closest Object to the Goal."
Note that this procedure is not in the standard heuristic search form discussed above. This procedure's control structure is such that ARF's distance measure is effective at providing direction during interpretation; hence, a solution is found after only a brief search.

A Waterjug Problem

**English statement**: Given a five gallon jug and an eight gallon jug, how can precisely two gallons be put into the five gallon jug? Since there is a sink nearby, a jug can be filled from the tap and can be emptied by pouring its contents down the drain. Water can be poured from one jug into another, but no measuring devices are available other than the jugs themselves.

The REF statement of this problem is presented in Figure 1.5b and a discussion of ARF's behavior in solving the problem is given in the section of Part IV entitled "Selecting the Closest Object to the Goal."

A Programming Problem

**English statement**: Assume a machine with two registers, A and B, each capable of holding a three decimal digit number. Transform the machine from the state \( A = 234, B = 000 \) to the state \( A = 230, B = 000 \) using only the following instructions:

- Shift register A right 1 digit
- Shift register A left 1 digit
- Shift both registers right 1 digit (When shifting both registers they are treated as if concatenated into a single long register)
- Shift both registers left 1 digit
- Load into register B the contents of register A.
REF statement:

begin;

set vector S to 2,3,4,0,0,0;

set vector F to 2,3,0,0,0,0;

ELIM: for I = 6 goto L1;

if ~ (S[I] = F[I]) then WEED;

L1:;

goto QED;

WEED: for J = 6 goto SATSFD;

set I to F[I];

for I = 6 goto FAILCOND;

if I = S[I] then SATSFD.

FAILCOND:;

condition 1 = 2;

SATSFD:;

L2: goto (SL1,SR1,SL2,SR2,LOAD)select (1,5);

SR2: for I = 5 goto SR22;

SR22: set S[-<I>+7] to S[-<I>+6];

set S[1] to 0;

goto ELIM;

SL2: for I = 5 goto SL22;

SL22: set S[I] to S[I+1];

set S[6] to 0;

goto ELIM;

LOAD: for I = 3 goto LOAD2;

LOAD2: set S[I+3] to S[I];

goto ELIM;

SR1: for I = 2 goto SR11;

SR11: set S[-<I>+4] to S[-<I>+3];

set S[1] to 0;
goto ELIM;
SL1:for I = 2 downto SL1;
SL1:set S[i] to S[i+1];
set S[3] to 0;
go to ELIM;
QED: end;

This procedure, written by Anita Jones, represents an attempt to
corporate heuristics intended to assist ARF into the REF statement
of a problem. The heuristic in the procedure above is a pruning mecha-
nism which sends the interpreter to the unsatisfiable condition statement
following statement FAILCOND if there is a digit in the goal state (the
F vector) which does not occur somewhere in the current state (the S
vector). The WEED for loop performs this test after each selection and
execution of a command. Since there is no way of entering a digit in
the registers which does not already appear somewhere in one of the
registers, any state which does not pass this test cannot possibly lead
to a solution and can therefore be "weeded." The full search tree for
this problem has 25 terminal nodes at the second level; 11 of those 25
nodes will be discarded by this WEED heuristic.

If one could write a parameterized REF procedure which defined a
class of problems, then it would be important to be able to augment ARF's
effectiveness for that class of problems by including heuristics in the
REF procedure as was done for this programming problem.

The Missionaries and Cannibals Problem

English statement: Three missionaries and three cannibals wish
to cross a river. The only means of conveyance is a small boat
which has a capacity of two people and which all six know how to
operate. If, at any time, there are more cannibals than
missionaries on either side of the river, those missionaries will be eaten by the cannibals. How can all six get across the river without any missionaries being eaten?

REF Statement:

begin;

set MISSIONARIES of LEFT.SIDE to 3;
set CANNIBALS to LEFT.SIDE to 3;
set MISSIONARIES to RIGHT.SIDE to 0;
set CANNIBALS to RIGHT.SIDE to 0;
set <DEPARTING.SIDE> to LEFT.SIDE;
set <ARRIVING.SIDE> to RIGHT.SIDE;
set <MISSIONARIES> to 0;
set <CANNIBALS> to 0;
L7:goto (L1,L2,L3,L4,L5) select (1,5);
L1: set <MISSIONARIES> to 0;
set <CANNIBALS> to 1;
goto L6;
L2: set <MISSIONARIES> to 0;
set <CANNIBALS> to 2;
goto L6;
L3: set <MISSIONARIES> to 1;
set <CANNIBALS> to 0;
goto L6;
L4: set <MISSIONARIES> to 1;
set <CANNIBALS> to 1;
goto L6;
L5: set <MISSIONARIES> to 2;
set <CANNIBALS> to 0;
L6: condition ~ (MISSIONARIES of <DEPARTING.SIDE> < <MISSIONARIES>);
    condition ~ (CANNIBALS of <DEPARTING.SIDE> < <CANNIBALS>);
    set MISSIONARIES of <DEPARTING.SIDE> to MISSIONARIES of 
        <DEPARTING.SIDE> + -<MISSIONARIES>;
    set CANNIBALS of <DEPARTING.SIDE> to CANNIBALS of <DEPARTING.SIDE> 
        + -<CANNIBALS>;
    condition MISSIONARIES of <DEPARTING.SIDE> = 0 ∨ ~ (MISSIONARIES 
        of <DEPARTING.SIDE> < CANNIBALS of <DEPARTING.SIDE>);
    set MISSIONARIES of <ARRIVING.SIDE> to MISSIONARIES of 
        <ARRIVING.SIDE> + <MISSIONARIES>;
    set CANNIBALS of <ARRIVING.SIDE> to CANNIBALS of <ARRIVING.SIDE> 
        + <CANNIBALS>;
    condition MISSIONARIES of <ARRIVING.SIDE> = 0 ∨ ~ (MISSIONARIES 
        of <ARRIVING.SIDE> < CANNIBALS of <ARRIVING.SIDE>);
    set <MISSIONARIES> to 0;
    set <CANNIBALS> to 0;
    if MISSIONARIES of RIGHT.SIDE = 3 ∨ CANNIBALS of RIGHT.SIDE 
        = 3 then SOLVED;
    set <TEMP> to <DEPARTING.SIDE>;
    set <DEPARTING.SIDE> to <ARRIVING.SIDE>;
    set <ARRIVING.SIDE> to <TEMP>;
    goto L7;
SOLVED: end;

Our experience with this problem has provided an interesting counter-
example to one of the basic premises on which the design of ARF is based.
The premise is that we have made ARF more powerful by using variables and 
thereby allowing many problem states to be represented in a single context 
structure. We would predict that whenever ARF is forced to create a 
large number of cases its effectiveness is severely lessened.
This premise has proven to be generally true in our experience with the program, but consider the following alternative REF statement of the missionaries and cannibals problem. Replace lines 9 through 23 (i.e., statements L7 through the statement preceding L6) with the following lines:

```plaintext
set <MISSIONARIES> to select(0,2);
set <CANNIBALS> to select(0,2);
condition 0 < <CANNIBALS> + <MISSIONARIES>;
condition ~(2 < <CANNIBALS> + <MISSIONARIES>);
```

