DOCUMENT RESUME

ED 072 817 LI 004 170

AUTHOR	Minker, Jack; And Others
TITLE	The Maryland Refutation Proof Procedure.
INSTITUTION	Maryland Univ., College Park. Computer Science Center.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	TR-208
PUB DATE	Dec 72
NOTE	94p.;(46 References)
EDRS PRICE	MF-$0.65 HC-$3.29
DESCRIPTORS	Algorithms; *Computer Programs; *Deductive Methods; *Experimental Programs; *Information Systems; Man Machine Systems; *Search Strategies
IDENTIFIERS	*Maryland Refutation Proof Procedure System; MRPPS; Question Answering Systems; University of Maryland

ABSTRACT

The Maryland Refutation Proof Procedure System (MRPPS) is an interactive experimental system intended for studying deductive search methods. Although the work is oriented towards question-answering, MRPPS provides a general problem solving capability. There are three major components within MRPPS. These are: (1) an inference system, (2) a search strategy and (3) a base clause selection strategy. The "inference system" is based on the resolution principle and performs the logical deductions specified. The user may select from a wide variety of refinements of resolution. The "search strategy" directs the deductions to be made by selecting from clauses already generated those that have a best merit. The "base clause selection strategy" determines which facts and general axioms to select from the data base. Such a clause may be selected regardless of whether or not it has the best merit. Heuristic techniques are applied within each of the three major components. This technical report describes the current implementation of MRPPS. It describes each of the components and how they are integrated into what has been termed the "Q" algorithm. (Author/NH)

Technical Report TR-208 December 1972
GJ-31456

THE MARYLAND REFUTATION PROOF PROCEDURE SYSTEM (MRPPS)

Jack Minker
Daniel H. Fishman
James R. McSkimin

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER
COLLEGE PARK, MARYLAND

ABSTRACT

The Maryland Refutation Proof Procedure Systems (MRPPS) is an interacti[illegible] [illegible]perimental system intended for studying deductive search methods. Although the work is oriented towards question - answering, MRPPS provides a general problem solving capability.

There are three major components within MRPPS. These are:

(1) an inference system,

(2) a search strategy, and

(3) a base clause selection strategy.

The *inference system* is based on the resolution principle and performs the logical deductions specified. The user may select from a wide variety of refinements of resolution. Current refinements are: set of support, linear, P1, SL, input, and combinations of the above. Paramodulation and deletion by tautologies and subsumption are also provided with the system.

The *search strategy* directs the deductions to be made by selecting from clauses already generated those that have the best merit. The merit of a clause is given by $f(n) = w_0 g(n) + w_1 h_1(n) + w_2 h_2(n) + \cdots + w_k h_k(n)$. If the user can specify tie-breaking rules for equal values of clause merit, an upwards diagonal search results in the sense of Kowalski. The upwards diagonal search included in MRPPS generalizes the Kowalski upwards diagonal search to an n-dimensional search.

The *base clause selection strategy* determines which facts and general axioms to select from the data base. Such a clause may be selected regardless of whether or not it has the best merit.

Heuristic techniques are applied within each of the three major components. This technical report describes the current implementation of MRPPS. It describes each of the components and how they are integrated

into what has been termed the Q* algorithm.

MRPPS is written in FORTRAN V for the UNIVAC 1108 (a version of FORTRAN IV) and runs under EXEC 8 at the University of Maryland. The current implementation is core bound and requires approximately 60K words of memory to run, of which 35K is for the data base and for working storage.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The current implementation of the system described in this report is the result of work by the authors as well as others including G. A. Wilson and J. G. Augustson and various students in the senior author's classes at the University of Maryland. The authors also appreciate the financial support provided to the research effort by the National Science Foundation under grant NSF GJ-31456.

1. Introduction

In this interim report we describe research being performed at the University of Maryland in the area of Question-Answering (QA) Systems. The major goal of the research is to attempt to provide some insight into the problem of performing deductive searches in a QA System. We therefore do not emphasize the natural language aspects of question-answering. The approach we have taken is to design and implement a QA System which provides the flexibility to experiment with various search heuristics as well as with different inference systems. The purpose of this technical report is to describe such a system termed the Maryland Refutation Proof Procedure Systems (MRPPS).

The deductive capability of MRPPS is provided by the resolution principle as developed by Robinson [1965a]. Many refinements of resolution are provided within MRPPS as described in this report. MRPPS consists of not only inference mechanisms but also a search strategy that directs the search for an answer to a query. Such a system has been termed a *proof procedure* system by Kowalski [1970b]. A *refutation proof procedure* is a term which applies to proof procedures in which the proof consists of a refutation of the negation of a statement to be proved. A refutation proof procedure system may therefore be defined as a system $P(I,\Sigma)$ that consists of two basic parts:

(1) a set of *inference rules* I based upon the resolution principle and

(2) a *search strategy* Σ for I.

The inference rules restrict the application of resolution to particular subsets of S , the set of clauses that could potentially enter into the proof. Examples of these rules are set-of-support and P1-deduction (as well as unrestricted binary resolution with factoring). On the other hand, the search strategy determines which two clauses from S should be chosen to resolve next as determined by some cost function f applied to each clause. Whether or not the two clauses chosen will actually be resolved is determined by the inference system in force, which "filters out" non-permissible inferences. We have also added to MRPPS a third major component in addition to the inference rules and search strategy:

(3) a selection strategy for determining the sequence and timing in which data entries and general rules are to be brought to bear on the problem.

The second of the above system components will be called the *deduction strategy* while the third component will be called the *base clause selection strategy*. These two components, together, comprise a search algorithm which we call the *Q* search algorithm*. The design of this algorithm has been strongly influenced by the $\sum^*$ algorithm of Kowalski [1970b]. The deduction strategy builds upon and extends analogous parts of the $\sum^*$ algorithm to permit additional heuristic considerations to be employed. The base clause selection strategy, which has no analog in the $\sum^*$ algorithm, permits the selective use of data and axioms, and also permits heuristic and semantic considerations to be brought to bear on the problem. Use of the selection strategy leads to a rapid and efficient derivation of a refutation in many instances.

The inference component of the system is based on the first-order predicate calculus, and in particular, on the Robinson resolution principle (Robinson [1965a]) and refinements thereof. This component permits the selection of one of several inference systems, or of combinations of these inference systems, for any given run of MRPPS.

Details of each of these three main system components are described in this report. The major innovations incorporated in MRPPS are listed below:

(1) MRPPS utilizes a refutation proof procedure that includes both heuristic search and logical deduction. In the past, most systems have used ad hoc approaches in either or both of these processes whereas MRPPS integrates and coordinates the two processes based on Kowalski's work.

(2) The concept of upwards diagonal search has been extended to the case where the heuristic component is a linear combination of an arbitrary number of heuristics.

(3) The concept of merit sets developed by Kowalski has been modified. Whereas Kowalski primarily treated the case of clause length and level as heuristic measures, MRPPS allows various additional heuristics to be defined.

(4) The concept of using cluster analysis as a basis for deriving heuristic measures has been introduced.

(5) A selection strategy is used for introducing base clauses into the search space. This strategy delays clauses from being used to generate inferences until it is essential that the clause be brought in. The strategy is such that base clauses are not given merit unless they are determined to be relevant to the search in progress.

Work on the effort began approximately in February 1972. The above three parts of the system were designed and implemented between the starting date and August 1972. The system is written in FORTRAN V for the UNIVAC 1108 (a version of FORTRAN IV) and runs under EXEC 8 at the University of Maryland[1]. The system can be used from teletype and has interactive capabilities that permit the user to select a wide range of options to run his problem. Primitive capabilities exist to enter, update, and maintain new data bases. The routines to handle the data base are described in an appendix to this report[2]. The entire system is core bound and requires approximately 60K words of memory to run, of which 25K is for programs and 35K is for data storage, including the entire data base and working storage.

A data base is currently being used that consists of genealogical information about Eskimos. It is expected that experimentation with this data base and others will be performed to gain insight into inference mechanisms, heuristics, and semantics that might be useful for QA systems.

---

(1) FORTRAN was chosen primarily because of availability and a high degree of maintenance at the computer center.

(2) This appendix is not included with this report, but may be obtained upon request from the authors.

2. Background

A QA System has been characterized by Kuno [1967] as consisting of:

(1) a source language input;

(2) a syntactic analyzer;

(3) a semantic analyzer;

(4) an inference and search mechanism; and,

(5) an output language.

There have been a large number of research efforts devoted towards QA Systems as evidenced by state-of-the-art summaries by Simmons [1965, 1970], Minker and Sable [1970], Montgomery [1969, 1972], Salton [1968], Bobrow, Fraser, and Quillian [1967].

The developments reported upon in the above survey articles have primarily emphasized the natural language processing portion represented by the source language, syntactic analysis, and semantic analysis. The major purpose of this effort is to investigate the inference and search mechanisms of such systems. If an inference mechanism is important to an application area, one can always simplify the source language input by placing the burden of the work on the user and hence, avoid machine problems in syntactic and semantic analysis. However, if the inference mechanism is weak in its deductive capabilities, or cumbersome, a QA System may not be a viable tool.

A number of QA Systems have been designed that contain a deductive capability. There have been three major areas of development which may be termed

(1) the predicate calculus approach;

(2) the relational system approach; and,

(3) the procedural language approach.

In the predicate calculus approach data is represented by instantiated predicate expressions such as I [illegible] ([illegible], Sara) and the rules used to define predicates and their interrelationships are in the form of general axioms such as $\mathrm{FATHER}(x, y) \wedge \mathrm{FATHER}(y, z) \Rightarrow \mathrm{GRANDFATHER}(x, z)$. Pioneering work in this approach was first performed by Darlington [1962] who applied results by Davis and Putnam [1960] to perform deductive searches. Green and Raphael [1968] were the first to employ the Robinson Resolution Principle [1965a] in QA Systems. Darlington [1969] continued his efforts in QA Systems and has speculated that an inference system based on A-ordering, P1-deduction and renaming, and set-of-support should be used for QA Systems. Darlington [1971] is continuing his work and is developing a system based on second order logic. Coles [1969] has performed some studies on QA Systems, but his results are inconclusive. Thus, the work has emphasized the "logical" aspects of inference making in contrast to the heuristic aspects, and little experimentation has been performed in each case.

In the relational data system approach, a set of subroutines exist that may be linked together to derive other relations. The subroutines handle some general cases such as relational "composition". Two relations may be composed when the relation $P_1(x,y)$ and the relation $P_2(y,z)$ can yield a third relation $P_3(x,z)$. A number of QA Systems have been designed using such general principles. A system termed *Relational Store Structure* (RSS), developed by Marill [1967], contains some general rules of this type and can, for n-ary relations, perform deductive searches. A breadth-first search is employed to derive inferences. Another system with a similar approach, but which only handles binary relations was developed by Ash and Sibley [1968]. Feldman and Kovner [1968] have also developed a system for performing inferences with binary relations.

In the procedural language approach, a language is available in which the user can write procedural statements that permit deductive searches. A system designed by Levien and Maron [1965] provides a procedural language, INFEREX, for specifying queries. Hewitt [1970] has developed PLANNER, a procedural language for problem solving PLANNER contains an automatic backtracking mechanism. QA4, a system similar in design to PLANNER has been developed by Rulifson [1971]. McDermott and Sussman [1972] have developed a procedural language called CONNIVER that is similar to PLANNER, but requires no backtracking. Also, Feldman et al [1972] have developed SAIL which is an extension of the LEAP language and has capabilities similar to CONNIVER.

Except for the recent developments of PLANNER, QA4, CONNIVER, and SAIL, the approaches taken to date have not used heuristic techniques extensively to enhance the search for a solution and decrease the search time. There has, in general, been little work in applying heuristic techniques to theorem proving or to QA Systems in general. Norton [1972] has performed some experiments with a theorem prover that includes paramodulation. A major part of this study is the development of techniques to permit heuristics to be added to deductive searches so as to help guide the search. We are not convinced that the simple addition of heuristics is sufficient. Extensive semantic information will also have to be used.

In his thesis, Kowalski [1970a] developed the concepts of a refutation proof procedure system and of a search strategy that can be employed in a theorem proving environment. The work by Kowalski generalizes the results developed by Hart, Nilsson and Raphael [1968] for state-space problems to theorem proving situations. Assuming that meaningful heuristics can be developed, Kowalski shows how to apply the results to theorem proving and has placed the results on a firm theoretical basis. Meltzer [1971] provides a very lucid elaboration of Kowalski's work.

Initial work convinced us that a proof procedure system alone will not be adequate for QA Systems. Some strategies must be used to decrease the number of clauses that are to be passed to the QA System. Semantic information about the particular domain, and the general rules used to deduce new results must be brought to bear on the problem. This report describes some of our considerations to date. As more experience is gained, we expect that additional semantic considerations will be developed. Indeed, at the time of this report, we are investigating other considerations. As this report was being written, we learned of independent work by Travis, Kellogg, and Klahr [1972] who, although taking a somewhat different approach than described in this report, have similar, if not identical ideas in this particular area. The work by Travis et al does not use theorem-proving techniques. MRPPS is similar to the work by Allen and Luckham [1970] who have developed an interactive theorem-proving system.

3. Current Search Strategies for Theorem Proving

3.1 Ordered Search Algorithms

In order to more clearly understand the operation of the Maryland Refutation Proof Procedure System described in the following sections, the basic concepts underlying current search strategies that may be used for theorem proving will be discussed. Many of these searching algorithms have defined a costing function f that is used to evaluate the relative merit (i.e., cost) of all those nodes of the search space that are available for expansion. At each step of the algorithm, the node with the smallest cost is expanded next and its successors are placed back on a list containing all unexpanded nodes. In addition, each time a node is expanded, it is removed from this list. The nodes on this list are then reordered with respect to f, the node with the smallest f value is chosen for expansion, and the process is repeated until a solution is found (or the list is empty). This is essentially the procedure used in the A* algorithm of Hart, Nilsson, and Raphael [1968].

It has been customary to define the above merit function as $f(n) = g(n) + h(n)$ where $g(n)$ is an estimate of the actual cost of a minimal cost path from a start node of the search space to the node n, and $h(n)$ is a heuristic estimate of the actual cost from node n to a solution node. For the problem domain of theorem proving using resolution, the following analogies may be established between a state-space search (as in A*) and the search for a null clause:

(1)	states (nodes)	$\leftrightarrow$	clauses
(2)	starting states	$\leftrightarrow$	data base clauses, clauses from the general axioms and clauses from the negation of the question

			or theorem
(3)	operators	$\leftrightarrow$	binary resolution and factoring
(4)	goal state	$\leftrightarrow$	null clause ($\Box$)

When using an ordered search algorithm for theorem proving such as that described above, it is necessary to order all the clauses in the data base in the sequence of best merit first, to avoid having to search the entire data base for the clause with best merit. This means that in setting up a data base, one must specify the merit for all the data base entries and axioms and then sequence them in merit order. Whenever the user wishes to alter the definition of $f(n)$, the data base must be reordered again. A great deal of work is therefore required at the time the set S is defined to the system. In addition to the large amount of work entailed in evaluating $f(n)$ for each clause in advance, the above method has a further disadvantage of requiring the value of the heuristic function $h(n)$ to be fixed in advance of a proof. What would be more desirable would be to avoid calculating $f(n)$ prior to a proof and to permit $h(n)$ to be based upon the query inputted to the system at *run-time*. We shall see that this will be possible to achieve.

There are two more limitations of the A* algorithm when it is used for theorem proving. First, it must be modified to allow the application of successor-generating operators requiring more than one input (e.g., resolution). This means that all of the successors for a clause cannot be formed at one time since that clause may later resolve with another clause that has not even been generated yet. Second, at any stage of a proof, every clause in the entire search space is potentially available for interaction with the clause being expanded (by resolution or paramodulation). Of course various inference systems could restrict the number of interactions, but experience with many theorem provers has shown that in general, space would still be exhausted before a solution would be found. What is needed is some technique for being more selective about which clauses are allowed to interact so that inferences are generated in a more optimal order than in A*. For instance, whereas the A* algorithm would find *all* successors for a clause C at once, an improved strategy might expand a successor of C *before* finding the rest of C's successors, if it decided that the course of action might be more productive than the latter. The Σ^* algorithm of Kowalski [1970b], described in the next section, attempts to alleviate these last two problems.

