AN ARCHITECTURE FOR KNOWLEDGE BASED DEDUCTION

Arthur J. Nevins

Department of Computer Information Systems
Georgia State University
Atlanta, Georgia 30303

ABSTRACT

This paper describes the design of a general purpose deduction engine for use in expert systems. It represents an extension of a natural deduction theorem prover that has eliminated the negation symbol in favor of certainty factors and has an improved method of splitting problems into subproblems. Formulas are object-centered and are stored at nodes which are designed to control and provide sharper focus to the search. These nodes either are pointed to by words in a dictionary or are created by the deduction engine in its attempt to solve a specific problem.

1. Introduction

This paper describes the design of a general purpose deduction engine for use in expert systems. It goes beyond MYCIN [8], [9] in utilizing more of the power of predicate logic, while at the same time, its treatment of certainty factors is not intertwined with a particular search strategy (such as the backward chaining gathering of evidence for certainty factor evaluation used by MYCIN).

The present system employs natural deduction [1], [3], [A] which offers some advantages over resolution [7] in that 1) it does not convert to clause form, thereby making its representation closer to that used by humans and saving potentially heuristic information, and 2) it is easier to split a problem into subproblems since expressions do not get "multiplied out" during the conversion to clause form.

The system maintains a dictionary of words, each word pointing to one or more subject areas, with each subject area containing formulas relevant to the subject. This enables it to process a large number of formulas by focusing on subject areas relevant to the query. The system has a deduction module whose primary purpose is to obtain answers to queries and will have a knowledge acquisition module whose purpose is to add new formulas to the data base and update the dictionary. This paper is concerned primarily with the deduction module.

This research was supported in part by the National Science Foundation under Grant DCR-8408389.

2. Definitions

A constant is either a number, a character string, or an executable function symbol. A term is either a constant, a variable, or a list of terms. An object is a property list (A(1):t(1),.....,A(n):t(n)) where each A(*i*) is a constant (called an attribute) and each t(*i*) is a term. There is a special attribute called NAME which identifies the object. The attributes A(1),A(2),.....,A(n) of an object are ordered, with A(1) always being the attribute NAME.

We employ the logical connectives \/ for "OR", /\ for "AND", and --> for "IMPLIES" but do not refer directly to the quantifiers "FOR ALL" and "THERE EXISTS" since well known procedures exist for their removal [4]. Instead of introducing uncertainty by adding an extra parameter to specific predicates [2], we find it easier to dispense with the negation symbol and use in its place real numbers between -1 and +1 which will be called certainty factors.

An elementary formula is either an object or an expression of the form P(O) where P is a certainty factor and O is an object. A procedural formula is a list whose first element is an executable function symbol and whose remaining elements are either terms or procedural formulas. A formula is either an elementary formula, a procedural formula, or an expression of the form P(F), F\/G, F/\G, or F-->G where F and G are formulas and P is a certainty factor. A production rule is a formula of the form F-->G where F is an elementary formula. Each formula is assigned to a group of formulas called a node. The nodes of the initial data base are pointed to by words in the dictionary. The words in the dictionary are called key words.

If application of formula F, as an input to an inference rule, produced as output the formula G, then F is said to be a parent of G and G is said to be a child of F. Formula F is an independent ancestor of G if (1) either F is a production rule or F has no parents and (2) either F is a parent of G or F is a parent of some formula H and H is an independent ancestor of G. Node F is an ancestor of node G if either PNODE:F appears on node G or there is some node H such that PNODE:H appears on node G and F is an ancestor of H. F is a descendant of G if G is an ancestor of F.

A formula-frame is a property list of attribute:value pairs, including the attribute DEFINITION whose associated value is a formula. For example, [DEFINITION:F, NODE:N, PARENTS:P, CHILDREN:C] *is* a formula-frame that is targeted for node N, and has formula F for its definition, and a list of parents P and a list of children C. Although each formula is stored in the data base as part of a formula-frame, we usually will just use the word "formula" as it can be tedious to keep making this distinction.

A <u>substitution</u> *is* a property list [x(1):t(1). . . . . ., x(n):t(n)] where x(1), ... x(n) are variables and t(l), ... t(n) are terms. If S is a substitution [x(l):t(1), ... x(n):t(n)] and F is a formula, then S(F) is the formula which results from substituting t(i) for x(i) in formula F, for each i = 1, ... , n.