These statements combine the five operators into a single operator containing two selections. This combining of operators allows ARF to interpret the procedure so that case analysis occurs only at the if statement test for completion. When the if statement is interpreted a case is created in which the solution state is assumed (i.e., MISSIONARIES of RIGHT.SIDE = 3 and CANNIBALS of RIGHT.SIDE = 3) and an attempt is made to satisfy this case; when it is found to be unsatisfiable, interpretation continues with the single remaining case. Hence, with this statement of the missionaries and cannibals problem ARF can proceed without case analysis and therefore should be most effective. In fact, the program would require more than 30 minutes of processing time to solve the problem stated this way. The difficulty is that each boat trip creates two new variables and as many as seven new constraints. Since the solution requires eleven boat trips, the final state will contain 22 variables and as many as 77 constraints. As the context grows to this size, interpretation becomes slower and the elimination of the case which branches to end each time the if statement is interpreted requires more and more processing.
So we see that for this problem at least, ARF is more effective when it creates a context for each possible state and interprets each one separately. When this occurs, the redundant context recognizer is able to eliminate most of the search tree so that, in fact, only a few cases need be interpreted.
VI. SUMMARY

We began by introducing the idea that a nondeterministic programming language could be used as a problem statement language for a general problem solving program. Nondeterministic languages seem to represent a reasonable compromise between the user's desire for ease of problem statement and the designer's desire to be able to create an effective program for solving problems stated in the language. The syntax and semantics of a nondeterministic programming language are formally defined so that problem statements can be algorithmically interpreted, yet the full representational power of the base programming language's control and data structures are available for describing the objects, relations, and functions involved in the statement of a problem.

By considering a particular example of a nondeterministic language, (i.e., REF), we have demonstrated how Boolean constraint satisfaction problems, process constraint satisfaction problems, and heuristic search problems can be stated in a natural manner. We also indicated that suitable facilities could be included in REF for representing other classes of problems such as optimization problems.

In Part II we introduced ARF, a program for solving problems stated in REF. A fundamental goal in the design of ARF was to translate a problem stated as a REF procedure into a form that would allow the application of effective problem solving methods. This goal was achieved by using variables to represent the values of select function calls during the interpretation of a procedure. This use of variables eliminates the necessity for considering a separate interpretation case for each possible value of a select function call and allows the interpreter to derive a set of Boolean expressions which constrain the values of the variables.
The translated problem, consisting of a set of variables, a finite range of possible values for each variable, and a set of constraints on the values of those variables, is in a form to which any of a variety of constraint satisfaction problem solving methods can be applied.

The use of variables is not always effective in eliminating the need for considering multiple cases during interpretation. When this case analysis is required, ARF needs methods for guiding the interpreter as to which case to pursue first, and once a case (i.e., context) is chosen, how long to continue its interpretation before returning to consider another one. The organization of ARF allows the interpretation of each context to proceed in small independent steps so that an executive routine can use heuristic search methods to guide the interpretation with the goal of interpreting a context to the **end** statement.

Therefore, we see that ARF combines constraint satisfaction methods and heuristic search methods to solve a problem. A search is conducted to find a case which allows interpretation to proceed to **end**, and the size of this search space is reduced by the use of variables which allow the representation of many cases as one. For a case in this reduced search space to represent a solution, constraint satisfaction methods must be applied to find acceptable values for its variables and thereby determine which of the many cases represented by the variables is a solution.

In Part III we discussed the design of methods for solving the constraint satisfaction problems created by ARF's interpreter. The most powerful of these methods perform algebraic manipulations on constraints to deduce that a variable can be expressed as a function of other variables, that an element can be deleted from the range of a variable, or that no set of values exists which can satisfy the set of constraints. These are
ARF's most powerful problem solving methods and ARF is most effective when the interpreter produces only a small number of cases so that these methods can play the primary role in solving a problem.

In Part IV we described ARF's methods for guiding the interpreter in its attempt to move a context to end. ARF can eliminate contexts during this search by using the constraint processing methods to deduce contradictions or by discovering redundant contexts which are equivalent to other contexts produced earlier in the interpretation. The search executive attempts to select at each search step the context which is nearest the end statement and has the fewest variables. This strategy gives the search a depth-first character when no new variables are being defined by the interpretation and a breadth-first character when new variables are being defined.

In Part V we presented a sampling of the problems which have been solved by ARF. Our experience with the program has shown that it can successfully solve an interesting class of problems. In addition, the majority of the approximately thirty people who have used REF-ARF found the nondeterministic language an acceptable one for stating problems.
ACKNOWLEDGEMENTS

I wish to thank Allen Newell for his thoughtful guidance which has continually provided insight and inspiration throughout the project. I also wish to thank him for initiating the student use of REF-ARF and to thank the students for providing me with a wealth of experience with the program and its idiosyncrasies. Finally, I wish to thank Stanford Research Institute for providing assistance in the preparation of this document.
REFERENCES


APPENDIX I

Shown below is the output produced by ARF during the solving of the magic square problem (see Fig. 1.2a for an English statement of the problem). The output consists of a listing of the REF procedure as it is read by ARF followed by a semantic trace of the problem solving process. The trace begins by printing each REF statement as it is interpreted. When the end statement is reached, the start of the backtrack search is announced and the context in which the search will be made is printed. Before the first value is assigned to a variable, each of the context's unprocessed constraints is processed. The trace of this processing is followed by another printout of the context. The first value assignment is then made \((S(9) \leftarrow 1)\), and a contradiction is derived. The second and third assignments both succeed and produce a solution as shown in the last context printed. The program then announces that a solution has been found (not shown) and terminates.
begin:
  set vector \( M \) to select(1,9), select(1,9), select(1,9), select(1,9), select(1,9), select(1,9),
  select(1,9), select(1,9), select(1,9), select(1,9), select(1,9);
  condition excl(M[1], M[2], M[3], M[4], M[5], M[6], M[7], M[8], M[9]);
end:

interpret set vector \( M \) to S(1,9), S(1,9), S(1,9), S(1,9), S(1,9), S(1,9),
  S(1,9), S(1,9), S(1,9);
interpret condition excl(M[1], M[2], M[3], M[4], M[5], M[6], M[7], M[8], M[9]);
  add unprocessed constraint excl(M[1], M[2], M[3], M[4], M[5], M[6], M[7],
  M[8], M[9])
Begin the backtrack search with the following context:

Context 14212

Cycle count: 109322

Data structure
M
vector: S(1), S(2), S(3), S(4), S(5), S(6), S(7), S(8), S(9)