3.2 Improving the Ordered Search Algorithm - The Σ^* Algorithm

The Σ^* algorithm employs a cost function $f(n) = g(n) + h(n)$ to measure the merit of a clause $C(n)$. In the ensuing discussion, $g(n)$ will be defined as the level of the clause $C(n)$, and $h(n)$ as its length. Kowalski defines both a diagonal search strategy using a diagonal merit ordering $\leq_d$ and also an upwards diagonal strategy using an upwards diagonal merit $\leq_d u$. Let n and n' be two nodes of our search space. Then

(1) $n \leq_d n'$ (n has better or equal diagonal merit than n') iff $f(n) \leq f(n')$ (3.1)

(2) $n \leq_d u\ n'$ (n has better or equal upwards diagonal merit than n') iff $f(n) \leq f(n')$ and $h(n) \leq h(n')$ whenever $f(n) = f(n')$.

It should be noted that (1) defines the diagonal search which is very similar to the A* algorithm mentioned previously. For an upwards diagonal search, if two clauses $C(n)$ and $C(n')$ have equal f values, we then expand that clause which is the shorter of the two since this action has the greatest possibility of producing shorter clauses (perhaps the null clause).

As clauses are generated during the search (by inclusion from the base set S, by resolution, or by factoring), they are placed in disjoint sets called A sets (actually implemented as lists). Thus a clause $C(n)$ is stored in set $A(i,j)$ if it has level j and length i. This is different than the A* strategy where clauses are essentially separated into disjoint subsets according to f value rather than by both g and h values separately.

$\sum^*$ attempts to generate clauses approximately in upwards diagonal merit order. That is, the strategy would try to generate clauses for set $A(i,j)$ before those in $A(i',j')$ if:

(1) $i + j < i' + j'$ or (3.2)

(2) $i + j = i' + j'$ and $i < i'$

If $\sum^*$ is generating clauses for the set $A(i,j)$ it may do so in the following ways:

(1) if $j = 0$, then a clause from the data base of length i may be placed in $A(i,j)$;

(2) if $j > 0$, a clause in $A(i + 1, j - 1)$ may be factored; or

(3) if $j > 0$, a clause $C(n_1)$ in $A(i_1,j_1)$ may be resolved against $C(n_2)$ in $A(i_2,j_2)$ where

$i = i_1 + i_2 - 2$

$j = \max(j_1,j_2) + 1$.

Note that at the start of a proof, all of the A-sets are *empty* (unlike A*). They are filled by a routine called FILL(i,j) that generates either by (1) or (3) above, clauses of merit (i,j). Note that during FILL(i,j) , all parent clauses of resolvents formed by (3) are clauses previously generated and placed in A-sets of *better* merit than (i,j) . When no more clauses can be formed in this manner, FILL(i',j') is called where (i',j') is the "next" merit after (i,j) with respect to the ordering $\leq_d u$. FILL(0,0) is called when $\sum^*$ begins.

A second subroutine, RECURSE(C(n)) , is called whenever FILL(i,j) generates a clause C(n) . Because C(n) is a newly formed clause, it may very well interact with other clauses previously generated to produce a new inference. Thus, RECURSE(C(n)) generates by (2) or (3) above, all clauses of merit $(i',j') <_d u\ (i,j)$ that are immediate descendents of C(n) . Upon generation of the successor C(n') , RECURSE(C(n')) is called to form immediate successors of clause C(n') . This recursive process continues until some level of RECURSE(C(n_i)) fails to generate any clauses meeting the above merit constraints. (Note that this does not mean that C(n_i) has no successors at all.) Control is then returned to the previous level of RECURSE which continues to find the "next" successor for the clause C(n_{i-1}) local to its level of recursion. Eventually FILL is re-entered, and the FILL-ing and RECURSE-ing process continues until the null clause is found (or time or space bounds are exceeded).

Although the $\sum^*$ algorithm is theoretically very elegant, it is bound to fail for any practical application where the data base is fairly large (several hundred clauses perhaps). This is because typical question-answering data bases consist primarily of unit clauses, (i.e., clauses of merit (1,0)). Thus, when FILL(1,0) is called, *all* of these units will be placed in A(1,0) before FILL(0,2) is called. As in the A* algorithm, we have again deluged our theorem prover with data clauses, most of which are *totally irrelevant* to the query.

3.3 An Example Using the Σ^* Algorithm

As an exmaple of the kind of trouble one can get into, let the data base S consist of $|S|$ clauses, $|S| - 1$ of which are units. Among the units are the clauses F(Jack, Sally) and F(Harry, Jack) where F(x,y) may be interpreted as "x is the father of y". The single non-unit (in clause form) is $-F(x,y) \vee -F(y,z) \vee G(x,z)$ which states: "if x is the father of y and y is the father of z, then x is the grandfather of z."

Now suppose that the negation of the query is $-(\exists x)\ G(x, \text{Sally})$. In clause form this is $-G(x, \text{Sally})$. We assume that there is no data base clause of the form G(A, Sally) for some constant A, such that an immediate contradiction would be possible. Also assume that in the implementation of Σ^*, all supported clauses are stored separately from the non-supported clauses of the same merit so that no explicit tests need be made to determine whether a clause has support or not.

When FILL(1,0) is called, each unit base clause is brought in one by one. As each clause C is generated, RECURSE is called with C as its argument. Since C cannot be factored (we assume a fully factored data base), C is resolved against all clauses in A(1,0) *that have already been generated and that have support.* Since only $-G(x, \text{Sally})$ has support, no resolvents are formed until this clause is brought in. When it is, it is resolved against every clause in A(1,0). We fail in each case since no direct contradiction exists. We thus continue to fill A(1,0) until we have brought in all $|S|$ unit clauses and have attempted $|S|$ resolution operations. Note that this figure includes the theorem that is also of merit (1,0). No more clauses are generated at all until FILL(3,0) is called and Σ^* then brings in the general axiom $-F(x,y) \vee -F(y,z) \vee G(x,z)$. RECURSE is called and $-F(x, \text{Sally})$

immediately resolves with the axiom to yield -F(x,y) ∨ -F(y, Sally) of merit (2,1). Thus, $|S|$ + 1 resolution operations have been attempted so far.

Now when RECURSE is called, -F(Sally, Sally) is formed as a factor of merit (1,2). Let us assume that no other clause successfully resolves with -F(Sally, Sally) when RECURSE is called. However, tests are made against all clauses in A(1,0) as well as -F(Sally, Sally) itself. This is allowed since a resolvent of $c_1 \in A(1,0)$ and $c_2 \in A($ $1,2)$ has merit (0,3) $\leq_d u$ (3,0); the same applies to A(1,2) . Thus $|S|$ + 1 tests are made. Next, all unit base clauses are resolved against the clause in A(2,1) . This is permissible since a clause in A(1,0) resolves with a clause of A(2,1) to give a clause of merit (1,2) $\leq_d u$ (3,0). Recall that we are filling A(3,0) . In the worst case F(Harry, Jack) and F(Jack, Sally) might be the next to last and last clauses on the A(1,0) list. Thus $|S|$ - 1 resolutions would be attempted before -F(Jack, Sally) would be placed in A(1,2) . When this clause is RECURSE-d on, another $|S|$ steps are taken before □ is found and placed in A(0,3) . Thus a total of 4$|S|$ + 1 resolution steps were needed to answer a very simple question! These figures ignore the possibility that -F(x,y) ∨ -F(y, Sally) might have resolved with other units before resolving with F(Harry, Jack) . Thus for large $|S|$, a proof might never have been found. Thus it seems that a search strategy is not practical for question-answering systems if it uses only length and level as components of f and employs no strategy for selecting semantically appropriate clauses from the data base.

4. Maryland Refutation Proof Procedure System (MRPPS)

4.1 Introduction to MRPPS

4.1.1 Overview of the System

The development of a new refutation proof procedure system has been motivated by the limitations of search strategies such as $\sum^*$ and A* , most of which have been mentioned in the previous section. These limitations are as follows:

(1) the data base must be reordered whenever the definition of $f(n)$ is altered so that the clause of best merit may be easily located;

(2) the merit of base clauses must be calculated prior to a proof;

(3) clauses are selected from the data base without reference to any semantic information; and,

(4) the merit function $f(n) = g(n) + h(n)$ is composed of only two parameters where $g(n)$ and $h(n)$ may be composite functions but are considered as single values.

It should be clear that the first three limitations must be corrected in order to successfully answer questions about a large data base. However, at the present time, it is not known whether the last point is really a limitation since composite g and h functions may permit as much discrimination between clauses as does an evaluation function f with several separate components. This question should be answered through further experimentation.

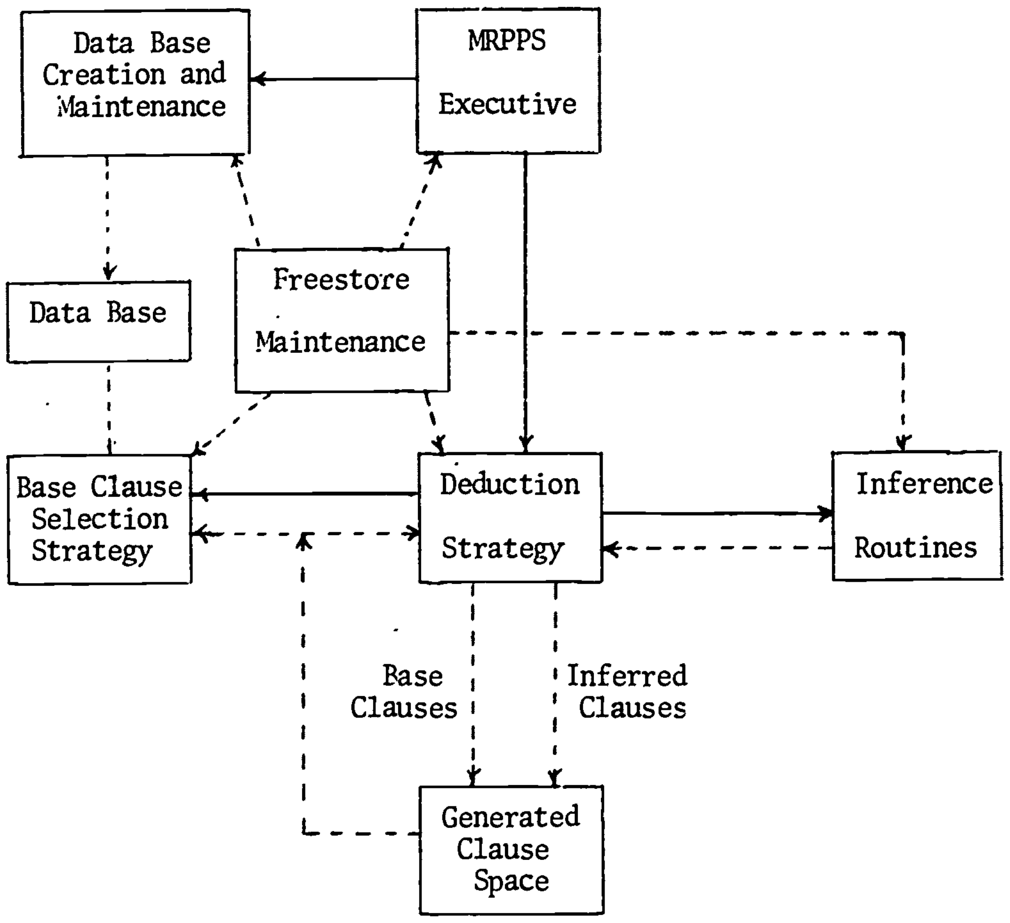
As described in the introduction, MRPPS consists of three main components: a set of inference rules, a deduction strategy, and a base clause selection strategy. The last two together comprise the Q* search algorithm that controls the search for a refutation. Figure 1 gives an overview of the system and shows how the three main components are integrated into the system. Note that solid lines represent control paths whereas dotted lines represent data paths between components of the system. MRPPS is controlled by an executive that communicates with the user as well as with the deduction strategy routines and data base creation and maintenance routines. There is also a freestore maintenance routine that handles the allocation and return of core storage.

The data base of MRPPS may be separated into two different data structures at the user's option. The first form indexes clauses by predicate sign, then by each predicate name of the clause, and within equal predicate names, by clause length. The second form has one additional level of indexing; namely, within predicate names, clauses are indexed by term names, and further indexed by clause length if the term names are identical. Thus, a greater degree of discrimination is permitted in the second form than the first. It is advisable that the user enter an axiom into the first structure only if it contains no functions or constants. In this way, the base clause selection strategy will select clauses in as optimal a fashion as possible.

There are several implementation details that should be mentioned here. As noted previously, MRPPS is implemented on the UNIVAC 1108. Since addresses on the 1108 are 16 bits long, only 65K words of memory may be accessed directly. However, up to 262K words may be referenced by indirect addressing. Since 65K is not a realistic limitation for an experimental system, all data referencing is therefore done indirectly, even though a loss of speed is suffered.

In addition, the current version of the system is core-based. For large data bases, however, this is out of the question, and thus it is planned that future versions of MRPPS will have the capability of accessing data on peripheral devices rather than only in core.

MRPPS has been designed to be a flexible as well as a computationally efficient system. We have attempted to provide a wide variety of inference mechanisms as well as various heuristic measures for use during the search for refutations. Details of the inference routines will be discussed in section 4.1.2, the deduction strategy will be described in section 4.5 and the base clause selection strategy will be described in section 4.4. Where appropriate, references are given for the various techniques that have been used in MRPPS.



Note: dotted lines indicate data paths

solid lines indicate control paths

Figure 1. System Overview of MRPPS

4.1.2 Inference Systems Implemented

MRPPS allows the user considerable flexibility in choosing which inference mechanisms (or combinations of mechanisms) to use. The following inference mechanisms are currently implemented. With each inference mechanism we provide a reference to the relevant literature.

(1) Binary Resolution and Factoring (Robinson [1965a])

Unrestricted resolution allows resolution between any two clauses $C_1 \in S$ and $C_2 \in S$ on two literals, $\ell_1 \in C_1$ and $\ell_2 \in C_2$, where S is the set of clauses available for consideration by resolution at some stage of a se^rch. S contains all factors and resolvents generated during the search and those base clauses made available by the base clause selection strategy.

(2) Set-of-Support (Wos, et al [1965])

In this inference system, the set S is subdivided into two subsets K and $S-K$ such that $S-K$ is satisfiable. Clauses in K are said to have support and thus initially, K consists of all clauses from the negation of the theorem. A resolvent $R(C_1,C_2)$ is permitted if and only if $C_1 \in K$ and $C_2 \in S$, and each resolvent is placed in K when it is formed. Set-of-support is important since it restricts the set of different potential proofs.

(3) P1-Deduction (Robinson [1965b], Meltzer [1966])

A resolvent $R(C_1,C_2)$ is permitted if and only if either C_1 or C_2 is a positive clause.

(4) Linear Resolution [Luckham [1968], Loveland [1968])

A linear derivation from a set of base clauses S is a sequence of clauses $C_1,C_2,\ldots,C_n$ such that $C_1 \in S$ and each

C_{i+1} is a resolvent of C_i and B where either

a) B is a base clause

b) B is an ancestor of C_i .

(5) Input Resolution (Chang [1970])

This is the same as linear resolution except that B may only be a base clause. Input resolution is not complete.

(6) Linear Resolution With Selection Function (Kowalski and Kuehner [1972])

SL-resolution is a refinement of linear resolution and is very similar to model elimination (Loveland [1969]). The fundamental difference between linear and SL-resolution is that in the latter, a single literal is selected from each clause C_i in the linear derivation. When input resolution is performed on the clause C_i , only the selected literal may be resolved upon, whereas in linear resolution, any literal could be resolved upon. This has the effect of eliminating redundant proofs.

All of the first six inference systems, with the exception of input resolution are complete and sound.

(7) Combinations

MRPPS allows many combinations of the above inference systems to be used simultaneously, although in some cases, the resulting inference system may not be complete. For instance if set-of-support and P1-deduction were used concurrently, we could guarantee completeness only if the conjectured theorem together with all positive clauses had support. The user must determine for himself whether the combination that he is using is complete. The system does not automatically prevent incomplete combinations of systems from being used.

(8) Paramodulation (Robinson and Wos [1969])

Paramodulation is a substitution rule that infers new clauses by substituting a term t_1 for a term t_2 in a clause C such that t_1 equals t_2. The resultant clause C' is said to be inferred by paramodulation. More formally, suppose that there exists a predicate $E(t_1,t_2)$ that expresses the equality of terms of t_1 and t_2. Let A and $E(t_1,t_2) \vee B$ be two clauses whose variables have been standardized apart. Here B is the remainder of the clause containing $E(t_1,t_2)$. Suppose that a term t in A and t_1 have a most general common instance:

$$t_1\sigma = t\sigma .$$

Let A' be the result of replacing in $A\sigma$ some *single occurrence* of $t\sigma$ by $t_2\sigma$. Then the clause $A' \vee B\sigma$ is inferred by paramodulation.