Variables

<table>
<thead>
<tr>
<th>S(9)</th>
<th>range: 1 2 3 4 5 6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(8)</td>
<td>range: 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S(7)</td>
<td>range: 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S(6)</td>
<td>range: 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S(5)</td>
<td>range: 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S(4)</td>
<td>range: 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S(3)</td>
<td>range: 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S(2)</td>
<td>range: 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S(1)</td>
<td>range: 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

Processed constraints: none

Unprocessed constraints
S(1)+S(2)+S(3)=15
S(4)+S(5)+S(6)=15
S(7)+S(8)+S(9)=15
S(1)+S(4)+S(7)=15
S(2)+S(5)+S(8)=15
S(3)+S(6)+S(9)=15
S(1)+S(5)+S(9)=15
S(3)+S(5)+S(7)=15
excl(S(1), S(2), S(3), S(4), S(5), S(6), S(7), S(8), S(9)
Process constraint S(1)+S(2)+S(3)=15

set value of S(1) to 15-S(2)-S(3)
  add unprocessed constraint 0<15-S(2)-S(3)
  add unprocessed constraint 15<S(2)-S(3)<10
  add unprocessed constraint excl(15-S(2)-S(3),S(2),S(3),S(4),S(5), S(6),S(7),S(8),S(9))
  add unprocessed constraint 15-S(2)-S(3)+S(5)+S(9)=15
  add unprocessed constraint 15-S(2)-S(3)+S(4)+S(7)=15
process constraint S(4)+S(5)+S(6)=15

set value of S(4) to 15-S(5)-S(6)
  add unprocessed constraint 0<15-S(5)-S(6)
  add unprocessed constraint 15-S(5)-S(6)<10
  add unprocessed constraint -S(2)-S(3)+15-S(5)-S(6)+S(7)=0
  add unprocessed constraint excl(15-S(2)-S(3),S(2),S(3),15-S(5)-S(6),S(5),S(6),S(7),S(8),S(9))
process constraint S(7)+S(8)+S(9)=15

set value of S(7) to 15-S(8)-S(9)
  add unprocessed constraint 0<15-S(8)-S(9)
  add unprocessed constraint 15-S(8)-S(9)<10
  add unprocessed constraint excl(15-S(2)-S(3),S(2),S(3),15-S(5)-S(6),S(5),S(6),15-S(8)-S(9),S(3),S(9))
  add unprocessed constraint S(3)+S(5)+15-S(8)-S(9)=15
process constraint S(2)+S(5)+S(8)=15

set value of S(2) to 15-S(5)-S(8)
  add unprocessed constraint 0<15-S(5)-S(8)
  add unprocessed constraint 15-S(5)-S(8)<10
  add unprocessed constraint 30=S(8)+S(9)+S(5)+S(6)+15-S(5)-S(8)+S(3)
  add unprocessed constraint excl(15-S(5)-S(8)-S(3),15-S(5)-S(8),S(3),15-S(5)-S(6),S(5),S(6),15-S(8)-S(9), S(8),S(9))
  add unprocessed constraint -15-S(5)-S(8)-S(3)+S(5)+S(9)=0
  add unprocessed constraint 5<15-S(5)-S(8)+S(3)
  add unprocessed constraint 15-S(5)-S(8)+S(3)<15
process constraint S(3)+S(6)+S(9)=15

set value of S(3) to 15-S(6)-S(9)
  add unprocessed constraint 0<15-S(6)-S(9)
  add unprocessed constraint 15-S(6)-S(9)<10
  add unprocessed constraint -S(5)-S(8)+15-S(6)-S(9)<0
  add unprocessed constraint 10<=S(5)-S(8)+15-S(6)-S(9)
  add unprocessed constraint S(5)-S(8)-15-S(6)-S(9)+S(9)+S(3)+S(5)+S(9)=15
  add unprocessed constraint excl(S(5)-S(8)-15-S(6)-S(9),15-S(5)-S(8),15-S(6)-S(9),15-S(5)-S(6),S(5),S(6), 15-S(8)-S(9),S(8),S(9))
  add unprocessed constraint S(9)+S(6)+15-S(6)-S(9)=15
  add unprocessed constraint -S(5)-S(9)+15-S(6)-S(9)+S(5)=0
process constraint S(6)+S(9)+S(5)+S(8)+S(5)+S(9)=30

set value of S(6) to 30-S(9)-S(5)-S(8)-S(5)-S(9)
  add unprocessed constraint 0<30-S(9)-S(5)-S(8)-S(5)-S(9)
  add unprocessed constraint 30-S(9)-S(5)-S(8)-S(5)-S(9)<10
  add unprocessed constraint 30-S(9)-S(5)-S(8)-S(5)-S(9)-S(5)-S(9)+S(5)+S(9)<0
  add unprocessed constraint 30-S(9)-S(5)-S(8)-S(5)-S(9)+S(9)+S(5)+S(9)<0
  add unprocessed constraint 15<30-S(9)-S(5)-S(8)-S(5)-S(9)+S(9)+S(5)+S(9)
  add unprocessed constraint 5<30-S(9)-S(5)-S(8)-S(5)-S(9)+S(9)
add unprocessed constraint 30-S(9)-S(5)-S(8)-S(5)-S(9)+S(9)<15
add unprocessed constraint 5<S(5)+30-S(9)-S(5)-S(8)-S(5)-S(9)
add unprocessed constraint S(5)+30-S(9)-S(5)-S(8)-S(5)-S(9)<15

process constraint S(5)=5

set value of S(5) to 5
add unprocessed constraint 15<S(9)+S(8)+5+S(9)
add unprocessed constraint S(9)+S(8)+5+S(9)<25
add unprocessed constraint 15<S(8)+5+S(9)
add unprocessed constraint 5+S(8)+5+S(9)<25
add unprocessed constraint 5+S(9)<15
add unprocessed constraint 5<S(9)
add unprocessed constraint S(5)-15-S(5)+S(8)+S(9)+S(5)+S(8)+15-S(5)-S(8),
30-S(9)-S(8)-S(9)+S(8)-S(5)-S(9)+S(9)+S(5)+S(8),15-S(5)-S(8),
S(5)-S(8)-5-S(9),15-S(8)-S(9),S(8),S(9))
add unprocessed constraint 20<S(9)+5+S(8)+5+S(9)
add unprocessed constraint S(9)+5+S(8)+5+S(9)<30
add unprocessed constraint 5<S(8)
add unprocessed constraint 5+S(8)<15

process constraint S(8)+S(9)<15
process constraint 5<S(8)+S(9)
process constraint 10<S(9)+S(8)+S(9)
process constraint S(9)+S(8)+S(9)<20
process constraint S(9)+S(8)+S(9)<20
process constraint S(9)+S(8)+S(9)<20
process constraint S(9)+S(8)+S(9)<20
process constraint S(9)+S(8)+S(9)<20