Paramodulation is important since it circumvents the problem resulting when equality between terms is handled by resolution. It is available in MRPPS as an option to the user whereas resolution and factoring are always performed. It has been shown by Robinson and Wos [1969] that paramodulation is complete when used in a functionally reflexive system; i.e., for all functions f, $x_i = x_j$ implies $f(x_i,\ldots) = f(x_j,\ldots)$.

4.1.3 Deletion Rules Used in MRPPS

Whenever a clause is generated by any means during a proof, various checks may be made to determine if the clause is redundant. A clause is considered redundant if it has already been formed before, or if its presence will contribute nothing to the search process. If any of these conditions are detected, the clause may be deleted (if so desired by the user).

MRPPS employs three optional deletion rules: deletion of tautologies, of alphabetic variants, and of subsumed clauses. A set S of clauses (namely all the clauses in our generated search space) is unsatisfiable irrespective of whether or not tautologies are deleted. If they were left in the search space, however, they would only tend to generate irrelevant inferences. Since detection of tautologous clauses is a reasonably efficient process, it is therefore advantageous to always perform tautology elimination. Also, completeness is not lost for any of the above systems.

We may define subsumption as follows: a clause C_1 subsumes a clause C_2 if $C_1\sigma \subseteq C_2$ for some substitution σ. It has been shown that the unsatisfiability of a set of clauses S is not affected if a clause $C_1 \in S$ subsumes a *newly formed clause* C_2, and C_2 is consequently deleted immediately after it has been formed. This is a special case of subsumption that preserves the completeness of the search strategy. Difficulties can occur if C_2 subsumes C_1 and C_1 is arbitrarily deleted (see Kowalski [1970a]). The process of subsumption is often costly in terms of computer time since a search of the set of generated clauses is necessary every time a clause is formed during a proof; furthermore, the subsumption test is itself a time-consuming operation. This option must therefore be used with care.

Since subsumption is relatively time-consuming, as a compromise we employ a third deletion rule for alphabetic variants. Two clauses are said to be alphabetic variants of each other if they are identical up to a change of variable names. Thus, $C_1 = P(f(a,x),y)$ is a variant of $C_2 = P(f(a,z),x)$. It should be clear that $C_1\sigma \subseteq C_2$ and thus C_2 can be safely removed since it is subsumed by C_1. This rule has the advantage that it is computationally efficient whereas in general, subsumption is not. Our experience has indicated that many redundant clauses are formed in a typical question answering search, and that deletion of alphabetic variants is advisable.

4.2 A Generalization of Σ^* and Upwards Diagonal Search

4.2.1 A Revised Definition of Diagonal and Upwards Diagonal Merit Ordering

As described in Section 3, when $f(n)$ is defined in terms of only clause length and level it does not discriminate well enough between those clauses that are relevant and those that are not, and it is doubtful that any costing function composed on only two clause features would be sufficient. It should be possible to define a more effective costing function by letting $g(n)$ and $h(n)$ be linear combinations of clause features, such that $g(n)$ and $h(n)$ are considered as single values. Alternately, f may be redefined to be a linear combination of an arbitrary number of functions rather than just one or two such that each function retains its value rather than being imbedded in $g(n)$ or $h(n)$. For instance, we might like f to be of the form $f(n) = w_0g(n) + w_1h_1(n) + w_2h_2(n)$, where the w_i are weights. Here g could be clause level, h_1 could be clause length and h_2 could be some form of functional complexity.

We would like to define our evaluation function f to be a function of $2m > 0$ arguments. That is, $v = f(w_1,w_2,\ldots,w_m; f_1,f_2,\ldots,f_m)$ where v is the value of the function, the w_i are weights and the f_i are feature values for a node in our search space (i.e., a clause).

We might then define f to be a linear discriminant function of the form

$$f(W,F) = w_1f_1 + w_2f_2 + \cdots + w_mf_m \tag{4.1}$$

where W is the row vector of weights and F is the column vector of features. Thus (4.1) can be re-written as a vector product:

$$f(W,F) = W\cdot F \ . \tag{4.2}$$

The function $f(W,F)$ defines a point in one-space and the vector W defines a point in m-dimensional weight space whereas F defines a point in m-dimensional feature space. Since the latter is a Euclidean space we shall call it E^m .

As an extension of this definition, we may redefine the merit orderings $\le_d$ and $\le_d u$ so that we can distinguish between clauses of equal f value. This parallels equation 3.1. Let node n have feature vector F and node n' have feature vector F' . In addition, let W be the weight *matrix* with off-diagonal elements equal to zero and with diagonal elements $w_1,w_2,\ldots,w_m$. Then

(1) $n \le_d u\ n'$ iff $f(W,F) \le f(W,F')$; (4.3)

(2) if $f(W,F) = f(W,F')$ then $n \le_d u\ n'$ iff $w_1f_1 \le w_1f_1'$;

(3) if $f(W,F) = f(W,F')$ and $w_1f_1 = w_1f_1'$ then $n \le_d u\ n'$ iff $w_2f_2 \le w_2f_2'$;

.

.

.

(m) if $f(W,F) = f(W,F')$ and for $j = 1,\ldots,m-2$, $w_jf_j = w_jf_j'$ then $n \le_d u\ n'$ iff $w_{m-1}f_{m-1} \le w_{m-1}f_{m-1}'$.

If in addition, $w_{m-1}f_{m-1} = w_{m-1}f_{m-1}'$ then $n =_d u\ n'$. Note that the case of $w_mf_m = w_mf_m'$ was not stated since if the equality condition holds for Steps 1 to m-1 and if $w_{m-1}f_{m-1} = w_{m-1}f'_{m-1}$ then $w_m f_m = w_m f'_m$ since $f(W,F) = f(W,F')$.

Equations 4.1 and 4.3 essentially define linear transformations upon the feature vector F to produce a new vector $\hat{F}$. The first transformation, τ_1 , was of the form $\tau_1\colon\ E^m \to E^1$ and produced a new vector $\hat{F}$ in one-space E^1 by the following matrix transformation:

$$\tau_1(F) = \hat{F} = (w_1 w_2 \ldots w_m) \cdot \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{pmatrix} = (w_1 f_1 + w_2 f_2 + \cdots + w_m f_m). \quad (4.5)$$

The second transformation τ_m was of the form $\tau_m\colon\ E^m \to E^m$ and formed $\hat{F}$ in m-space E^m by the following transformation:

$$\tau_m(F) = \hat{F} = \begin{pmatrix} w_1 & & \bigcirc \\ & w_2 & \\ & & \ddots \\ \bigcirc & & & w_m \end{pmatrix} \cdot \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{pmatrix} = \begin{pmatrix} w_1 f_1 \\ w_2 f_2 \\ \vdots \\ w_m f_m \end{pmatrix} \qquad (4.6)$$

As an example of such a transformation, let $m = 4$, $F(0,1,0,1)$ and the diagonal elements of $w = (1,2,1,1)$. Then

$$\hat{F} = \tau_4(F) = W \cdot F = \begin{pmatrix} 1 & & & \bigcirc \\ & 2 & & \\ & & 1 & \\ \bigcirc & & & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ 0 \\ 1 \end{pmatrix}$$

These two transformations are two members of a family of linear transformations we shall call T. Each $\tau_i \in T$ is defined $\tau_i\colon\ E^m \to E^i$, $1 \le i \le m$. Each vector component of $\hat{F} \in E^i$ will be a linear combination of some subset of the components of $F \in E^m$. Nodes of our search space are thus evaluated on the basis of their merit which is represented by an i-dimensional vector in E^i, namely $\hat{F}$ (not F).

We must thus work directly with the transformation matrix W instead of a vector W as before. It will be convenient to define W_j as the vector representing the j<u>th</u> row of W; similarly, f_j and $\hat{f}_j$ are the j<u>th</u> elements of F and $\hat{F}$ respectively. $\hat{F}$ is then derived from F by the following transformation:

$$\tau_i(F) = \hat{F} = W \cdot F = \begin{pmatrix} w_{11}w_{12}\cdots w_{1m} \\ w_{21}w_{22}\cdots w_{2m} \\ \cdot\quad\cdot\quad\quad\cdot \\ \cdot\quad\quad\cdot\quad\cdot \\ \cdot\quad\quad\cdot\quad\cdot \\ w_{i1}w_{i2}\cdots w_{im} \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ \cdot \\ \cdot \\ \cdot \\ f_m \end{pmatrix} = \begin{pmatrix} W_1 \cdot F \\ W_2 \cdot F \\ \cdot \\ \cdot \\ \cdot \\ W_i \cdot F \end{pmatrix} \tag{4.7}$$

As an example of a linear transformation of F, let $\tau_3\colon\ E^4 \to E^3$. Then $\hat{F} = W \cdot F$ might be represented by

$$\hat{F} = \begin{pmatrix} 3 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} = \begin{pmatrix} 3f_1 + f_4 \\ f_2 \\ 2f_3 + f_4 \end{pmatrix}$$

where $\hat{f}_1 = W_1 \cdot F = 3f_1 + f_4$

$\hat{f}_2 = W_2 \cdot F = f_2$ and

$\hat{f}_3 = W_3 \cdot F = 2f_3 + f_4$.

Keeping this transformation in mind, we may now give a revised definition for $\leq_d$ and $\leq_d u$ with respect to the transformation τi: $E^m \to E^i$. This new definition will be termed ordering-A. Let node n have feature vector F and node n' have feature vector F' . W is the weight matrix with i rows and m columns. Define

$$f(W,F) = \sum_{j=1}^{i} W_j \cdot F = \sum_{j=1}^{i} \hat{f}_i \tag{4.8}$$

That is, f is the sum of the components of $\hat{F}$. Note that some other measure of diagonal merit could have been used. The above form was chosen because it resembles a linear discriminant function.

Diagonal and upwards diagonal merit for ordering-A can thus be defined by:

(1) $n \leq_d n'$ iff $f(W,F) \leq f(W,F')$; (4.9)

(2) if $f(W,F) = f(W,F')$, then $n \leq_d u\ n'$ iff $W_1 \cdot F \leq W_1 \cdot F'$;

(3) if $f(W,F) = f(W,F')$ and $W_1 \cdot F = W_1 \cdot F'$ then $n \leq_d u\ n'$ iff $W_2 \cdot F \leq_d u\ W_2 \cdot F'$;

.
.
.

(m) if $f(W,F) = f(W,F')$ and for $j = 1,\ldots, m-2$, $w_j \cdot F = W_j \cdot F'$ then $n \leq_d u\ n'$ iff $W_{m-1} \cdot F \leq W_{m-1} \cdot F'$.

If in addition, $W_{m-1} \cdot F = W_{m-1} \cdot F'$ then $n \;{}_d\!\!=^u n'$.

We thus see that the merit vector $\hat{F}$ is composed of i components, each of which is a linear combination of some subset of m clause features. However, up to this point, we have not related the concept of distinct g and h functions with that of multiple components of $\hat{F}$. In order to do this, some components will be defined as "g-type" and others as "h-type" since certain features inherently are a measure of the cost of a path from a start node to a node n , whereas others are a measure of the heuristic cost of a path from node n to a goal node. One would expect a search strategy that utilizes this information to perform better than one that does not. Note that for ordering-A, the lower subscripted components of $\hat{F}$ are minimized before those numbered higher. Since the heuristic function h should in practice be minimized before the g function, the user should make sure that the lower numbered components correspond to the heuristic component h .

A variation of ordering-A that differentiates more between g and h parameters will be called Ordering-B. In this ordering all h components would by convention be the subvector $(W_1 \cdot F , W_2 \cdot F, \ldots, W_j \cdot F)$ and the g components would be $(W_{j+1} \cdot F , W_{j+2} \cdot F, \ldots, W_i \cdot F)$ where there are j h-components and $i-j = k$ g-components for the transformation $\tau_i\colon\; E^m \to E^i$. Let

$$H = \sum_{\ell=1}^{j} W_\ell \cdot F \quad \text{and} \quad G = \sum_{\ell=j+1}^{i} W_\ell \cdot F \tag{4.10}$$

for node n ; H' and G' are defined simi'arly for node n' .

Diagonal and upwards diagonal merit for Ordering-B can thus be defined by:

(1) $n \leq_d u\ n'$ iff $f(W,F) \leq f(W,F')$; (4.11)

(2) if $f(W,F) = f(W,F')$ then $n \leq_d u\ n'$ iff $H \leq H'$;

(3) if $f(W,F) = f(W,F')$ and $H = H'$ then $n \leq_d u\ n'$ iff $W_{j+1} \cdot F \leq W_{j+1} \cdot F'$ (i.e., we are comparing the first g components);

.
.
.

(k+1) if $f(W,F) = f(W,F')$, $H = H'$ and for $\ell = j+1,\ldots, i-2$, $W_\ell \cdot F = W_\ell \cdot F'$ then $n \leq_d u\ n'$ iff $W_{i-1} \cdot F \leq W_{i-1} \cdot F'$;

(k+2) if in addition, $W_{i-1} \cdot F = W_{i-1} \cdot F'$ then $n \leq_d u\ n'$ iff $W_1 \cdot F \leq W_1 \cdot F'$ (i.e., we now start discriminating on the basis of h components);

.
.
.

(i) if $f(W,F) = f(W,F')$, $H = H'$, and for $\ell = j+1 ,\ldots, 1-1$ $W_\ell \cdot F = W_\ell \cdot F'$ and for $\ell = 1 ,\ldots, j-2$, $W_\ell \cdot F = W_\ell \cdot F'$ then $n \leq_d u\ n'$ iff $W_{j-1} \cdot F \leq W_{j-1} \cdot F'$.

If in addition $W_{j-1} \cdot F = W_{j-1} \cdot F'$ then $n =\,=_d u\ n'$. Thus, whereas Ordering-A treats all merit parameters alike (except for their order in the sequence $\hat{F}_1,\hat{F}_2,\ldots,\hat{F}_i$ defined by the user), Ordering-B groups all h-parameters together and minimizes their sum before resorting to pair-wise parameter comparisons. At the present time, it is not clear which method leads to a more efficient search strategy. Further experimentation will hopefully help us in this regard.

It is interesting to note several special cases of the above two orderings that result when the transformation $\tau_i\colon\ E^m \to E^i$ is varied. In all these cases, however, Ordering-A is identical to Ordering-B since only two components of $\hat{F}$ are allowed.

The above merit orderings are identical to a strictly diagonal merit ordering (i.e., only values of $f(W,F)$ are compared) when $i = 1$ and $\tau_1\colon\ E^m \to E^1$ is used (equation 4.5). For $i = 2$, $\tau_2\colon\ E^m \to E^2$ is the transformation used. If in addition, $m = 2$ and $w_{jk} = \delta_{jk}$, the orde ing defined is the same as that described by Kowalski [1970b]. Thus, if $f_1 = g$ and $f_2 = h$

$$\hat{F} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} \qquad (4.12)$$

and

$$f(W,F) = f_1 + f_2 = g + h$$

A similar case occurs when $m = 2$, $f_1 = g$, $f_2 = h$, $w_{11} = (1-\omega)$, $w_{22} = \omega (0 \le \omega \le 1)$, and $w_{12} = w_{21} = 0$. Then

$$\hat{F} = \begin{pmatrix} 1-\omega & 0 \\ 0 & \omega \end{pmatrix} \begin{pmatrix} g \\ h \end{pmatrix} = \begin{pmatrix} (1-\omega)g \\ \omega h \end{pmatrix} \qquad (4.13)$$

and $f(W,F) = (1-\omega)g + \omega h$ corresponds to the representation of evaluation functions given by Pohl [1970]. A more general case that can be made to conform to Pohl's representation is for $\tau_2\colon\ E^m \to E^2$ and $m \ge 2$. Recall that we defined $f(W,F)$ to be $\sum_{j=1}^{i} W_j \cdot F$. If $i = 2$, $W_1 \cdot F$ may be thought of as g and $W_2 \cdot F$ as h, where g and h are *composite* functions. If we now assign *additional weights* $(1-\omega)$ and ω to g and h respectively, we can redefine f as

$$f(W,F) = \sum_{j=1}^{2} \alpha_j W_j \cdot F = (1-\omega)\ W_1 \cdot F + \omega W_1 \cdot F \qquad (4.14)$$

where $\alpha_1 = 1-\omega$ and $\alpha_2 = \omega$, $0 \leq \omega \leq 1$.

The advantages of using a representation such as (4.14) are four-fold.

(1) As stated previously, it seems intuitive that a search strategy that differentiates as much as possible between g and h parameters should perform better than one that does not. The ordering of (4.14) enables us to do this.

(2) Results have been obtained by Kowalski for the case of $m = 2$, $i = 2$ with respect to admissibility and optimality. We therefore know a great deal about how such a search strategy should behave.

(3) The general definition of upwards diagonal merit ordering describes an algorithm for ascertaining whether or not a node n is of better or equal merit than another node n' . We have found that such as algorithm is fairly efficient for $i = 2$ but rapidly becomes computationally inefficient for $i > 2$. We would therefore prefer to use only two components to keep our search overhead as low as possible.

(4) It seems very likely that the same amount of discrimination between clauses achieved by the transformation τ_m: $E^m \to E^m$ may be obtained through judicious choice of the weights in W for the transformation τ_2: $E^m \to E^2$. If this is the case, equation 4.14 is the preferable ordering to use.

It is planned that future versions of MRPPS will allow all of the above merit orderings including that of Equation 4.14. However, the current implementation allows only Ordering-A and Ordering-B with an identity weight matrix W used in conjunction with a transformation τ_m: $E^m \rightarrow E^m$ for $m = 1,\cdots,5$ (Section 4.5.3 lists the clause features currently available). In particular MRPPS now gives the user the capability of selecting (1) the ordering to be used, (2) which parameters to treat as "g-type" and which to treat as "h-type", and (3) which clause feature to assign to each component of F. Note that (2) is only significant if Ordering-B is chosen.

It should be emphasized that insufficient attention to points (1)-(3) above can lead to a very inefficient proof procedure. For instance, if level were considered an "h-type" and length as a "g-type" parameter, the search process would tend to generate long clauses before shorter clauses -- quite contrary to the meaning of heuristic distance!

4.2.2 The Completeness, Admissibility, and Optimality of Ordering-A and Ordering-B

It is the purpose of this section to show the compatibility of these orderings with the diagonal and upwards diagonal orderings defined by Kowalski for the function $f(n) = g(n) + h(n)$. Let $\leq_d$ and $\leq_d u$ be diagonal and upwards diagonal orderings so defined (refer to Equation 3.1) and let $\sum$ be a search strategy. Step (1) of Equation 4.9 for Ordering-A is essentially identical to Step (1) of Equation 3.1 that defines a diagonal merit ordering. The other steps of 4.9 serve to discriminate more finely between clauses of equal f values. Thus, all clauses generated by $\sum$ using $\leq_d$ for a given f will also be generated by $\sum$ using Ordering-A for that f value. Thus, $\sum$ with Ordering-A is certainly complete when $\sum$ when $\leq_d$ is complete. Also, Ordering-A permits at least as "good" or minimal a solution as $\le_d$ does, and also allows $\sum$ to be as optimal (i.e., as well informed) as $\le_d$ allows $\sum$ to be. If the heuristics are such that in advance of performing resolution, the sets to which the resolvent belongs is known with certainty, and the conditions on the heuristic specified below are met, then the algorithm is admissible since the null clause is found on some diagonal. Each diagonal is explored before going on to the next one. Hence, admissibility cannot be violated.

A similar correspondence can be made between Ordering-B and $\le_d u$. Steps (1) and (2) of Equation 4.11 are essentially identical to Steps (1) and (2) of Equation 3.1 that define an upwards diagonal merit ordering. Again, the remaining steps of Equation 4.11 serve only to discriminate more finely between clauses with equal f values and equal H values (recall that H is the sum of all h-type components of $\hat{F}$). $\sum$ with Ordering-B is therefore complete when $\sum$ with $\le_d u$ is complete. Also, Ordering-B is admissible for the same reason that Ordering-A is admissible.

It should be noted that the completeness, admissibility and optimality of $\sum$ depends upon the characteristics of the heuristic components used in the evaluation function $\hat{F}$. For instance, if only the parameter of clause length is used as the heuristic measure, $\sum$ is not even complete and therefore not admissible or optimal. Although we do not feel that admissibility and optimality are crucial for question-answering systems, *refutation* completeness is a more desirable property. By refutation completeness we mean that $\sum$ will generate the null clause $\Box$ whenever a contradiction does indeed exist (although the entire search space need not be generated as is required for completeness). Thus, great care should be taken in choosing the heuristic components.

In relation to the above, Kowalski has shown [1970b] that diagonal and upwards diagonal merit orderings allow $\sum$ to be admissible if the following two conditions are met :

(1) $\leq_d$ (or $\leq_d u$) must be δ-finite; that is, for any node n in our search space, $\sum$ will generate only a finite number of nodes of better or equal merit; and,

(2) that part of $f(n)$ designated as the heuristic function $h(n)$ must satisfy the lower bound condition; that is, $h(n) \leq h^*(n)$ for all n in the search space, where h^* is the *actual* heuristic cost of node n , and h is only an estimate.

In particular, if $\leq_d$ and $\leq_d u$ are δ-finite, diagonal and upwards diagonal search are complete. Optimality follows when $h(n)$ satisfies conditions (1) and (2) above and also a monotonicity condition, namely:

(3) $f(n') < f(n)$ for $n' \prec n$ and $f(n^*) = g(n^*)$ for a null clause n^* , where $n' \prec n$ means that node n' is generated before node n .

4.3 Heuristic Measures Used in MRPPS

4.3.1 Clause Level

It is often advantageous to define the evaluation function f with a component being clause level. The level of a clause is defined as:

$$\ell(C) = \begin{cases} 0 \text{ if } C \text{ is a base clause} \\ \max(\ell(C_1), \ell(C_2)) + 1 \text{ where } C = \text{Resolvent } (C_1,C_2) \\ \ell(C_1) + 1 \text{ where } C = \text{Factor } (C_1) . \end{cases}$$

The level of a clause C is a measure of how many inference steps were required to derive C from the base clauses. Thus, if level were the only component of f used with an algorithm such as Σ^* , a breadth-first search would result since *all* derivations of level i would be formed before those of level i+1 . Obviously it should not be used alone. However, it can be valuable when used in conjunction with other components since it tends to keep the search from running away in an infinite deduction path.

4.3.2 Clause Length

The number of literals of a clause C is a very rough measure of the cost incurred in inferring the null clause from C . This heuristic has been used in many other theorem provers (e.g., Garvey and Kling [1969], Norton [1972]), and is a very effective component of f(n) . The system QA3.5 used *only* length as a heuristic function by employing the unit preference strategy (Wos, Robinson, Carson [1964]). However, length alone, or even in conjunction with clause level does not lead to a practical heuristic, as seen in Section 3.3. In *combination* with other components, though, length will help the search for a contradiction.

It is interesting to note that if length is the sole heuristic used with the Σ^* algorithm (or with the Q* algorithm of MRPPS, to be described later), a different search results than for unit preference as defined in QA3.5. Recall that in the latter, all units are resolved against other units to produce all inferences possible. If none are found, all length two clauses are resolved with all units, then length three clauses, etc. On the other hand, Σ^* would resolve units with units, then two-clauses with units, then two-clauses with two-clauses, then three-clauses with units, etc. Thus, Σ^* tries to minimize the length of resolvents, rather than always attempting to *reduce* clause length as does the unit preference strategy.

4.3.3 Clause Complexity

A simple definition of functional complexity may also be used as a heuristic component of f. The complexity of a clause is defined to be the maximum level of function nesting taken over all predicates of a clause. For example, the following list gives samples of the complexity of several clauses:

Clause	Complexity
P	0
$P(x,y)$	1
$P(f(g(a)),f(z)) \vee Q(c)$	3
$P(f(a),g(b),h(c,x))$	2

The complexity heuristic is as follows: if C_1 and C_2 are clauses with complexity h_1 and h_2 respectively, and $h_1 < h_2$, then it is better to generate successors for C_1 rather than for C_2. In other terms, we would like to keep our proof as "simple" as possible with respect to clause complexity.

On the other hand, there is a very good reason to generate clauses of high complexity *before* those of low complexity. Clauses that have deeply nested functions will probably resolve with a very small number of other clauses whereas a simple clause with variables is apt to resolve with a large number.

This effect is compounded when a complex clause is resolved with another clause by substituting terms containing constants for other terms. The resulting clause is partially instantiated and thus unlikely to resolve with many clauses. Since we are trying to keep the number of clauses generated to a minimum, this seems like a reasonable approach to take.

In any case, the above definition of complexity does not seem to distinguish finely enough between clauses and we do not know whether or not it will be useful. Perhaps a definition that sums the complexities of all predicates of a clause would yield a better heuristic. Future versions of MRPPS will utilize different complexity definitions.

4.3.4 Cluster Distance

The heuristics described in the previous sections are syntactic in nature. That is, they measure properties of a clause which can be determined by simply examining the clause and measuring how much of this property is present, e.g., how many literals are in the clause. Previous work in theorem proving has also made use of these types of measures and although they have proven theoretically useful, they do not provide sufficient direction to the search to permit its use in practical situations.

In the present section we shall define a heuristic measure which is more semantic in nature. The measure will take into account the nature of the general axioms in the system's data base and will provide some insight as to when in the search certain of the axioms are likely to be useful. As will be demonstrated by an example, the heuristic should prove useful in restricting the number of clauses generated during the course of the search. The use of clusters as a heuristic, was proposed by Minker in Minker and Sable [1970]. It has been defined more precisely in this report.

Before we are able to define what we mean by cluster distance, some preliminary discussion will be necessary to establish what we mean by *semantic clusters* relative to which the distances are measured.

In the data base of a question-answering system we shall have a set A of general axioms given in clause form, $A = \{A_1, A_2, \ldots, A_n\}$. By taking an accounting of the predicates occurring in these clauses we may construct a set $P = \{P_1, P_2, \ldots, P_m\}$ of predicates occurring in A. We may then define an *axiom-predicate matrix* $A_p = [a_{ij}]$ as the following:

$$a_{ij} = \begin{cases} 1 & \text{if predicate } P_j \text{ occurs at least once in axiom } Ai \text{, regardless of whether it appears as a negated literal or not negated} \\ 0 & \text{otherwise} \end{cases}$$

We may now define a *predicate-predicate matrix* $B_p = [b_{ij}]$ as:

$$B_p = A_p^T \cdot A_p \, .$$

where A_p^T is the transpose of A_p.

The matrix B_p indicates the degree of relatedness of predicates in the following sense: the value of the element b_{ij} of B_p gives the exact number of axioms in which both predicates P_i and P_j occur. If $b_{ij} \neq 0$, we may say that P_i and P_j are related. But, if $b_{ij} = 0$, then P_i and P_j do not co-occur in any axiom. It may be that $b_{ij} = 0$, but that there is some P_k such that $b_{ik} \neq 0$ and $b_{kj} \neq 0$ in which case since both P_i and P_j are related to P_k they are somewhat related to each other. In general, there may be predicates $P_{k_1}, P_{k_2}, \ldots, P_{k_r}$ such that $b_{ik_1} \neq 0$ and $b_{k_rj} \neq 0$ and $b_{k_1k_2} \neq 0, \ldots, b_{k_{r-1}k_r} \neq 0$ in which case we might say with that P_i and P_j are rather tenuously related. If there were no such chain of predicates between P_i and P_j we would say that they are completely unrelated.

The relationships described by the matrix B_p may be depicted by a graph. To B_p there corresponds an undirected graph G_p, which we term a *semantic graph*, containing one node per predicate and an edge between predicates P_i and P_j iff $b_{ij} \neq 0$. In general, the graph will consist of a number of connected components, that is, of individual subgraphs which are not connected to each other. We term these connected components *semantic clusters.*

We provide the following specific example to make the above notions more concrete:

The set A of general axiom clauses:

A1. $\overline{P}_8(x,y) \vee P_6(f(x),y)$

A2. $\overline{P}_2(x,y) \vee P_3(x) \vee P_1(v,x)$

A3. $\overline{P}_3(x,y) \vee \overline{P}_4(x)$

A4. $\overline{P}_1(x,v) \vee \overline{P}_1(y,z) \vee P_2(x,z)$

A5. $\overline{P}_6(x,v) \vee \overline{P}_7(x) \vee P_7(y)$

A6. $\overline{P}_5(x,x) \vee P_4(x)$

The Set P of Predicates:

$P = \{P_1, P_2, \ldots, P_8\}$

From A we find that the axiom-predicate matrix A_p has the value:

$$A_p = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

From A_p we find the predicate-predicate matrix B_p has the value:

$$B_p = A_p^T \cdot A_p = \begin{bmatrix} 2 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

The semantic graph G_p corresponding to B_p is shown below as consisting of two semantic clusters:

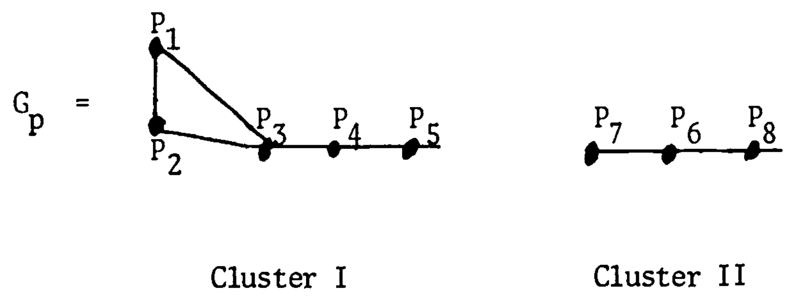


FIGURE 2. A SEMANTIC GRAPH WITH TWO CLUSTERS

We note that the clusters give a graphical account of the interrelatedness of the various predicates. For example, we see that P_1, P_2, P_3 are closely interrelated, but are less closely related to, say, P_5. Also, the predicates in Cluster I are completely unrelated to those in Cluster II.

Suppose we are presented with a query, either posed in, or converted to the clause form of the first-order predicate calculus. Suppose that all of the predicates in the query occur in, say, Cluster I. Then, any axioms which give rise to Cluster II, i.e., axioms containing instances of P_6, P_7, or P_8, are irrelevant to responding to the query and need never be considered for this purpose. Since the "cluster II" axioms, and any of their successors cannot logically interact with the query clauses, they are never used in the proof. Less obvious advantages of using the clusters will be discussed presently.

We note that in the above definitions we have used only the general axioms as the basis for forming the semantic graph. We believe that the facts stored in a QA System's data base will be predominantly in the form of fully-instantiated *unit* clauses which do not, in the above sense, interact with other axioms. Should any of the facts be represented in clauses of length greater than one, then these too would be included in the axiom-predicate matrix. Furthermore, in the event that there are fully-instantiated unit clauses whose predicates do not occur in the general axioms, explicit entry of these predicates into the axiom-predicate matrix would be made. Such predicates would give rise to isolated points in the semantic graph.

At this point we are now ready to define what we mean by cluster distance. Two alternative definitions will be presented, the min cluster distance of a clause and the max cluster distance. These distances are measured from a clause to a set of clauses. The set of clauses we measure the distance to will be those derived from the negation of a query. The distance is measured relative to the semantic clusters developed from the set of general axioms.

Let A be a set of general axioms, and let G_p be the semantic graph derived from A as described above. Before defining the cluster distance of a clause from a set of clauses, we define the usual graph-theoretic distance measure:

DEFINITION 1. (Graph-Theoretic Distance)

Let P_i and P_j be two nodes (predicates) in G_p. Then the (graph-theoretic) *distance* between P_i and P_j, denoted $\delta(P_i,P_j)$, is the length of the shortest path in G_p from P_i to P_j. If P_i and P_j occur in different clusters of G_p, then $\delta(P_i,P_j)$ is set to ∞.

DEFINITION 2. (Max Cluster Distance)

Let C be a clause in which there occurs the predicates

$P_1, P_2, \ldots, P_n$, and let Q be a query in which the predicates $Q_1, Q_2, \ldots, Q_m$ occur. Then the *max cluster distance of C from the query* Q , denoted $\nabla_Q(C)$, or simply $\nabla(C)$ when Q is understood, is given

$$\nabla_Q(C) = \max_{P_i \in C} \{\min_{Q_j \in Q} \{\delta(P_i, Q_j)\}\} .$$

It will be shown below how this distance measure may be used to advantage by a heuristic search algorithm. For the moment, we content ourselves with some example computations of the max cluster distance of a clause.

EXAMPLE

Let the negation of the query, which we denote by $\{\sim Q\}$, and the clause C be as given below:

$$\begin{aligned} \{\sim Q\}&: \quad P_1 \vee P_3 \\ C &: \quad P_1 \vee P_5 \end{aligned}$$

Then $\nabla_Q(C)$, relative to the semantic graph G_p of Figure 2 is calculated below:

$$\begin{aligned} \nabla_Q(C) &= \max_{P_i \in C} \{\min_{P_j \in Q} \{\delta(P_i, P_j)\}\} \\ &= \max \{\min \{\delta(P_1,P_1), \delta(P_1,P_3), \min \{\delta(P_5,P_1), \delta(P_5,P_3)\}\} \\ &= \max \{\min \{0,1\}, \min \{3,2\}\} \\ &= \max \{0,2\} \\ &= 2. \end{aligned}$$

Thus, clause C is a max cluster distance of 2 from query Q .