Context 14212

Data Structure

\[ M \]

vector: 10-S(9),10-S(9),-5+S(8)+S(9),-10+S(9)+S(8)+S(9),S(9),5,20-S(9)-S(8)-S(9),S(8),S(9),S(9)

Variables

S(9)
range: 1 2 3 4 6 7 8 9
S(8)
range: 1 2 3 4 6 7 8 9
S(7)
value: 15-S(8)-S(9)
S(6)
value: 20-S(9)-S(8)-S(9)
S(5)
value: 5
S(4)
value: -10+S(9)+S(8)+S(9)
S(3)
value: -5+S(8)+S(9)
S(2)
value: 10-S(8)
S(1)
value: 10-S(9)

Processed constraints

\[ \text{excl}(10-S(9),10-S(8),-5+S(8)+S(9),-10+S(9)+S(8)+S(9)+S(9),S(9),20-S(9)-S(8)-S(9),15-S(8)-S(9),S(8),S(9)) \]
S(9)+S(8)+S(9)<20
10<S(9)+S(8)+S(9)
5<S(8)+S(9)
S(8)+S(9)<15
S(9)←1

set value of S(9) to 1
add unprocessed constraint S(8)+1<15
add unprocessed constraint 5<S(8)+1
add unprocessed constraint 10<1+S(8)+1
add unprocessed constraint 1+S(8)+1<20
add unprocessed constraint excl(10-1,10-S(8),-5+S(8)+1,-10+1+S(8)+1,5,20-1-S(8)-1,15-S(8)-1,S(8),1)
process constraint 4<S(8)
process constraint 8<S(8)
set value of S(8) to 9
add unprocessed constraint excl(9,10-9,-4+9,-8+9,5,18-9,14-9,9,1)
contradiction
fail
contradiction
fail
S(9)←2

set value of S(9) to 2
add unprocessed constraint S(8)+2<15
add unprocessed constraint 5<S(8)+2
add unprocessed constraint 10<2+S(8)+2
add unprocessed constraint 2+S(8)+2<20
add unprocessed constraint excl(10-2,10-S(8),-5+S(8)+2,-10+2+S(8)+2,5,20-2-S(8)-2,15-S(8)-2,S(8),2)
process constraint 3<S(8)
process constraint 6<S(8)
process constraint excl(8,10-S(8),-3+S(8),-6+S(8),5,16-S(8),13-S(8),S(8),2)

Context 21549
Cycle count: 631517

Data structure
M
Vector: 8,10-S(8),-3+S(8),-6+S(8),5,16-S(8),13-S(8),S(8),2

Variables
S(9)
value: 2
S(6)
range: 7 9
S(7)
value: 13-S(8)
S(6)
value: 16-S(8)
S(5)
value: 5
S(4)
value: -8+S(8)
S(3)
value: -3+S(8)
S(2)
value: 10-S(8)
S(1)
value: 8

Processed constraints
excl(8,10-S(8),-3+S(8),-6+S(8),5,16-S(8),13-S(8),S(8),2)
S(8) = 7

set value of S(8) to 7

add unprocessed constraint excl(8, 10-7, -3+7, -6+7, 5, 16-7, 13-7, 7, 2)

Context 280049

Data structure

M

Vector: 8, 3, 4, 1, 5, 9, 6, 7, 2

Variables

S(9)
value: 2

S(8)
value: 7

S(7)
value: 6

S(6)
value: 9

S(5)
value: 5

S(4)
value: 1

S(3)
value: 4

S(2)
value: 3

S(1)
value: 8

Processed constraints: none

Unprocessed constraints: none

Cycle count: 647485
\( \langle \text{procedure} \rangle := \text{begin} \langle \text{statement} \rangle \ldots ; \langle \text{statement} \rangle ; \langle \text{end} \rangle \)

\( \langle \text{end} \rangle := \text{end} \mid \langle \text{label} \rangle :: \text{end} ; \)

\( \langle \text{statement} \rangle := \langle \text{label} \rangle :: \langle \text{unlabeled.statement} \rangle \mid \langle \text{unlabeled.statement} \rangle \mid \langle \text{label} \rangle :: \)

\( \langle \text{unlabeled.statement} \rangle := \text{goto} \langle \text{label} \rangle \mid \text{set} \langle \text{slot.expression} \rangle \text{ to } \langle \text{expression} \rangle \mid \text{set vector} \langle \text{identifier.expression} \rangle \text{ to } \langle \text{expression} \rangle \ldots , \),

\( \langle \text{expression} \rangle \mid \text{for} \langle \text{identifier} \rangle \rightarrow \langle \text{integer.expression} \rangle \text{ do } \langle \text{label} \rangle \mid \text{if} \langle \text{Boolean.expression} \rangle \text{ then } \langle \text{label} \rangle \mid \text{condition} \langle \text{Boolean.expression} \rangle \mid \text{goto} \langle \text{label} \rangle , \ldots , \langle \text{label} \rangle \mid \langle \text{integer.expression} \rangle \)

\( \langle \text{label} \rangle := \langle \text{identifier} \rangle \)

\( \langle \text{expression} \rangle := \langle \text{slot.expression} \rangle \mid \langle \text{identifier.expression} \rangle \mid \langle \text{integer.expression} \rangle \mid \langle \text{Boolean.expression} \rangle \)

\( \langle \text{slot.expression} \rangle := \langle \langle \langle \text{identifier.expression} \rangle \rangle \rangle \mid \langle \text{identifier.expression} \rangle \langle \langle \langle \text{integer.expression} \rangle \rangle \rangle \mid \langle \text{identifier.expression} \rangle \text{ of } \langle \text{identifier.expression} \rangle \)

\( \langle \text{identifier.expression} \rangle := \langle \text{slot.expression} \rangle \mid \langle \text{identifier} \rangle \mid \langle \langle \text{identifier.expression} \rangle \rangle \)

\( \langle \text{integer.expression} \rangle := -\langle \text{integer.expression} \rangle \mid \langle \text{integer.expression} \rangle + \langle \text{integer.expression} \rangle \mid \langle \text{select} \langle \langle \text{integer.expression} \rangle , \langle \text{integer.expression} \rangle \rangle \rangle \mid \langle \text{slot.expression} \rangle \mid \langle \text{integer} \rangle \mid \langle \langle \text{integer.expression} \rangle \rangle \)

\( \langle \text{Boolean.expression} \rangle := \langle \langle \langle \text{identifier.expression} \rangle \rangle \rangle \mid \langle \text{integer.expression} \rangle \mid \langle \text{integer.expression} \rangle \mid \langle \text{integer.expression} \rangle \mid \langle \text{ref} \langle \langle \langle \text{expression} \rangle \rangle \rangle \mid \langle \text{expression} \rangle \mid \langle \text{expression} \rangle , \ldots , \langle \text{expression} \rangle \rangle \mid \langle \text{Boolean.expression} \rangle \mid \langle \text{Boolean.expression} \rangle \mid \langle \text{Boolean.expression} \rangle \mid \langle \text{Boolean.expression} \rangle \mid \langle \text{TRUE} \rangle \mid \langle \text{FALSE} \rangle \mid \langle \text{Boolean.expression} \rangle \mid \langle \langle \text{Boolean.expression} \rangle \rangle \)