EXAMPLE

Suppose that Q and G_p are as given in Figure 2, and the clause C is $P_7 \vee P_6$. Then we have

$$\begin{aligned}\nabla_Q(C) &= \max\{\min\{\delta(P_7,P_1), \delta(P_7,P_3)\ , \min\{\delta(P_6,P_1), \delta(P_6,P_3)\\ &= \max\{\min\{\infty,\infty\}\ \min\{\infty,\infty\}\}\\ &= \max\{\infty,\infty\}\\ &= \infty\ .\end{aligned}$$

If clause C happened to be an axiom in the data base, then one could tell by its max cluster distance from the query that it is irrelevant in processing the query.

EXAMPLE

Suppose we have

$$\begin{aligned}\{\sim Q\}&:\ P_1 \vee P_8\\ C&:\ \sim P_4 \vee P_6\\ G_p&:\ \text{as above}\end{aligned}$$

Then we have

$$\begin{aligned}\nabla_Q(C) &= \max\{\min\{\delta(P_4,P_1), \delta(P_4,P_8)\}, \min\{\delta(P_6,P_1), \delta(P_6,P_8)\}\}\\ &= \max\{\min\{2,\infty\},\min\{\infty,1\}\\ &= 2.\end{aligned}$$

Thus, we see that if each of the predicates of the clause occurs in a cluster with at least one of the query predicates, then the max cluster distance of the clause from the query will be finite.

A distance measure which is analogous to the above is the min cluster distance:

DEFINITION 3. (Min Cluster Distance)

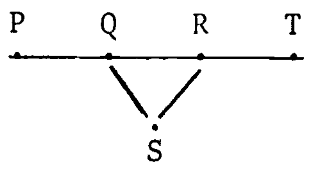
Given the assumptions of Definition 2, above, the min cluster distance of C from query Q , denoted $\Delta_Q(C)$, or simply $\Delta(C)$ when Q is understood, is given by

$$\Delta_Q(C) = \min_{P_i \in C} \{\min_{Q_j \in Q} \{\delta(P_i,Q_j)\}\} .$$

Below we will show how the min cluster distance may be used in directing a search.

EXAMPLE

Suppose we have the unsatisfiable set of clauses $\{\overline{S},\overline{T},\overline{Q},\overline{P},RT,PQ,\overline{QRS}\}$. The semantic graph for these clauses (based on the three non-unit clauses) is

$G_p =$ 

Assume that the negation of the theorem is $\overline{P}$. Then the min cluster distances relative to $\overline{P}$ are:

C	$\overline{S}$	$\overline{T}$	$\overline{Q}$	$\overline{P}$	RT	PQ	$\overline{QRS}$
$\Delta(C)$	2	3	1	0	2	0	1

In this example, assume the merit function is simply the composite of length and min cluster distance and that each A(i,j) corresponds to A(length,Δ) . The algorithm will behave like the $\sum^*$ algorithm (section 3.2) in all respects except for the substitution of Δ for level.

1. Fill(0,0) is unsuccessful: $A(0,0) = \varphi$
2. Fill(0,1) is unsuccessful: $A(0,1) = \varphi$

3. Fill(1,0) is successful: $A(1,0) = \{\overline{P}\}$

   Recurse($\overline{P}$) is unsuccessful

4. Fill(0,2) is unsuccessful: $A(0,2) = \varphi$

5. Fill(1,1) is successful: $A(1,1) = \{\overline{Q}\}$

   Recurse($\overline{Q}$) is successful:

6. Fill(2,0) is successful: $A(2,0) = \{PQ\}$

   Recurse(PQ) yields Q : $A(1,1) = \{\overline{Q},Q\}$

   Recurse(Q) yields $\square$: $A(0,2) = \{\square\}$

A summary of the steps in the search follows.

1. $\overline{P} \in A(1,0)$
2. $\overline{Q} \in A(1,1)$
3. $PQ \in A(2,0)$
4. $Q \in A(1,1)$ (from 1,3)
5. $\square \in A(0,2)$ (from 2,4)

If the search were performed by the $\sum^*$ algorithm using length and level, the following steps would have occurred:

1. $\overline{S} \in A(1,0)$
2. $\overline{T} \in A(1,0)$
3. $\overline{Q} \in A(1,0)$
4. $\overline{P} \in A(1,0)$
5. $RT \in A(2,0)$
6. $R \in A(1,1)$ (from 2,5)
7. $PQ \in A(2,0)$
8. $Q \in A(1,1)$ (from 4,7)
9. $\square \in A(0,2)$ (from 3,8)

While the above example handles just a trivial problem, we can observe from it how the use of the cluster distance discriminates, among clauses of equal length, in favor of those which are more "closely related" to, and in this sense, more relevant to the query. On the other hand, the $\sum^*$ algorithm, using only length and level cannot accomplish this sort of discrimination and so it generates all of the unit clauses before bringing in the two-clauses. Of course, the use of clustering would also permit these unit clauses to enter if the search were any deeper. However, if there were any base clauses whose predicates are not in the semantic cluster with query predicates, such clauses could be generated by $\sum^*$, but not by the cluster distance based algorithm.

Aside from prohibiting base clauses which are unrelated to the query from entering the search space, the cluster distances appear to be useful heuristics for still other reasons. In the first place, they will have the effect of sequencing the axioms which will enter the search in such a way that when an axiom is generated, it has a good chance of being used in an inference. Secondly, it has the effect of ordering the inferences in such a way that the clauses which are more closely related to the query are used in inferences before more "distant" clauses. Whether the min cluster distance is better than the max cluster distance or vice versa, must be determined by experimentation, as indeed, must the question of whether or not this measure will have any utility at all for question answering systems.

Cluster analysis may be used in another way. If one finds clusters of the clauses that result from the axioms, then these clauses should be located near one another. This is particularly true when the data base is stored on peripheral devices. Then, when one clause is brought into core, a clause in the same cluster might also be brought into core.

4.4 The Base Clause Selection Strategy of MRPPS

4.4.1 General Discussion

It was noted in Section 3.1 that search algorithms such as the A* or $\sum^*$ algorithms require that the merit of all clauses in the starting set S be established prior to the start of the search so that the clauses may be ordered by their merit. This requirement has at least two disadvantages. In the first place, if the merit of a clause is to depend upon the query, as is the case with the cluster heuristic, then the merit of clauses in the data base must be calculated, and the data base reordered, at the start of every search. For any "reasonably" sized data base, this situation would clearly be intolerable. Furthermore, and of even greater significance, the static assignment of merit to the base clauses at the start of the search ignores relevance clues which may become evident during the course of the search or which may be provided initially. In fact, as it will be shown, it is often possible to glean from a given search step which of the available base clauses would be "best", in some sense, to generate (add to an A-set) next. As a trivial example of this, suppose that the search strategy has somehow generated the unit clause ~F(Jack, Sally) , and is now ready to bring in an axiom from the data base. Suppose that the merit function is composed of clause length and clause level. Then, with respect to this merit function, the clause M(Rita, Mike) and F(Jack, Sally) would be indistinguishable, as would be the large bulk of unit clauses in the system's data base. Thus, the clause brought in next would be a matter of how the clauses happened to be ordered even though the clause F(Jack, Sally) , if brought in, would lead to an immediate refutation. Similarly, the use of the clustering heuristic, while it might restrict consideration to only the unit "F" clauses, the obvious best clause would be indistinguishable from a possbily large list of other such clauses.

The above observations have led to the development of the Q* search algorithm which embodies two search strategies. One strategy, called the *deduction strategy*, (or the deduction algorithm), determines which inferences to attempt next and forms deductions by resolution, factoring and paramodulation. This is essentially the $\sum^*$ algorithm (Section 3.2) with several modifications (to be described in Section 4.5). The second strategy is the *base clause selection strategy* (or algorithm) which determines the next base clause to be generated. It is this latter strategy that is the subject of the present section.

The base clause strategy is in part based upon the premise that no axiom should be generated unless, at a minimum, it possesses a literal which will unify with a literal of a clause already generated. At that time, a base clause may become a "candidate for generation." For certain inference systems stricter constraints may be imposed. For example, if the inference system is, or includes, set-of-support, then only axioms which could interact logically with a supported clause will be considered for generation. In this case an axiom will be considered for generation if it possesses a literal which is opposite in sign to and unifies with a literal of a supported clause which has already been generated. Similar restrictions for other inference systems are possible. However, the restrictions imposed are such that the refutation completeness of the Q* algorithm is preserved.

From the above comments it should be evident that the base clauses are not initially assigned a merit. Rather, each base clause is initially assumed to have an *undefined* merit, which is only to become defined when the clause becomes a candidate for generation. Furthermore, not all candidates will have their merit calculated explicitly when they become candidates. In general, their merit will be defined *implicitly* by virtue of their membership in a list of clauses of equal merit.

For example, suppose a clause has been generated which contains the literal $\overline{F}(x,y)$. Then the list of data clauses $F(a_1,b_1)$, $F(a_2,b_2)$, $\ldots,F(a_n,b_n)$ would all become candidates for generation. But all of these clauses have the same merit with respect to length, level,complexity, and cluster distances, and so we need only assign a merit value to the list which applies to all of its members. By this use of dynamic merit calculation for base clauses and implicit clause merits, we are able to avoid a great deal of computation.

The use of these lists of base clauses has another striking advantage which is central to the base clause strategy. Suppose we have one list of candidate base clauses which have become candidates because of the generated literal $\sim F(x,y)$, and another list of candidates because of the generated literal $\sim F(x, \text{Sally})$. For example, one might have: $\sim F(x, \text{Sally})$ points at the list $F(\text{Jack, Sally})$, and $\sim F(x,y)$ points at the list $F(a_1,b_1)$, $F(a_2,b_2),\ldots,F(a_n,b_n)$. Then, while all of these data axioms have equal merit with respect to the merit function used by the deduction strategy, the base clause strategy will rank F(Jack, Sally) above the other clauses. This is done by virture of the presence of the constant "Sally" in the generated literal. Thus, if we are asked a question about an individual, we wish to explore avenues which are relevant to that individual before we explore other avenues. If, by pursuing this policy we come across another individual, then that individual, being relevant to the first, will also be used to direct the search.

From the above discussion we see that the query will have a strong influence on the search. Its literals will be used to establish which base clauses should become candidates for generation, and its constants will be used to order the candidates. Generated clauses will be used in the same way. Thus, if we have a constant in one literal of the query, resolving on this literal will often have the effect of forming an instantiated literal in the resolvent whose occurrence in the resolvend was uninstantiated; the term used in the instantiation will be relevant to the query constant. By transferring these clues from parent clauses to generated clauses by means of instantiations, the generated clauses can have a profound effect on the search for relevant base clauses.

In Figure 3 is depicted a refutation which could easily have been derived using the above ideas. In that example, suppose the query was "Who is Sally's mother's mother-in-law?" As a wff, this query might have been phrased:

$$(\exists x)(\exists y)(ML(x,y) \wedge M(y, Sally)) .$$

In clause form, the negation of the query would be

$$\overline{ML}(x,y) \vee \overline{M}(y, Sally) .$$

Suppose we had among the general axioms the two axioms

(a) $\overline{M}(u,v) \vee \overline{H}(v,w) \vee ML(u,w)$, and

(b) $\overline{F}(x,y) \vee \overline{M}(z,y) \vee H(x,z)$.

(a) states that if "u" is the mother of "v" and if "v" is the husband of "w" , then "u" is the mother-in-law of "w" , while (b) states that if "x" is the father of "y" and "z" is the mother of "y" then "x" is the husband of "z" . Suppose also that there are a large number of data axioms describing who is the mother and who is the father of particular individuals, among which were the three needed for

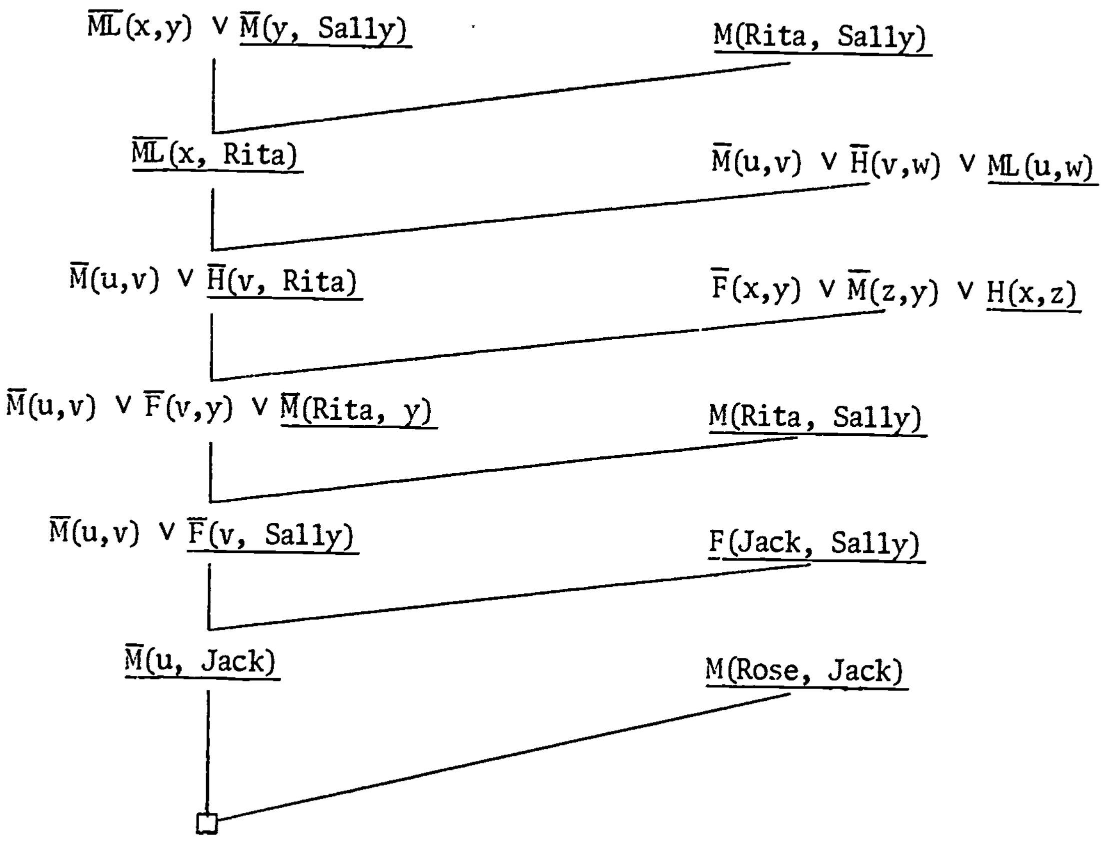

$\overline{ML}(x,y) \vee \underline{\overline{M}(y, Sally)}$
$\underline{M(Rita, Sally)}$
$\underline{\overline{ML}(x, Rita)}$
$\overline{M}(u,v) \vee \overline{H}(v,w) \vee \underline{ML(u,w)}$
$\overline{M}(u,v) \vee \underline{\overline{H}(v, Rita)}$
$\overline{F}(x,y) \vee \overline{M}(z,y) \vee \underline{H(x,z)}$
$\overline{M}(u,v) \vee \overline{F}(v,y) \vee \underline{\overline{M}(Rita, y)}$
$\underline{M(Rita, Sally)}$
$\overline{M}(u,v) \vee \underline{\overline{F}(v, Sally)}$
$\underline{F(Jack, Sally)}$
$\underline{\overline{M}(u, Jack)}$
$\underline{M(Rose, Jack)}$
$\square$

FIGURE 3. REFUTATION BASED ON INSTANTIATED LITERALS

the refutation: M(Rita, Sally) , M(Rose, Jack) , and F(Jack, Sally) . Under these circumstances the base clause strategy will assure that the necessary base clauses are available when they are needed. Note that, in violation of the $\sum^*$ algorithm, the two axioms of length three will be generated in this case *before* other axioms of length one since the base clause strategy deemed them to be most relevant at that stage of the search.

As a final note we observe an important characteristic of the Q* algorithm derived from the base clause strategy. In the case that there is an *immediate* answer to a question, as in the case of a direct look-up type question one encounters with conventional data management systems, then the Q* algorithm will find it immediately. For example, suppose one asks the question, "Who is the father of Sally?" which corresponds to the wff $(\exists x)\ F(x, \text{Sally})$. The negation of this query, $\overline{F}(x, \text{Sally})$ will be entered into an A-set and the very first clause provided by the base clause strategy will be F(Jack, Sally) . Thus, in one step the search will be concluded successfully making this approach competitive with generalized data management systems which are basically table look-up methods. This property is noted by Coles [1969] to be an important one for QA systems, i.e., answering simple questions quickly and efficiently. In this connection, the actual answer to this, or any other question, will be available in the system by means of the Luckham-Nilsson answer extraction algorithm [1970].

4.4.2 The Base Clause Selection Algorithm

The base clause selection algorithm attempts to provide directionality to the overall search for a refutation. This is attempted by using the literals and constants occurring in the query and in subsequently generated clauses as clues to the relevance of axioms in the data base. Subject to certain constraints to be described below, each literal of a generated clause gives rise to an entry on a list called the SPECLIST (for "specification list"). The list is so named since each literal entered onto the list is regarded as a specification for that set of axioms which, in the least restrictive case, contain a literal which is, or whose negation is, unifiable with the "spec literal". Each entry on the SPECLIST points at a list of one or more axioms which have been found to satisfy the specification. The axioms pointed to by the entries on the SPECLIST are candidates for generation. The SPECLIST is ordered so that "better" candidates come first; only candidates are available for generation.

As previously stated, that part of the Q* search algorithm which is concerned with generating clauses (i.e., adding to A-sets those clauses obtained from the base clause algorithm or from logically interacting clauses already generated) will be called the deduction algorithm. The deduction algorithm communicates with the base clause selection algorithm in two distinct ways; one way is direct, the other is indirect. Each time a clause is generated, that clause together with its merit (actually, a pointer to the clause and a pointer to its merit) are placed on a list called USPECS (for "unprocessed specs"); this is the indirect means of communication. Each time the deduction algorithm finds it appropriate to try to generate a base clause of merit better than or equal to M (or sometimes strictly better than M) the base clause routine BASEC (for "base clause") is called to attempt to satisfy the request; this is the direct means of communication. When a request is made for a base clause satisfying a given merit condition, all of the entries on USPECS, if any, are processed to produce zero or more entries for the SPECLIST. Once the SPECLIST has been updated, BASEC determines whether or not it can return the "best" clause. If it can, it does so; otherwise, it exits with failure. In the following two sections we describe the major functions performed by the base clause selection algorithm, namely, creating SPECLIST entries, and providing clauses to the deduction algorithm.

Creation of SPECLIST Entries

Each literal of the clauses found on USPECS, that is, of the clauses entered into A-sets by the deduction algorithm, is used as a basis for creating an entry on the SPECLIST. Each SPECLIST entry contains the following information:

a. The address of the literal upon which the SPECLIST entry is based; this literal will be called the *spec literal.*

b. The address of the clause containing the *spec literal;* this will be called the *spec clause.*

c. The address of the merit vector for the spec clause; this merit will be called the *spec clause merit.*

d. The address of the first axiom of a list of one or more axioms which are said to satisfy the specification; this first axiom will be called the *spec axiom.* The merit of the spec axiom will be better or equal to the merit of all other axioms in this list, where this measure of the merit is the same as that used by the deduction strategy.

e. The address of the merit vector for the spec axiom; this merit will be called the *spec axiom merit.*

f. The address of the merit vector giving the (predicted) upper bound merit of a resolvent of two clauses having merits equal to the spec clause merit and the spec axiom merit respectively; this merit will be called the *spec upper bound* merit.

g. A code number, called the *spec type*, which has the value 1 if the spec literal contains a constant, has the value three if the spec literal contains a function symbol, but no constants, and has the value five if the spec literal contains neither constants nor functions.

Each clause on USPECS is processed in turn, and is removed from USPECS when it has been completely processed. The exact nature of the processing will depend upon the inference system(s) in force. If set-of-support is in force either by itself, or in combination with other inference systems, or imbedded within an inference system as it is with SL-resolution, then any clause found on USPECS which does not have support will be removed from USPECS and not processed at all. If the clause does have support, then those of its literals which may validly be used in an inference (and which are not alphabetic variants of spec literals already contains in the SPECLIST) will be matched against the axioms. Axioms containing a literal which is opposite in sign to and unifiable with the (potential) spec literal will become the axioms pointed to by the SPECLIST entry. An axiom with the best merit will become the spec axiom. Note that with an inference system such as SL-resolution, only the selected literal will become the spec literal. Note also that with a suitably chosen data structure, the amount of searching for axioms containing literals which are opposite in sign to and unifiable with a given literal can be kept to a minimum.

For other than set-of-support based systems, unsupported clauses (i.e., axioms) can also be used to locate axioms. Furthermore, we must now be somewhat less selective in considering axioms and ignore the signs of the unifying literals. Thus, in non-set-of-support cases an axiom will become a candidate for generation if it contains a literal which unifies with a literal of a generated clause when the signs of the literals are ignored.

The process, then, for creating an entry for the SPECLIST involves finding the list of candidate axioms for the spec literal, assuring that one of best merit occurs at the front of the list, and filling in the remaining spec entries. The spec clause merit is obtained from USPECS, while the spec axiom merit, and the spec upper bound merit must be calculated. The spec type must also be determined by scanning the spec literal for constants and functions. In this regard, we have the capability which allows a user to specify that certain constants are to be treated as variables. Thus, certain non-specific, class-specifying types of constants which occur in a large subset of the data axioms can be prevented from getting the same privileged status given to more meaningful constants. In a data base about people, for example, such constants as "Male" or "Female" might be arguments in literals describing individuals.

Once the spec is completed it must be placed on the SPECLIST. As we noted earlier, the SPECLIST will be ordered so that the "best" spec axiom will be the next one delivered to the deduction algorithm. The algorithm orders specs by spec type. A spec of type 1 precedes one of type 3, etc. If a constant occurs in the query, this ordering has the effect of focusing the search to data axioms in which the query constants occur, and to general axioms which may interact with the constant carrying literals of the query and those of its successors which receive constants by instantiation as inferences are made. Among a group of specs of the same type we are experimenting with various sub-orderings. Among the properties for basing the sub-orderings, we have either tried or may try the following properties:

a. spec upper bound merit

b. spec clause merit

c. spec axiom merit

d. where the spec literal has a constant, order by how "far away" that constant is from the query constant (informally, e.g., if a resolution on a literal $\overline{P}(a,x)$ of a clause $\overline{P}(a,x) \vee Q(x,y)$ causes the literal $Q(b,y)$ to occur in the resolvent, (because the other literal taking part in the resolution was $P(a,b)$, then we might say that b is a distance of one away from a, and so on)

e. the number of potential spec literals which were found to be alphabetic variants of the given spec literal

f. the cluster distance (max or min) of the spec literal to the query predicates

g. the degree extracted from the semantic graph G_p that gives the number of predicates that co-occur with the predicate of the spec literal in the axioms

h. the ratio of constant arguments to non-constant arguments.

The "goodness" of the specs might also be defined as a linear combination of some of the above properties.

Notice that, in general, the spec *axioms* will not be in merit order (as used in the deduction strategy). This is a very significant departure from the Σ^* algorithm, and it is done at the expense of losing admissibility. But this sacrifice is only of theoretical importance since it would seem to be more important in practical applications to find *any solution* quickly rather than to carry out an exhaustive search and either run out of space or time and find no solution at all, or to find a simplest solution at great expense.

Handling Requests For Axioms

There are two modes of operation for the base clause request processor BASEC; these are the "constants on" mode and the "constants off" mode. The former mode is used whenever the query contains one or more constants, while the latter mode is used otherwise. We will describe the latter mode of operation first.

In the *constants off* mode, BASEC may receive a request for an axiom of merit better or equal to M . If the merit of any spec axiom satisfies this condition, the axiom is returned to the requesting program; otherwise, failure is reported.

Any time an axiom is removed from a SPECLIST entry, the axiom must be tagged as in use, since it may be pointed at by other entries (there is never more than one copy of any axiom), and the SPECLIST entry must be updated. If the removed axiom is the only one in the list of axioms corresponding to the entry, the entry is removed from the SPECLIST. Otherwise, an axiom in the list of best merit becomes the new spec axiom, the spec axiom merit is revised, a new spec upper bound merit is computed, and the revised SPECLIST entry is reinserted into an appropriate position of the SPECLIST.

Before a spec axiom is returned, its in-use status is checked to determine if it has already been generated. If it has, the axiom is removed and the SPECLIST is updated as described in the previous paragraph. This process will be repeated until the spec axiom of the first SPECLIST entry is not in use, or until the SPECLIST is empty.

In the *constants on* mode, BASEC can be called either to request an axiom of merit better than some merit value M , or to request an axiom whose merit is equal to M . In the former case, if the spec type of the first SPECLIST entry is greater than one, or if the spec axiom merit of this entry is worse than or equal to M , BASEC returns to the requesting program and reports failure. On the other hand, if the first spec axiom merit is better than M , and the spec type is equal to one, the address of the axiom, together with the address of its merit vector are returned to the requesting program.

If the request is for an axiom of merit equal to M , the following processing occurs. If the first SPECLIST entry has spec type equal to one, and if the spec axiom merit is equal to M , the address of the spec axiom is returned together with the address of its merit vector. If the first SPECLIST entry is of type one, and the spec axiom merit is worse than M and M is worse than a *predicted merit,* PM , (which will be described below), then a type three or type five spec axiom is sought whose merit is better or equal to M . (Only the first type three and the first type five entries need to be checked.) If such an axiom is found, it, together with its merit, are returned to the requesting program. In any other case, failure is reported to the requesting program.

In the constants on mode only, whenever a type one axiom is returned, its corresponding spec upper bound merit (i.e., the upper bound of the merit of a resolvent of two clauses whose respective merits are those of the spec axiom and the corresponding spec clause) will be saved in the merit vector PM if the current PM value is better ($<_d u$) than the upper bound merit. (PM is initialized with the zero vector when the system is presented with a query.) On subsequent calls to BASEC, PM is used to keep other axioms from being returned until the requested merit value, M , is worse than PM . This guarantees that the generated type one axiom which gave rise to the PM value has had a chance to interact logically with the corresponding spec clause. The result of such interaction may be (hopefully, it will be) the transference of one or more constants of the spec literal and/or spec axiom into literals of the resolvent. Such literals would then be used to cause additional axioms to be considered for generation.

4.5 The Deduction Strategy of MRPPS

4.5.1 Introduction

The deduction strategy used in MRPPS is closely modeled after the Σ^* algorithm of Kowalski [1970b]. The evaluation function f is defined by the user at the time of a proof. In particular, the user may specify m components for the feature vector F , as well as a linear transformation matrix W (which currently is an identity transform by default). If the transformed feature vector is $\hat{F} = (\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_i) = W \cdot F$, then the merit for a clause generated during a proof is $M = (\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_i)$. The merit orderings $\leq_d u$ used are defined by Equations 4.9 and 4.11 (refer back to section 4.2.1).

At the present time, the following five parameters are available to the user for selection as part of the evaluation function F :

1) clause length;
2) clause level;
3) minimum cluster distance;
4) maximum cluster distance; and,
5) clause complexity.

Various other measures may be incorporated in the future.

The basic idea of the deduction strategy is to *generate* (or to bring into an active status) clauses of best merit first, then next best merit, etc. The methods for generation used are inclusion of a clause from the set of base clauses, factoring, resolution, and paramodulation. Only clauses that have been generated (i.e., placed in an A-set) may interact to generate new clauses. An attempt is made to *estimate* or *predict* the merit of a resolvent or paramodulant from the merit of its two parents without explicitly forming the resultant clause. We therefore interact those clauses that we *predict* will yield the best merit successors. This is necessary since in general, we cannot give an *exact formula* for the merit of a resolvent or paramodulant but only an upper bound. If only length and level were used, however, an exact value would be known. Whenever a clause is generated by any means, its merit M is calculated (rather than predicted) and it is placed in a merit set designated by $A(\hat{f}_1,\hat{f}_2,\ldots,\hat{f}_m)$ or $A(M)$. In the following sections, these ideas will be explained in more detail and the three major subroutines of the algorithm will be described along with any differences they have with respect to the $\sum^*$ algorithm.

4.5.2 Subroutine NEXTMERIT

Since clauses are to be generated in increasing upper diagonal merit order, a routine NEXTMERIT is used to enumerate the merit components $\hat{f}_i$ in the order that the corresponding A-sets are to be filled. The sequence of merits generated is of course dependent upon whether the user has chosen Ordering-A or Ordering-B (Equations 4.9 and 4.11, respectively). In the current version of MRPPS, it is assumed that all features take on values of $0, 1, 2, 3,\ldots,n$ where n is a positive integer. Also, only a linear transformation of the form $\tau_m: E^m \to E^m$ is allowed, $m = 1,\cdots,5$, and W is the identity matrix. Thus, the components of $\hat{F}$ are actually equal to the corresponding components of F.

NEXTMERIT is called whenever a new A-set is to be filled by the FILL subroutine, at which time it generates the next merit vector $\hat{F}$ with respect to the ordering $\leq_d u$ chosen. FILL then attempts to form clauses of merit $M = {}_d u.\hat{F}$. As an example of possible sequences generated during a proof, consider the enumerations given in Figure 4. Assume for Ordering-B that $\hat{f}_1$ and $\hat{f}_2$ are "h-type" parameters and the $\hat{f}_3$ is a "g-type" parameter.

4.5.3 Subroutine FILL(M)

At the start of a proof, there exist no A-sets. When the query is input by the user on the teletype, the merit of each of the n clauses in the negation of the query is calculated, header cells for the corresponding A-sets are created, and the clauses are placed in the correct sets. If $M_0 = \min_d u \{M_i | M_i$ is the merit of $C_i \in \sim Q$, $i = 1 \cdots n\}$ then a call to $\text{FILL}(M_0)$ is made.

Each time FILL(M) is called, a check is made to see if any clauses are already in the A-set being filled. For instance, clauses from the negation of the theorem are found in this manner. If a clause is found, pointers to it and its merit are placed on the list USPECS (refer back to Section 4.4.2) and the clause is passed to RECURSE. If no clause is found, FILL attempts to find a base clause of merit $M' \leq_d u\ M$ by calling subroutine BASEC. As described earlier, BASEC processes the entries on USPECS, creates entries on SPECLIST and uses the literals on the SPECLIST to determine what clause to select from the data base. If BASEC returns a clause C of merit CM , pointers

$\hat{F} = (\hat{f}_1, \hat{f}_2, \hat{f}_3)$

Ordering-A	Ordering-B
000	000
001	001
010	010
100	100
002	002
011	011
020	101
101	020
110	110
200	200
003	003
012	012
021	102
030	021
102	111
111	201
120	030
201	120
210	210
300	300

FIGURE 4. ENUMERATIONS OF MERIT VECTORS FOR ORDERING-A AND ORDERING-B AND $\tau_3 \colon E^3 \rightarrow E^3$

to C and CM are placed in USPECS and RECURSE(C) is called.

If BASEC fails to find any suitable clause, resolution is attempted between clauses already generated. FILL tries to *predict* the merit of resolvents by performing calculations upon the merit components of two *prospective* parents. They are *prospective* since we need not look at individual clauses in making these predictions but *only at the merits of the sets containing the parents*. We can thus often eliminate entire sets of clauses from consideration as parents without forming any resolvents at all.

As noted previously, for many types of components it is impossible to predict exactly the merit for successors. Instead, we can find either lower bounds or upper bounds for the resulting merit. In attempting to FILL $A(M)$, it would be advantageous to be able to guarantee that no successor clause would *unexpectedly* yield merit $M' >_d u\ M$, for if this did happen, we would have to save it temporarily on a list and for the time being ignore it since we want to generate all those clauses of merit $\leq_d u\ M$ before clauses with worse merit than M. This can be accomplished by delaying the interaction of two clauses $C_1 \in A(M_1)$ and $C_2 \in A(M_2)$ until the *upper bound* for the merit of their successors *equals* M, the merit of the set being filled. That is, after explicit calculation of a resolvent, we may discover its merit to be better than or equal to M, but never worse.

Note that this is a departure from the technique used in the $\sum^*$ algorithm, since when only length and level are used, the merit of resolvents are known exactly. Another difference is that in the Q* algorithm, M_1 and/or M_2 may *equal* M rather than being strictly better than M as is the case in $\sum^*$. As an example, consider an evaluation function whose sole component is clause length. The merit of a resolvent of C_1 of length $M_1 = \ell_1$ and C_2 of length $M_2 = \ell_2$ is $\ell_1 + \ell_2 - 2$. If $M = 2$, then a possible combination of merits is $M_1 = M_2 = M = 2$. Thus, if no base clause is found with merit M , resolution is attempted between clauses previously generated and whose merit is *better than or equal to* M .