\( \langle \text{identifier} \rangle := \langle \text{letter} \rangle \langle \text{alphanumeric} \rangle \ldots \langle \text{alphanumeric} \rangle \mid \langle \text{letter} \rangle \)

\( \langle \text{alphanumeric} \rangle := \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \mid . \)

\( \langle \text{integer} \rangle := \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \ldots \langle \text{digit} \rangle \)

\( \langle \text{digit} \rangle := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \)

\( \langle \text{letter} \rangle := A | B | \ldots | Z \)

\text{TA-710522-112}

\text{Figure I.1 Description of REF Syntax}
Assign one of the first 9 positive integers to each element of a 3-by-3 matrix such that no integer appears more than once in the matrix and the sum of the integers in each row, column, and diagonal is 15.

Figure I.2a  English Statement of the Magic Square Problem

---

begin:

set vector M to select(1,9), select(1,9), select(1,9), select(1,9),
        select(1,9), select(1,9), select(1,9), select(1,9), select(1,9);

condition excl(M[1], M[2], M[3], M[4], M[5], M[6], M[7], M[8], M[9]);

end:

TA-710522-113

Figure I.2b  REF Statement of the Magic Square Problem
Assign a decimal digit to each of the letters in the words 'send', 'more', and 'money' such that when the letters are replaced by the corresponding digits the following summation is true:

\[
\text{SEND} + \text{MORE} = \text{MONEY}
\]

No digit may be assigned to more than one letter, and leading zeros are not allowed in the numbers formed by the addends and the sum.

---

**Figure I.3a** English Statement of a Crypt-Addition Problem

```plaintext
begin;
  set vector A1 to X,S,E,N,D;
  set vector A2 to X,M,O,R,E;
  set vector SUM to M,O,N,E,Y;
  set vector L to D,N,E,S,R,O,M,Y;
  for i = 8 do to L1;
    L1: set <L[i]> to select(0,9);
    <L[7]>,<L[8]>);
    condition ~(<L> = 0) \& ~(<S> = 0);
    set <CARRY> to 0;
    for j = 4 do to L2;
    set <I> to 6 + -<I>;
    if <A1[<I>] > + <A2[<I>] > + <CARRY> < 10 then L3;
    set <CARRY> to 1;
    goto L2;
    set <CARRY> to 0;
  L2:;
  condition <M> = <CARRY>;
end;
```

---

**Figure I.3b** REF Statement of a Crypt-Addition Problem

TA-710522-114
Figure 1.4 General Form for REF Statement of a Heuristic Search Problem
Given a five gallon jug and an eight gallon jug, how can precisely two gallons be put into the five gallon jug? Since there is a sink nearby, a jug can be filled from the tap and can be emptied by pouring its contents down the drain. Water can be poured from one jug into another, but no measuring devices are available other than the jugs themselves.

Figure I.5a  English Statement of a Water Jug Problem

```
begin;
    set <A> to 0;
    set <B> to 0;
L10: set <D> to select(1,6);
goto (L1,L2,L3,L4,L5,L6) <J>;
L1: set <A> to 8;
goto L7;
L2: set <B> to 5;
goto L7;
L3: set <A> to 0;
goto L7;
L4: set <B> to 0;
goto L7;
L5: if 5 < <A> + <B> then L9;
    set <A> to <A> + <B>;
goto L4;
L9: set <B> to <A> + <B> + 5;
goto L1;
L6: if 8 < <A> + <B> then L8;
    set <B> to <A> + <B>;
goto L3;
L8: set <A> to <A> + <B> + 5;
goto L2;
L7: if ~ (<B> = 2) then L10;
end;
```

Figure I.5b  REF Statement of a Water Jug Problem
In a room is a monkey, a box, and some bananas hanging from the ceiling. The monkey wants to eat the bananas, but he cannot reach them unless he is standing on the box when it is sitting under the bananas. How can the monkey get the bananas?

Figure 1.6a English Statement of the Monkey Problem

begin:
    set vector X to X1, X2, UNDER.BANANAS;
    set vector Y to ON.FLOOR, ON.BOX;
    set vector MONKEY to X1, ON.FLOOR;
    set vector BOX to X2, ON.FLOOR;
WALK: set MONKEY[1] to X[select(1,3)];
    if ¬(MONKEY[1] = BOX[1]) then WALK;
L1: goto (WALK,CLIMB, MOVE.BOX)select(1,3);
CLIMB: set MONKEY[2] to ON.BOX;
    if ¬(MONKEY[1] = UNDER.BANANAS) then STEP.DOWN;
    goto (GET.BANANAS,STEP.DOWN)select(1,2);
STEP.DOWN: set MONKEY[2] to ON.FLOOR;
    goto L1;
MOVE.BOX: set MONKEY[1] to X[select(1,3)];
    set BOX[1] to MONKEY[1];
    goto L1;
GET.BANANAS: end;

TA-710522-117

Figure 1.6b REF Statement of the Monkey Problem
Solve the following problem:

\[ 3X_1 + 2X_2 \leq 8, \]
\[ X_1 + 4X_2 \leq 10, \]
\[ X_1, X_2 \geq 0, \ X_1, X_2 \text{ integers}, \]
\[ \max Z = 3X_1 + 4X_2. \]

---

Figure I.7a Textbook Statement of an Integer Linear Programming Problem

---

begin;

\text{set } <X1> \text{ to } \text{select(nonnegative.integers)}; \\
\text{set } <X2> \text{ to } \text{select(nonnegative.integers)}; \\
\text{condition } 3*<X1> + 2*<X2> < 9; \\
\text{condition } <X1> + 4*<X2> < 11; \\
\text{maximize } 3*<X1> + 4*<X2>; \\
end;

---

Figure I.7b REF Statement of an Integer Linear Programming Problem
Data Structure

A1
  vector: X,S,E,N,D
A2
  vector: X,M,O,R,E
SUM
  vector: M,O,N,E,Y
L
  vector: D,N,E,S,R,O,M,Y
D  
  <D>: S(1)
N  
  <N>: S(2)
E  
  <E>: S(3)
S  
  <S>: S(4)
R  
  <R>: S(5)
O  
  <O>: S(6)
M  
  <M>: S(7)
Y  
  <Y>: S(8)
CARRY
  <CARRY>: 0

Variables

S(1)
  range: 1 2 3 4 5 6 7 8 9
S(2)
  range: 1 2 3 4 5 6 7 8 9
...
...
S(8)
  range: 1 2 3 4 5 6 7 8 9

Constraints

excl(S(1), S(2), S(3), S(4), S(5), S(6), S(7), S(8))
  S(7) = 0
  S(4) = 0
S(1) + S(3) < 10
S(1) + S(3) = S(8)
S(2) + S(5) < 10
S(2) + S(5) = S(3)
S(3) + S(6) < 10
S(3) + S(6) = S(2)
S(4) + S(7) < 10
S(4) + S(7) = S(6)
S(7) = 0

Figure II.1 Crypt-Addition Context where all Carry Values are 0
Operator for \texttt{begin}

Set as the next applicable operator in the input context the operator for the statement at line 2 of the procedure.

Operator for a \texttt{set} statement at line $k$ of a procedure

Call the \texttt{set} statement interpreter with the statement and the input object. Set as the next applicable operator in each of the resulting objects the operator for the statement at line $k+1$ of the procedure.

Operator for statement \texttt{'set vector $x$ to $y_1,y_2,\ldots,y_n$'} at line $k$ of a procedure

Call the \texttt{set} statement interpreter with the statement \texttt{'set $x[1]$ to $y_1$'} and the input context. For $i = 2, 3, \ldots, n$ call the \texttt{set} statement interpreter with the statement \texttt{'set $x[i]$ to $y_i$'} and each of the objects produced during the interpretation of the $x[i-1]$ \texttt{set} statement. Set as the next applicable operator in each of the resulting objects the operator for the statement at line $k+1$ of the procedure.

Operator for \texttt{condition} statement at line $k$ of a procedure.

Add the Boolean expression of the statement as a constraint to the input object. Set as the next applicable operator in the resulting object the operator for the statement at line $k+1$ of the procedure.

Operator for statement \texttt{'goto $x$'}

Set the operator for statement $x$ as the next applicable operator in the input object.

Operator for statement \texttt{'if $x$ then $y$'} at line $k$ of a procedure.

Add $x$ as a constraint to a copy of the input object and set the operator for statement $y$ as the next applicable operator in the resulting object. Add $\neg(x)$ as a constraint to the input object and set as the next applicable operator in the resulting object the operator for the statement at line $k+1$.

\texttt{TA-710522-120}

\textbf{Figure II.2} Search Space Operators Formed by REF Statements
Operator for statement ‘goto \( y_1, y_2, \ldots, y_n \) x’.

For \( i = 1,2,\ldots,n - 1 \) set ‘\( x = i \)’ as a constraint in a copy of the input object and set the operator for statement \( y_i \) as the next applicable object. Add ‘\( x = n \)’ as a constraint to the input object and set the operator for statement \( y_n \) as the next applicable operator in the resulting object.

Operator for end

Call the constraint satisfier to find a set of variable values which satisfy the constraints in the input object. If the constraint satisfier is successful, then output an object in which the solution values are assigned to the variables. If no solution is found, then produce no output object.

Note: for statements are translated by the ARF loader into set, if, and goto statements as follows:

Given the for loop

\[
\text{for } i \rightarrow \langle N \rangle \text{ goto L1;} \\
\ldots \\
\ldots \\
\]

L1:

ARF’s loader will translate it into the following statements:

\[
\text{set } \langle i \rangle \text{ to } 1; \\
S2: \text{if } i = \langle N \rangle + 1 \text{ then } S1; \\
\ldots \\
\ldots \\
\]

L1:

\[
\text{set } \langle i \rangle \text{ to } \langle i \rangle + 1; \\
\text{goto } S2; \\
S1: \\
\]

where S1 and S2 are internally generated labels.
27 TERMINAL NODES

Figure II.3a Three Levels of the Standard Search Tree for the Monkey Problem

11 TERMINAL NODES

Figure II.3b Three Levels of the ARF Search Tree for the Monkey Problem
Figure III.1 Ordering of Variables for Generation in the Example Case from the Crypt-Addition Problem
Figure III.2 Executive for Constraint Satisfier
Figure III.3 ARF's Expression Evaluation Executive
1. $=+(a_1, \ldots, a_k, i)$ where $i$ is an inte expression, $a_1$ is not an inte expression, and not all of $a_1, \ldots, a_k$ are ‘-' expressions. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
</table>

2. $=(a_1, i)$ where $i$ is an inte expression and $a_1$ is not an inte, ‘+', or ‘-' expression. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 = 3$</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

3. $=(a_1, a_2)$ where neither $a_1$ nor $a_2$ are ‘+', ‘-', sele, or inte expressions. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
</table>

< 1. $<i, +(a_1, \ldots, a_k))$ or $<+(a_1, \ldots, a_k), i)$ where $i$ is an inte expression, $a_1$ is not an inte expression, and not all of $a_1, \ldots, a_k$ are ‘-' expressions. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
</table>

2. $<(i, a_1)$ or $<a_1, i)$ where $i$ is an inte expression and $a_1$ is not an inte, ‘+', or ‘-' expression. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 &lt; 4$</td>
<td>TRUE</td>
</tr>
<tr>
<td>$-5 &lt; B[1]$</td>
<td>$-5 &lt; B[1]$</td>
</tr>
</tbody>
</table>

~ 1. $\sim(a_1)$ where $a_1$ is not a ‘\sim', ‘\langle', ‘\wedge', or ‘\lor' expression and is not symb(TRUE) or symb(FALSE). E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sim(B[1] &lt; 3)$</td>
<td>$2 &lt; B[1]$</td>
</tr>
<tr>
<td>\simTRUE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

Figure III.4 Standard Forms for Simplified Expressions
+ 1. \(+(a_1, \ldots, a_k)\) or \(+i_1, \ldots, a_k\) where \(i\) is an \text{inte}\ expression but not \text{inte}(0)\ and none of \(a_1, \ldots, a_k\) are \text{inte} or \text{'}\ plus\' expressions. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
</table>

- 1. \(-(a_1)\) where \(a_1\) is not a \text{'}\ minus\', \text{'}\ plus\', or \text{inte} expression. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-B[1])</td>
<td>(B[1])</td>
</tr>
<tr>
<td>(-7)</td>
<td>(-7)</td>
</tr>
</tbody>
</table>

\(\wedge\) 1. \(\wedge(a_1, \ldots, a_k)\) where none of \(a_1, \ldots, a_k\) are \text{'}\ and\' expressions. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalents</th>
</tr>
</thead>
</table>

\(\vee\) 1. \(\vee(a_1, \ldots, a_k)\) where none of \(a_1, \ldots, a_k\) are \text{'}\ or\' expressions. E.g.,

<table>
<thead>
<tr>
<th>expressions</th>
<th>standard form equivalent</th>
</tr>
</thead>
</table>