When no more clauses can be resolved, paramodulation may be attempted if the user selected paramodulation when the search was initiated. This process is very similar to that just described for resolution. In either case, if FILL forms a clause C , its merit CM is calculated, pointers to C and CM are placed on USPECS as described before, and RECURSE(C) is called.

As implied above, it is necessary to know the upper bounds between pairs of feature values. The following table gives the upper bounds between parameter f_1 of C_1 and f_2 of C_2 for resolution and paramodulation as used by both FILL and RECURSE.

parameter	resolution	paramodulation
length	$f_1 + f_2 - 2$	$f_1 + f_2 - 1$
maximum cluster	$\max(f_1,f_2)$	$\max(f_1,f_2)$
minimum cluster	$\max \delta$	$\max \delta$
functional complexity	$f_1 + f_2$	$f_1 + f_2$
level	$\max(f_1,f_2) + 1$	$\max(f_1,f_2) + 1$

Here $\max \delta$ is the maximum finite graph theoretical distance in the semantic graph for the data base.

The complete vector for an upper bound between clauses C_1 and C_2 is simply the vector whose components u_i are defined as u_i = upper bound(f_{1i},f_{2i}) for feature f_{1i} of C_1 and f_{2i} of C_2 .

4.5.4 Subroutine RECURSE(C)

During the following discussion of RECURSE(C) assume that M is the merit of the A-set currently being filled, C is the clause being recursed upon, and CM is its merit. RECURSE attempts to either (1) select appropriate clauses from the data base or to generate all possible successors of clause C by (2) factoring C , by (3) resolving C with a previously generated clause, or by (4) paramodulating C with another clause. In general, any clause generated by (1) must have merit $N <_d u\ CM$ whereas those generated by (2), (3) or (4) must have merit $N \leq_d u\ M$. Thus, RECURSE proceeds in four stages as described below.

The first stage determines whether there are data base clauses of merit $N <_d u\ CM$. BASEC is called and the literals on SPECLIST are referred to for guidance in selecting the "best" clause to enter into an A-set. This operation is done during RECURSE (rather than only in FILL as is the case in $\sum^*$) in order to interact base clauses as soon as possible after the literals permitting such an interaction are placed on SPECLIST.

If a clause D is found by BASEC, the merit DM of D is computed, D is placed in an A-set, pointers to C and CM are saved on a stack, pointers to D and DM are placed on USPECS, and RECURSE is called on D . We have therefore decided to delay recursing on C temporarily in favor of D since this seems more promising at the moment.

Note that this operation is not permitted in the $\sum^*$ algorithm which requires that base clauses may only be entered into an A-set during FILL but not during RECURSE. However, between the time a clause C is generated in FILL, is recursed upon, and control returns to FILL again, many literals may have been placed on SPECLIST because of the inferences generated during this time. Thus in $\sum^*$, these literals are *ignored* until FILL is re-entered, and we have therefore unnecessarily delayed the interaction of data base clauses with clauses already generated. We do not know how much efficiency will be gained by calling BASEC during RECURSE and it is thus available as an option to the experimenter.

If no more clauses can be generated by Stage 1 of RECURSE, Stage 2 attempts to factor C . Factoring is always performed without any reference to merits since normally, a factor has better merit than its parent. If, however, a factor D of merit $>_d u$ M is formed, RECURSE(D) is not called but deferred until a future call to FILL. Else, if a factor D is formed using literals $\ell_1 \in C$ and $\ell_2 \in C$, the merit DM of D is calculated, D is entered into the correct A-set, pointers to C , ℓ_1 , ℓ_2 and CM are stacked, pointers to D and DM are placed on USPECS, and RECURSE(D) is called. If no more factors can be formed, Stage 3 is entered.

Stage 3 finds all resolvents of C with clauses C' such that Upperbound(Merit(Resolvent(C,C'))) $\leq_d u$ M . When such a clause D is formed using literals $\ell_1 \in C$ and $\ell_2 \in C'$, its merit DM is calculated, it is placed in the correct A-set, pointers to D and DM are placed on USPECS, pointers to C , C' , ℓ_1 , ℓ_2 and CM are stacked and RECURSE(D) is called.

If the above upperbound is N $>_d u$ M , the resolvent is not generated since a future call to FILL(N) will generate the clause. Recall that RECURSE of $\sum^*$ formed resolvents of merit N $<_d u$ M . This was because when only length and level are used as components of the evaluation function f , it is impossible to form a clause of merit equal to

M , whereas in Q* it is quite possible (for example, consider only length as a parameter).

When Stage 3 fails to find any more resolvents with C , paramodulation may be attempted in Stage 4, if desired. This process is similar to that of Stage 3 except that when a paramodulant D is formed between D and C' , pointers to C and C' and the terms from each clause used in making the substitutions must be stacked before recursing. If all four stages cannot generate any more clauses, the stack is popped and control is returned to the previous level of RECURSE which continues to find more base clauses or to find more successors to C , depending on what stage has resumed execution.

Although selecting base clauses during RECURSE may help optimize the order of clause generation, it also can cause the generation of duplicate clauses. For instance, let clause C be stacked because clause C' has been found by BASEC when RECURSE(C) is called. Then RECURSE(C') may cause D = Resolvent(C',C) to be formed. When C is later unstacked and recursed upon, the same clause D = Resolvent(C,C') may again be formed. This duplication can be avoided either by checking for alphabetic variants after forming inferences or by some bookkeeping procedure. Since alphabetic variants are normally eliminated, Q* uses this method.

4.5.5 Halting Conditions for the Deduction Strategy

It should be noted that under certain conditions, the deduction strategy can detect that no more inferences can be produced. If no null clause has been produced up to that point, the search may be halted and the query may be considered unanswerable or false. This condition may be detected as follows. At any given point of a proof, WM points to the worst merit non-empty A-set. The merit of the worst merit resolvent (or paramodulant) between $C_1 \in A(WM)$ and $C_2 \in A(M)$ where $M \leq_d u\ WM$ is calculated and is pointed to by M' . Any time a clause of merit $N >_d u\ WM$ is formed, $WM \leftarrow N$ and M' is recalculated. If no null clause has been found after FILL(M') is complete, the search is halted because no more inferences can be formed.

4.6 The Q* Search Algorithm - The Search Strategy of MRPPS

4.6.1 Introduction

This description of the Q* algorithm ties together the deduction strategy and the base clause selection strategy of MRPPS. Details covering SPECLIST entries were discussed in Section 4.4. In particular, keep in mind that each time a base clause is selected, the SPECLIST is reordered, spec literals are removed if necessary, a new spec axiom is found, and the spec upperbound merit is recalculated. Also, the following algorithm does not include provisions for paramodulation since the control mechansim is identical to resolution. The algorithm employs a pointer variable n to point at the current clause. The vectors M , N , WM , M' , and PM store merit values. The notation $PM \leftarrow \vec{0}$ means that all components of vector PM are set to 0 . Merit M_1 is better or equal to merit M_2 , denoted $M_1 \leq M_2$, according to the merit orderings described in Section 4.2.1. The notation $n \leftarrow$ FILTER(n) means that clause n is passed to a "filtering" routine that eliminates tautologies, subsumed clauses, and alphabetic variants. BASER = 1 means that BASEC is called during RECURSE. With these comments in mind, the algorithm will now be given.

4.6.2 The Q* Algorithm

[1] [Initialize]. Set $PM \leftarrow \vec{0}$, $WM \leftarrow \vec{0}$, $M' \leftarrow \vec{0}$, FILLMODE $\leftarrow 1$, STACK1 $\leftarrow \Lambda$, STACK2 $\leftarrow \Lambda$, and STACK3 $\leftarrow \Lambda$.

[2] [Enter Query Clauses]

[2.1] [Test For Constants In Query]. If a constant appears in a clause of $\{\bar{Q}\}$, set CONST $\leftarrow 1$, otherwise set CONST $\leftarrow 0$.

[2.2] [Enter Query Clauses Into Merit Sets]. Enter each clause of $\{\bar{Q}\}$ into an appropriate merit set. If the null clause is in $\{\bar{Q}\}$, print message to this effect and stop, otherwise, set M to the merit of the best merit clause in $\{\bar{Q}\}$, and set FILLMODE $\leftarrow 1$.

[3] [Fill Merit Set A(M)]. If FILLMODE = 1 go to [3.1], otherwise, if FILLMODE = 2 go to [3.2], otherwise go to [3.3.3].

[3.1] [Fill By Scanning Merit Set]. If there are no clauses in A(M) to which RECURSE has *not* been applied, set FILLMODE $\leftarrow 2$, and go to [3.2]; otherwise, let n point at the first such clause, mark the clause as having been explored by RECURSE, set N $\leftarrow$ M, and go to [5].

[3.2] [Fill With Base Clause]. If CONST = 1, go to [3.2.1], otherwise go to [3.2.2].

[3.2.1] [Constants Are Turned On]. If a constant occurs in the first spec literal and the associated spec axiom has merit = M, continue; otherwise, go to [3.2.1.1]. Let n point at the spec axiom. set N $\leftarrow$ M, remove axiom n from the SPECLIST entry and enter it into merit set A(N). Calculate the predicted merit, m_p, of a resolvent of two clauses whose merits are equal to the merit of the spec clause and the merit of the spec axiom, respectively. If $PM < m_p$, set $PM \leftarrow m_p$. Go to [5].

[3.2.1.1] If a constant occurs in the first spec literal and the associated spec axiom has merit $> M$ and $M > PM$, continue; otherwise go to [3.3]. Is there a SPECLIST entry whose spec literal contains no constants? If not, go to [3.3]. Otherwise, if the spec axiom of the first such entry has merit $\leq M$, let n point at this axiom, set $N \leftarrow MERIT(n)$, remove clause n from the SPECLIST entry, enter n into A(N) , and go to [5].

[3.2.2] [Constants Are Turned Off]. If the spec axiom of any SPECLIST entry has merit $\leq M$, call the clause n , set $N \leftarrow MERIT(n)$, remove clause n from the SPECLIST entry, enter n into merit set A(N) , and go to [5].

[3.3] [Fill By Resolution]. Set FILLMODE $\leftarrow 3$.

[3.3.1] [Find Merit Sets]. Find the next pair of merit sets $A(M_1)$, $A(M_2)$ such that $\text{Upperbound}(M_1,M_2) = M$. If no such pair was found and $M' > M$, go to [4]; else if no pair was found and $M' = M$ then stop and declare the query to be unanswerable; otherwise go to [3.3.2].

[3.3.2] [Find Clauses in $A(M_1)$ and $A(M_2)$]. Let C_1 and C_2 be the next pair of clauses in $A(M_1)$ and $A(M_2)$ respectively. If no such pair was gound go to [3.3.1]; otherwise go to [3.3.3].

[3.3.3] [Find Resolvents of C_1 and C_2]. Let n point to the next resolvent of C_1 and C_2. If no such resolvent can be found, go to [3.3.2]; otherwise, set $n \leftarrow$ FILTER(n) . If n should be eliminated, go to [3.3.2]. Else, let $N =$ MERIT(n) , enter clause n into merit set $A(N)$, and go to [5].

[4] [Find Next Merit Set]. Set $M \leftarrow$ NEXTMERIT(M) , FILLMODE $\leftarrow 1$, and go to [3].

[5] [Create SPECLIST Entries]. For each literal in clause n , as permitted by the inference system in force, create a SPECLIST entry as described in Section 4.4.

[6] [Test For Null Clause]. If $N > WM$, $WM \leftarrow N$, and recalculate M' to be the merit of the worst merit resolvent between $C_1 \in A(WM)$ and $C_2 \in A(M)$ such that $M \leq WM$. Else, if clause n is the null clause, print a message that a refutation has been found, and stop.

[6.2] [Test For Axioms On Merit < N]. If CONST = 0 or if no constant occurs in the first spec literal, or if the user has set BASER $\leftarrow 0$, go to

[6.3]. If the first spec axiom has merit $\geq N$, go to [6.3]; otherwise, let n_1 point at the spec axiom and remove the axiom from the SPECLIST. Calculate the predicted merit, m_p, of a resolvent of two clauses whose merit are equal to those of the spec clause and the spec axiom, n_1, respectively. If $PM < m_p$, set $PM \leftarrow m_p$. Set OP ← "Axiom", and go to [6.4].

[6.3] [Infer Clause of Merit $\leq M$].

[6.3.1] [From Factors]. Is there another factor n_1 of clause n such that $\text{MERIT}(n_1) \leq M$? If not, go to [6.3.2]; otherwise, set $n_1 \leftarrow \text{FILTER}(n_1)$. If n_1 should be eliminated, go to [6.3.1]. Else, OP ← "Factor", and go to [6.4].

[6.3.2] [Find Merit Set]. Find the next merit set, $A(M')$ such that $\text{Upperbound}(N,M') \leq M$. If no such set was found, go to [6.5]; otherwise, go to [6.3.2.1].

[6.3.2.1] [Find Next Clause]. Let C_1 be the next clause in $A(M')$. If no such clause is found, go to [6.3.2]; otherwise, go to [6.3.2.2].

[6.3.2.2] [Form Resolvents]. Let n_1 point to the next resolvent of n and C_1. If no such resolvent can be formed, go to [6.3.2.1]; otherwise, set $n_1 \leftarrow \text{FILTER}(n_1)$. If n_1 should be eliminated, go to [6.3.2.1]. Else, set OP ← "Resolvent" and go to [6.4].

[6.4] [Stack The Current Clause]. STACK1 ← n , STACK2 ← N , STACK3 ← OP , set N ← MERIT(n_1) , enter clause n_1 into A(N) , set n ← n_1 , and go to [5].

[6.5] [Pop The Stack]. If STACK1 = Λ , go to [3]; otherwise, n ← STACK1 , N ← STACK2 , and OP ← STACK3 . If OP = "Axiom" , go to [6.2]; otherwise, if OP = "Factor" , go to [6.3.1]; otherwise, go to [6.3.2.2].

4.6.3 Completeness, Admissiblity, and Optimality of Q*

Search algorithms are said to be complete if all nodes in the search space will be visited (or generated) at some stage of the search. Such search algorithms might be said to be exhaustive. In terms of a theorem proving problem, a complete search strategy will generate, at some stage of the search, every clause C such that C is either in, or is deducible from a starting set of clauses (where deductions are made using some given fixed set of inference rules).

The Q* algorithm is not complete in the above sense, since in general, the algorithm will fail to generate portions of the search space. In particular, the Q* algorithms will not generate axioms whose predicates do not occur with query predicates in a connected component of the semantic graph G_p .

The fact that the Q* algorithm, in general, does not exhaustively generate the entire search space is *not* a negative aspect of the algorithm. Indeed it is a positive aspect. However, we must assure ourselves that the Q* algorithm is guaranteed to generate the null clause whenever the null clause is a node in the search space (i.e., whenever it would be generated by a complete search strategy). If such is the case, we will call the search strategy *refutation complete.* Using a refutation complete strategy for a question answering system assures us, at least theoretically, that if a question can be answered positively, then the search algorithm will find the answer.

The method of making axioms candidates for generation selects those axioms which contain a literal which will unify (ignoring literal signs) with a literal of a generated clause. Initially, the query literals are used for this purpose. Subsequently, the literals of inferred clauses, and, when the inference system is not employing set-of-support, the literals of generated axioms will be used for making axioms candidates for generation. Thus, the only axioms which will ever become candidates are those which are a finite cluster distance from the query (see Section 4.3.4). Any other axioms cannot logically interact with the candidate axioms or the query clauses, or successors of these clauses, and hence, could never take part in a refutation since the set of axioms is assumed to be consistent. If a refutation required the use of a lemma, then the literals of the lemma would descend from possibly more general instances of these literals in the ancestor axioms. Hence these axioms would become candidates.

It is evident that any axioms required in a refutation will become candidates for generation at some stage of the search. The remaining question, then, is whether all of the candidates can be generated. The base clause algorithm gives preference to those axioms which have become candidates because of generated clause literals containing constants. Since there are only a finite number of axioms in the starting set, these will eventually be exhausted, and hence, at some stage the other candidate axioms will be generated. Since the deductive search strategy is exhaustive, and since all of the axioms which could possibly be of use will eventually be turned over to that strategy, the Q* algorithm is refutation complete.