TA-710522-127
Figure III.5 Constraint Processing Executive
<table>
<thead>
<tr>
<th>Identifying Number</th>
<th>Brief Description</th>
<th>Possible Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eliminates TRUE and FALSE.</td>
<td>STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>2</td>
<td>Uses $S(i)=j$ to set $j$ as the value of $S(i)$.</td>
<td>CONTINUE, STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>3</td>
<td>Uses $S(i)&lt;j$ or $j&lt;S(i)$ to delete elements from the range of $S(i)$.</td>
<td>CONTINUE, STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>4</td>
<td>Uses $\neg(S(i)=j)$ to delete $j$ from the range of $S(i)$.</td>
<td>CONTINUE, STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>5</td>
<td>Uses constraint $a_1 \land a_2 \land \ldots \land a_k$ to form new constraints $a_1, a_2, \ldots, a_k$.</td>
<td>STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>6</td>
<td>Uses an equation to determine a value for a vector element, identifier-attribute pair, or variable.</td>
<td>CONTINUE, STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>7</td>
<td>Eliminates variable range elements by generating and testing combinations of variable values.</td>
<td>CONTINUE, STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>8</td>
<td>Uses an excl constraint and action no. 7 to eliminate variable range elements.</td>
<td>CONTINUE, STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>9</td>
<td>Uses a '$&lt;$' constraint to create new constraints by replacing occurrences of $S(i)$ by $\max(S(i))$ or $\min(S(i))$.</td>
<td>CONTINUE, STOP.OK, STOP.FAIL</td>
</tr>
<tr>
<td>10</td>
<td>Uses $x=c$, where $c$ is a symb or inte expression, to replace all occurrences of $x$ in processed constraints by $c$.</td>
<td>STOP.FAIL</td>
</tr>
<tr>
<td>11</td>
<td>Eliminates the redundant member of the constraint pair $i&lt;x$ and $j&lt;x$ (or $x&lt;i$ and $x&lt;j$), where $i$ and $j$ are integers.</td>
<td>STOP.OK, TA-710522-129</td>
</tr>
</tbody>
</table>

Figure III.6 ARF's Constraint Manipulation Action Routines
Figure III.6 Continued

12 Deletes \( \neg(x=y) \) when the constraint pair \( \neg(x=y) \) and \( x<y \) (or \( y<x \)) occurs. 

13 Deletes \( \neg(x=y) \) when the constraint pair \( \neg(x=y) \) and \( \text{excl}(...,x,...,y,...) \) occurs. 

14 Uses constraints \( x \) and \( \neg(x) \) to derive a contradiction. 

15 Uses constraints \( x=y \) and \( \text{excl}(...,x,...,y,...) \) to derive a contradiction. 

16 Uses \( x<j \) and \( i<x \), where \( i \) and \( j \) are integers such that \( j<i+1 \), to deduce a contradiction. 

<table>
<thead>
<tr>
<th>Constraint Operator</th>
<th>Action List</th>
</tr>
</thead>
<tbody>
<tr>
<td>symb</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>1.</td>
</tr>
<tr>
<td>( \neg )</td>
<td>2,6,7,14,15,10.</td>
</tr>
<tr>
<td>( \text{excl} )</td>
<td>3,9,10,16,11,12.</td>
</tr>
<tr>
<td>( \wedge )</td>
<td>4,7,14,10,12,13.</td>
</tr>
<tr>
<td>( \vee )</td>
<td>8,15,10,14,13.</td>
</tr>
<tr>
<td></td>
<td>5.</td>
</tr>
<tr>
<td></td>
<td>7,10.</td>
</tr>
</tbody>
</table>

TA-710522-130
Data Structure

J
\langle \rangle: 2

CARRY
\langle CARRY \rangle: 0

Y
\langle Y \rangle: S(1)+S(3)

M
\langle M \rangle: S(7)

O
\langle O \rangle: S(6)

R
\langle R \rangle: S(5)

S
\langle S \rangle: S(4)

E
\langle E \rangle: S(3)

N
\langle N \rangle: S(2)

D
\langle D \rangle: S(1)

I
\langle I \rangle: 4

L
vector: D,N,E,S,R,O,M,Y

SUM
vector: M,O,N,E,Y

A2
vector: X,M,O,R,E

A7
vector: X,S,E,N,D

Variables

S(8)
value: S(1)+S(3)

S(7)
range: 1 2 3 4 5 6 7 8 9

S(6)
range: 0 1 2 3 4 5 6 7 8 9

S(5)
range: 0 1 2 3 4 5 6 7 8 9

S(4)
range: 1 2 3 4 5 6 7 8 9

S(3)
range: 0 1 2 3 4 5 6 7 8 9

S(2)
range: 0 1 2 3 4 5 6 7 8 9

S(1)
range: 0 1 2 3 4 5 6 7 8 9

Processed Constraints

S(1) + S(3) < 10
excl(S(1),S(2),S(3),S(4),S(5),
S(6),S(7),S(1)+S(3))

Unprocessed Constraints: none

Figure III.7 A Crypt-Addition Context During Interpretation
Begin

CREATE THE CURRENT CONTEXTS LIST WITH THE INITIAL CONTEXT ON IT.

CALL THE EVALUATION FUNCTION TO SELECT A CONTEXT FROM THE CURRENT CONTEXTS LIST.

DELETE THE SELECTED CONTEXT FROM THE CURRENT CONTEXTS LIST.

CALL THE INTERPRETER WITH THE SELECTED CONTEXT.

ADD THE CONTEXTS PRODUCED BY THE INTERPRETER TO THE CURRENT CONTEXTS LIST.

CURRENT CONTEXTS LIST EMPTY?

Solution Found

Yes
No Solution

No

Figure IV.1 ARF's Top Executive
Context 1.2

Data Structure

BOX
  vector:  X2,ON.FLOOR
MONKEY
  vector:  X[S(1)],ON.FLOOR
Y
  vector:  ON.FLOOR,ON.BOX
X
  vector:  X1,X2,UNDER.BANANAS

Variables

S(1)
  range:  1 3

Processed Constraints:  none
Unprocessed Constraints:  none

Context 1.1.1.2

Data Structure

BOX
  vector:  X2,ON.FLOOR
MONKEY
  vector:  X[S(3)],ON.FLOOR
Y
  vector:  ON.FLOOR,ON.BOX
X
  vector:  X1,X2,UNDER.BANANAS

Variables

S(1)
  value:  2
S(2)
  value:  1
S(3)
  range:  1 3

Processed Constraints:  none
Unprocessed Constraints:  none

TA-710622-133

Figure IV.2 Two Nonidentical Equivalent Contexts
Figure IV.3
Control Structure and Distance Measures
for the Monkey Problem Procedure

TA-710522-134
Figure IV.4 ARF's Search Tree for the Monkey Problem
Figure IV.5 Control Structure and Distance Measures for the Crypt-Addition Problem Procedure
Figure IV.6 ARF's Search Tree for the Crypt-Addition Problem
Figure IV.7  Control Structure and Distance Measures for the Water Jug Problem Procedure
1 indicates the branch was found to be redundant and eliminated.

Each six-way branch has the following form:

(x,y) indicates that the 8 gallon jug contains x gallons and the 5 gallon jug contains y gallons.

Figure IV.8 ARF’s Search Tree for the Water Jug Problem
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Sorting Problem</th>
<th>Maxj Speaker Problem</th>
<th>Postfix Problem</th>
<th>Parsing Problem</th>
<th>Instant Insanity Problem</th>
<th>Chinese Traveler Problem</th>
<th>Eight Queens Problem</th>
<th>Graph Coloring Problem</th>
<th>Knot and Maze Problem</th>
<th>Complete Solution Problem</th>
<th>Subgoal Formulation Problem</th>
<th>Sequence Formation Problem</th>
<th>Hypothesis Formation Problem</th>
<th>Marker Problem</th>
<th>Waiting Problem</th>
<th>Programming Problem</th>
<th>Multiview and Constraint Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Total processing time to solution.</td>
<td>0.50</td>
<td>2.19</td>
<td>3.19</td>
<td>4.25</td>
<td>4.50</td>
<td>6.59</td>
<td>8.09</td>
<td>9.37</td>
<td>2.08</td>
<td>2.27</td>
<td>5.32</td>
<td>6.20</td>
<td>0.26</td>
<td>1.23</td>
<td>2.23</td>
<td>6.11</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Statements in REF procedure.</td>
<td>6</td>
<td>10</td>
<td>17</td>
<td>34</td>
<td>50</td>
<td>56</td>
<td>71</td>
<td>85</td>
<td>23</td>
<td>29</td>
<td>60</td>
<td>76</td>
<td>92</td>
<td>102</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>3.</td>
<td>Proportion of processing time spent interpreting.</td>
<td>.38</td>
<td>.73</td>
<td>.19</td>
<td>.43</td>
<td>.24</td>
<td>.71</td>
<td>.03</td>
<td>.59</td>
<td>.62</td>
<td>.61</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>4.</td>
<td>Actual time spent interpreting.</td>
<td>0.19</td>
<td>0.38</td>
<td>0.53</td>
<td>1.10</td>
<td>4.58</td>
<td>0.16</td>
<td>5.34</td>
<td>1.19</td>
<td>1.35</td>
<td>5.32</td>
<td>6.20</td>
<td>0.29</td>
<td>1.23</td>
<td>2.23</td>
<td>6.11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Total statements interpreted.</td>
<td>33</td>
<td>10</td>
<td>51</td>
<td>82</td>
<td>63</td>
<td>347</td>
<td>4</td>
<td>54</td>
<td>81</td>
<td>76</td>
<td>587</td>
<td>508</td>
<td>23</td>
<td>111</td>
<td>421</td>
<td>718</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Contexts processed during interpretation.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>150</td>
<td>8</td>
<td>41</td>
<td>69</td>
<td>124</td>
<td>15</td>
<td>56</td>
<td>15</td>
<td>125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Contexts eliminated by constraint processing during interpretation.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>45</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Nonterminal nodes in the interpretation search tree.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>21</td>
<td>62</td>
<td>16</td>
<td>63</td>
<td>15</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Terminal nodes in the interpretation search tree.</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>13</td>
<td>29</td>
<td>12</td>
<td>54</td>
<td>5</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>Total nodes in the interpretation search tree.</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>12</td>
<td>54</td>
<td>91</td>
<td>28</td>
<td>117</td>
<td>20</td>
<td>126</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>Statements on the solution interpretation path.</td>
<td>33</td>
<td>10</td>
<td>51</td>
<td>63</td>
<td>63</td>
<td>347</td>
<td>4</td>
<td>54</td>
<td>43</td>
<td>58</td>
<td>209</td>
<td>166</td>
<td>13</td>
<td>18</td>
<td>178</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>Nodes on the solution interpretation path.</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>14</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>Contexts eliminated as redundant during interpretation.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>Average selectivity provided by distance measure.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>1.3</td>
<td>1.9</td>
<td>1.5</td>
<td>3.6</td>
<td>2.5</td>
<td>2.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>Average selectivity provided by distance measure and variable count.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>1.3</td>
<td>1.9</td>
<td>1.3</td>
<td>2.5</td>
<td>1.8</td>
<td>2.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>Average number of contexts from which selection was made.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>3.2</td>
<td>3.6</td>
<td>3.6</td>
<td>3.1</td>
<td>8.3</td>
<td>4.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>Proportion of time spent in backtrack search.</td>
<td>.02</td>
<td>.07</td>
<td>.91</td>
<td>.57</td>
<td>.69</td>
<td>.29</td>
<td>.97</td>
<td>.41</td>
<td>.26</td>
<td>.39</td>
<td>.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td>Actual time spent in backtrack search.</td>
<td>0.31</td>
<td>2.00</td>
<td>2.41</td>
<td>2.22</td>
<td>2.40</td>
<td>2.01</td>
<td>5.22</td>
<td>4.02</td>
<td>1.49</td>
<td>0.32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.</td>
<td>Number of backtrack searches.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.</td>
<td>Steps in the backtrack search which produced the solution.</td>
<td>13</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>28</td>
<td>11</td>
<td>44</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.</td>
<td>Constraints in solution context at end of interpretation.</td>
<td>4</td>
<td>9</td>
<td>21</td>
<td>18</td>
<td>12</td>
<td>45</td>
<td>3</td>
<td>22</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.</td>
<td>Contexts processed during backtrack search.</td>
<td>15</td>
<td>22</td>
<td>46</td>
<td>32</td>
<td>93</td>
<td>45</td>
<td>137</td>
<td>35</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23.</td>
<td>Size of solution space at beginning of backtrack search.</td>
<td>3.1*</td>
<td>103</td>
<td>96</td>
<td>24</td>
<td>1.5*</td>
<td>105</td>
<td>126</td>
<td>2.5*</td>
<td>101</td>
<td>1.2*</td>
<td>103</td>
<td>1.7*</td>
<td>107</td>
<td>5.1*</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.</td>
<td>Size of solution space at end of interpretation.</td>
<td>3.1*</td>
<td>103</td>
<td>96</td>
<td>24</td>
<td>1.5*</td>
<td>105</td>
<td>126</td>
<td>2.5*</td>
<td>101</td>
<td>1.2*</td>
<td>103</td>
<td>1.7*</td>
<td>107</td>
<td>5.1*</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25.</td>
<td>Values in solution context's data structure.</td>
<td>12</td>
<td>9</td>
<td>10</td>
<td>39</td>
<td>62</td>
<td>111</td>
<td>8</td>
<td>51</td>
<td>26</td>
<td>24</td>
<td>15</td>
<td>55</td>
<td>9</td>
<td>2</td>
<td>15</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>26.</td>
<td>Variables in solution context.</td>
<td>5</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>8</td>
<td>18</td>
<td>8</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**Table V.1** Summary of ARE's Behavior on 16 Sample Problems