A search algorithm is said to be *admissible* if it is guaranteed to find the shortest path to a goal node. In the case of a search algorithm for the theorem proving problem, an admissible algorithm would be guaranteed to find a shortest proof (refutation). The Q* algorithm is *not* an admissible algorithm for theorem proving. The algorithm may overlook a shortest proof because in order to find that proof it would have to generate axioms to interact with general literals in the negation of the theorem (query) before it generated axioms to interact with more specific literals. For example, suppose that the set S of axioms contains the clauses,

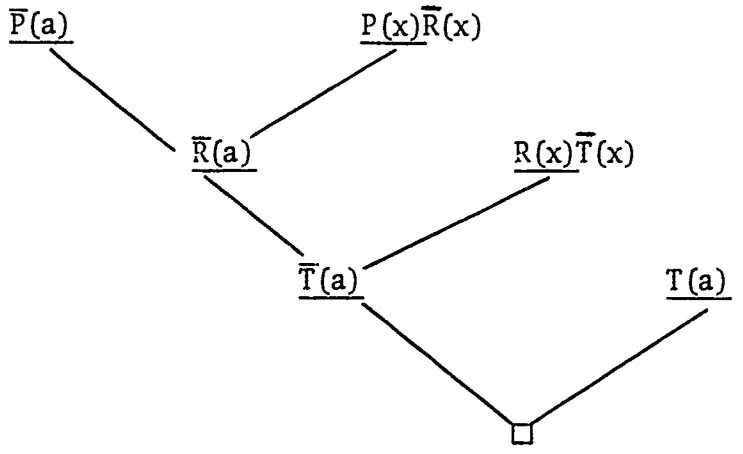
$$S = \{P(x)\overline{R}(x),\ R(x)\overline{T}(x),\ Q(x)\overline{S}(x),\ T(a),\ S(b)\}$$

and that the negation of the theorem contains the two clauses,

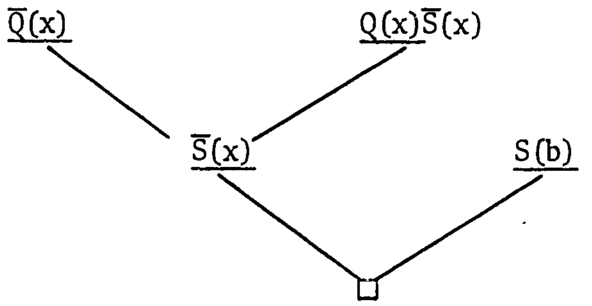
$$\{\sim Q\} = \{\overline{P}(a),\ \overline{Q}(x)\}\ .$$

The Q* algorithm would first generate $\overline{P}(a)$ and $\overline{Q}(x)$. When these are generated, entries would be made on the SPECLIST for the literals $\overline{P}(a)$ and $\overline{Q}(x)$. The candidate axioms for $\overline{P}(a)$ would precede those for $\overline{Q}(x)$. Thus, the first axiom which would be generated is $P(x)\overline{R}(x)$. As soon as it is generated, it would be used in an inference, resulting in the generation of the clause $\overline{R}(a)$. The SPECLIST entry created for this literal would again precede that for $\overline{Q}(x)$, and so the axioms

$R(x)\overline{T}(a)$ which would, in turn, be used to enable the axiom $T(a)$ to be generated, and immediately thereafter, the null clause would be generated. Thus, the Q* algorithm will find the refutation



However, a shorter proof is:



Although the Q* algorithm fails to be admissible, we believe that this will not be a significant disadvantage of the algorithm. If the heuristics prove powerful enough to bring this sort of deductive power to bear on a large enough class of QA problems in a practical setting, then indeed, this will be a small price to pay.

An admissible search algorithm is said to be *optimal* if there is no other "comparable" admissible search algorithm which generates fewer nodes. Since the Q* algorithm is not admissible, there is nothing to be said about its optimality.

5. Conclusions and Future Directions

The main direction of this research is to explore various deductive search mechanisms for question answering systems. The research has been restricted to this scope primarily because of limited funding. It is hoped that in the future we may extend the work to develop a full QA System. Experiments could then be conducted with all aspects of such systems rather than only the deductive mechanism. In particular, it will be important in the near future to consider large data bases. However, for the present time, we plan to emphasize experimental work with the current system, investigating small data bases and incorporating modifications as necessary.

There are several modifications to the existing system that are in progress. First, the inference mechanism is being expanded to include A-ordering (Darlington [1969], Kowalski, Hayes [1969], Slagle [1967]). Darlington [1969] has speculated that A-Ordering is a promising refinement of resolution for QA Systems, although this fact must be supported by more experimentation than currently exists in the literature. In addition, since paramodulation is available to the user as an option, experimentation will be performed with it in conjunction with the Q* search strategy. We would like to discover whether the heuristics currently available are sufficient to allow paramodulation to be used efficiently.

In the area of search strategies MRPPS currently allows the user to select

1) the type of generalized merit ordering to be used (i.e. Ordering-A or Ordering-B),
2) which parameters to treat as "g-type" and which to treat as

"h-type",

and 3) which clause feature to assign to each component of the feature vector F.

It is planned that the Q* algorithm will be modified so that weight matrices other than an identity matrix will be allowed as will as an ordering corresponding to that of Pohl (equation 4.14). We presently do not know whether the generalized merit ordering will be superior than an evaluation function $f(n) = g(n) + h(n)$ with only two components. Additional experiments are required in this regard. In addition, new combinations of heuristics need to be devised for the search strategy.

We believe that the base clause selection strategy is crucial to a practical QA system and it is in this component that semantics can be utilized most effectively. Various semantic considerations have been developed and will be incorporated into the system, although all of these ideas have not been described in this report.

Experimentation will be performed using the current data base that consists of genealogical data about Eskimos. Typical data might be a certain Eskimo's age, name, husband or wife and the names of his or her children. Since this data base is somewhat limited with respect to the type of questions that can be asked, it is planned that other large data bases will be developed that are stored on auxiliary storage rather than in core. This would be one step towards our long-range goal of implementing a *large-scale* question-answering system of *practical* utility. However, this is not within the range of the current funding.

Some promising results have already been obtained using the Eskimo data base. In fact, using only length and level as heuristics, fairly deep proofs have been obtained efficiently with a data base of 300 unit data clauses and 85 general axioms. In particular, questions such as "who is Joe's mother's motherinlaw?" that require six inference steps involving the use of general axioms, have been answered in about one second. Questions requiring *only* data facts in order to be answered have required about .1 second. It is hoped that these times can be shortened in future versions of MRPPS and that similar results may be obtained in the future using a much larger (and more realistic) data base.

Several steps have been taken towards developing an entire QA System that have not been described in this text. The user can enter his own data base provided it fits in the available space and is expressible as a set of clauses. In addition, an algorithm to translate wff's in first-order predicate calculus into clause form has been implemented and integrated with the overall system, as has an answer-extraction algorithm based on the work of Luckham and Nilsson [1970].

Although we have outlined some of the plans we have concerning future directions for research in QA Systems at Maryland, there are several conclusions that may be drawn from the current work. MRPPS is designed so that the data base, the inference mechanisms and the search strategy can be regarded as separate but interacting entities. We believe that the inference mechanisms may be regarded as *primitive* routines (or operators) that logically deduce new clauses. They should in no way be considered search strategies that select the "most appropriate" inference to make. This latter decision must be made by a high-level control mechanism that references extensive semantic information.

As a step towards building such a control mechanism, we have utilized some semantic information about the data base by the use of cluster heuristics in an evaluation function $f(n) = g(n) + h(n)$ and other heuristics used in the base clause selection strategy. (This is in contrast to the purely syntactic considerations of the inference mechanism.) Some preliminary experiments indicate that both of these techniques are promising. At the same time, it seems clear that much more semantic information is needed in order to enable efficient answering or questions. This should be in the form of advice to the search strategy about which derivation paths are most likely to succeed or which will probably never succeed. This advice could be the result of human insight about the problem to be solved (such as in PLANNER, Hewitt [1970]) or could be generated by the system based upon past experience with analogous problems.

One way that this could be accomplished is to store proof *schemata* of theorems that have been previously proven. Each schema would consist of several possible derivation paths, each specifying (among other things) recommended axioms and their corresponding weights indicating the relative probability that each axiom has of aiding in the search. These schemata would be referenced continuously during a proof and would direct the proof to a large extent. Some limited capabilities currently exist in the system which are able to do this. However, we would like to try to achieve a more sophisticated mechanism and attempt to interface it with the Q* algorithm. In any event, some more "informed" type of planning mechanism is certainly necessary and this problem is currently being addressed.

In general, MRPPS provides us with a flexible interactive system in which the parameters may be varied by the user. Hopefully, we will be able to gain insight into:

(a) inference mechansims,

(b) heuristic measures,

(c) and semantic considerations

to help guide a QA System in answering questions. Work with a data base should permit experiments to be conducted and reported upon in subsequent reports.

REFERENCES

(1) Allen, J. and Luckham, D. "An Interactive Theorem-Proving Program" In: Meltzer, B. and Michie, D. (Eds.), Machine Intelligence 5, American Elsevier, New York, 1970, 321-336.

(2) Ash W. and Sibley, E. "TRAMP, an Interpretive Associative Processor with Deductive Capabilities" Proc ACM 23rd National Conference, Brandon Systems Press, Princeton, N.J., 1968, 143-156.

(3) Bobrow, D.G., Fraser, J.B. and Quillian, M.R. "Automated Language Processing" In: Cuadra, C.A. (Ed.), Annual Review of Information Science and Technology, Interscience, New York, 1967, Vol.2, 161-186.

(4) Chang, C.L. "The Unit Proof and the Input Proof in Theorem Proving" J.ACM 17, 4(Oct., 1970), 698-707.

(5) Coles, L.S., Colwell, W.T., Jones, J.H., Whitby, O.W. and Younker, L. Design of a Remote-Access Medical Information Retrieval System. Final Report on Contract NLM 69-13 for the National Library of Medicine (SRI, 1969).

(6) Darlington, J.L. A COMIT Program for the Davis-Putnam Algorithm. Res. Lab. Electron., Mech. Translation Group., MIT, Cambridge, Mass., 1962.

(7) Darlington, J.L. "Theorem Proving and Information Retrieval" In: Meltzer, B. and Michie, D. (Eds.) Machine Intelligence 4, American Elsevier, New York, 1969, 173-181.

(8) Darlington, J.L. "A Partial Mechanization of Second-Order Logic" In: Meltzer, B. and Michie, D. (Eds.), Machine Intelligence 6, Edinburgh U. Press, 1971, 91-100.

(9) Davis, M. and Putnam, H. "A Computing Procedure for Quantification Theory" J. ACM 5, 7(July, 1962), 394-397.

(10) Feldman, J.A. and Rovner, P.D. "The LEAP Language and Data Structure" Proc. IFIPS Congress (August, 1968).

(11) Feldman, J.A., Low, J.R., Swinehart, D.C., and Taylor, R.H. "Recent Develpments in SAIL-An ALGOL-based Language for Artificial Intelligence", Proc. FJCC, AFIPS Press, Montvale, N.J., 1972, 1193-1202.

(13) Garvey, T.D. and Kling, R.E. Users Guide to QA 3.5 Question-Answering System. A.I. Technical Note 15, Stanford Research Institute, 1969.

(14) Green, C.C. and Raphael, B. "The Use of Theorem Proving Techniques in Question-Answering Systems" In: Proc.-1968 ACM National Conference, Brandon Systems Press, Princeton, N.J., 1968, 169-181.

(15) Hart, P., Nilsson, N. and Raphael, B. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths" IEEE Trans. Sys. Sci. Cybernetics Vol. SSC-4, No. 2, 100-107.

(16) Hewitt, C. Planner, MAC-M-386, Project MAC, MIT, 1970.

(17) Kowalski, R. Studies in the Completeness and Efficiency of Theorem-Proving by Resolution. Ph.D. Thesis, U. of Edinburgh, 1970a.

(18) Kowalski, R. "Search Strategies for Theorem Proving" In: B. Meltzer and D. Michie (Eds.), Machine Intelligence 5, American Elsevier, New York, 1970b, 181-200.

(19) Kowalski, R. and Hayes, P.J. "Semantic Trees in Automatic Theorem Proving" In: Meltzer, B. and Michie, D. (Eds.) Machine Intelligence 4, American Elsevier, New York, 1969, 87-101.

(20) Kowalski, R. and Kuehner, D. "Linear Resolution with Selection Function", Artificial Intelligence 2,3 (1971), 221-260.

(21) Kuno, S. "Computer Analysis of Natural Languages" In: J.T. Schwartz (Ed.), Proc. of Symposia in Applied Mathematics XIX, American Mathematical Society, Providence, R.I., 1967, 52-110.

(22) Levien, R.E., and Maron, M.E. Relational Data File: A Tool for Mechanized Inference Execution and Data Retrieval. The RAND Corporation, RM-4793-PR, 1965.

(23) Loveland, D.W. "A Linear Format for Resolution" IRIA Symp. on Automatic Demonstration, Versailles, France, 1968.

(24) Loveland, D.W. "Theorem Provers Combining Model Elimination and Resolution" In: Meltzer, B. and Michie, D. (Eds.), Machine Intelligence 4, American Elsevier, New York, 1969, 73-86.

(25) Luckham, D. The Ancestry Filter Method in Automatic Demonstration. A.I. Project Memo, Stanford University, Stanford, Calif., 1968.

(26) Luckham, D. and Nilsson, N. "Extracting Information from Resolution Proof Trees" Artificial Intelligence, 2, 1(1972), 27-54.

(27) Marill, T. Relational Structures Research. Computer Corporation of America, Final Report on Contract DA18-119-AMC-03409(X), F/R18-119-36-00326(X), 1967.

(28) Meltzer, B. "Theorem Proving for Computers: Some Results on Resolution and Renaming" Computer J. 8, (Jan., 1966), 341-343.

(29) Meltzer, B. "Prolegomena to a Theory of Efficiency of Proof Procedures" In: Findler and Meltzer (Eds.) Artificial Intelligence and Heuristic Programming, Edinburgh Univ. Press, 1971.

(30) Minker, J. and Sable, J.D. Relational Data System Study. Auerbach Corporation, AUER-1776-TR-1, 1970.

(31) Montgomery, C.A. "Automated Language Processing" In: Cuadra, C.A. (Ed.), Annual Review of Information Science and Technology, Encyclopedia Brittanica, Inc.. Chicago, 1969, Vol. 4, 145-174.

(32) Montgomery, C.A. "Linguistics and Information Science" J. ASIS. 23, 3(May-June, 1972), 195-219.

(33) Norton, L.M. "Experiments with a Heuristic Theorem-Proving Program for Predicate Calculus with Equality" Artificial Intelligence 2, 3(1971), 261-284.

(34) Pohl, I. "First Results on the Effect of Error in Heuristic Search" In: Meltzer, B. and Michie, D. (Eds.) Machine Intelligence 5, American Elsevier, New York, 1970, 219-236.

(35) Robinson, G.A. and Wos, L. "Paramodulation and Theorem Proving in First-Order Theories with Equality" In: Meltzer, B. and Michie, D. (Eds.) Machine Intelligence 4, American Elsevier, New York, 1969, 135-150.

(36) Robinson, J.A. "A Machine Oriented Logic Based on the Resolution Principle" J.ACM 12, 1(Jan., 1965a), 23-41.

(37) Robinson, J.A. "Automatic Deduction with Hyper-Resolution" Int. J. Comput. Math. 1(1965b), 227-234.

(38) Rulifson, J.F. QA4 Programming Concepts. Artificial Intelligence Technical Note 60, Artificial Intelligence Center Stanford Research Institute, Menlo Park, Calif., 1971.

(39) Salton, G. "Automated Language Processing" In: Cuadra, C.A. (Ed.) Annual Review of Information Science and Technology. Encyclopedia Brittanica, Inc., Chicago, 1968, vol.3, 169-199.

(40) Simmons, R.F. "Answering English Questions by Computer: A Survey" Comm. ACM, 8, 1(Jan., 1965), 53-70.

(41) Simmons, R.F. "Natural Language Question-Answering Systems: 1969" Comm. ACM 13, 1(Jan. 1970), 15-30.

(42) Slagle J.R. "Automatic Theorem Proving with Renamable and Semantic Resolution" J. ACM 14, 4(Oct., 1967), 687-697.

(43) Sussman, G.J. and McDermott. "From PLANNER to CONNIVER -a Genetic Approach" Proc. FJCC, AFIPS Press, Montvale, N.J., 1972, 1171-1179.

(44) Travis, L., Kellogg, C. and Klahr, P. Inferential Question-Answering: Extending Converse. Unpublished Draft, 1972.

(45) Wos, L.T., Carson, D.F. and Robinson, G.A. "The Unit Preference Strategy in Theorem Proving" Proc FJCC, Thompson Book Co., New York, 1964, 615-621.

(46) Wos, L. T., Carson, D. F. and Robinson, G. A. "Efficiency and Completeness of the Set-of-Support Strategy in Theory Proving", J.ACM 12, 4 (Oct, 1965), 536-541.