Two elementary formulas PI(01) and P2(02) can be fully (partially) <u>unified</u> by substitution S if PI < 0, P2 > 0 and object 01 (called the negative object) can be fully (partially) unified with object 02 (called the positive object). Substitution S will fully (partially) unify negative object 01 with positive object 02 if for every attribute:term AI:tl in 01 (or, for partial unification, at least two attribute:terms in 01 including attribute NAME) there exists an attribute:term A2:t2 in 02 such that A1 - A2 and either (1) S(tl) - S(t2) or (2) negative object t1 is fully
unified with positive object t2 by substitution S, or (3) S(tl) *is* of the form F(r) where F *is* an executable function symbol and execution of F(S(t2),r) returns TRUE. We will refer to a unification as partial only if it is not full. When substitution S unifies negative object 01 with positive object 02, it produces a <u>residual</u> which is empty if S(01) and S(02) are fully unified by S. Otherwise, this residual *is* an object with the same NAME as S(01) and contains also any attribute:term from S(01) that could not be unified with a corresponding attribute:term from S(02),

For example, substitution [x:J0HN, y:ATLANTA] partially unifies -l[NAME:x, AGE.-GREATER-THAN(28), FLIES:[NAME:VERB, TENSE:PAST, DEST:y]
with [NAME:JOHN, FLIES:[NAME:VERB, TENSE:PAST, DEST:ATLANTA]] and would produce as output -1(0) where 0 is the residual [NAME:JOHN, AGE:GREATER-THAN(28)].

However, the negative object would have been fully unified with [NAME:JOHN, AGE:43, FLIES:[NAME:VERB, TENSE:PAST, CARRIER:DELTA, DEST:ATLANTA, DURING:1983]] assuming that the execution of GREATER-THAN(A3,28) returned TRUE.

3. <u>Inference Rules</u>

Rules that are prefixed with the letter R are replacement rules and operate on a single input. Rules prefixed by T involve two inputs whereas rules prefixed by S involve the splitting of a disjunction F\/G. Since formula P(F) will be regarded as equivalent to F when P - 1, any rule which applies to P(F) when P - 1 will apply also to formula F.

R1: Delete P(F) if |P| is <= some threshold such as .2.

R2: Replace P1(P2(F)) by Min(|P1|,|P2|)F if both P1 and P2 have the same sign.

R3: Replace P1(P2(F)) by -1*Min(|P1|,|P2|)F if P1 and P2 have opposite signs.

R4: Replace P(F/\G) by P(F), P(G) if P > 0.

R5: Replace P(F/\G) by P(F)\/P(G) if P < 0.

R6: Replace P(F-->G) by P(F)-->G if P > 0.

R7: Replace P(F-->G) by (-1*P)F, P(G) if P < 0.

R8: Replace P(F\/G) by P(F)\/P(G) if P > 0.

R9: Replace P(F\/G) by P(F), P(G) if P < 0.

R10: Replace P1(P2(F))-->G by (Min(|P1|,|P2|)F)-->G if both P1 and P2 have the same sign.

R11: Replace P1(P2(F))-->G by (-1*Min(|P1|,|P2|)F)-->G if P1 and P2 have opposite signs.

R12: Replace P(F/\G)-->H by P(F)-->(P(G)-->H) if P > 0.

R13: Replace P(F/\G)-->H by P(F)-->H, P(G)-->H if P < 0.

R14: Replace P(F\/G)-->H by P(F)-->H, P(G)-->H if P > 0.

R15: Replace P(F\/G)-->H by P(F)-->(P(G)-->H) if P < 0.

R16: Replace P(F-->G)-->H by ((-1*P)F)-->H, P(G)-->H if P > 0.

R17: Replace P(F-->G)-->H by ((-1*P)F)-->(P(G)-->H) if P < 0.

R18: Replace P(F)-->G by ((-1*P)F)\/G if F is a procedural formula.

R19: Delete P(O) if object O is empty or NAME is its only attribute.

R20: If P > 0 and object O has NAME:N, A1:t1, A2:t2 where A1 = A2, then replace P(O) by P[NAME:N, A2:t2] and P(O1) where object O1 has every attribute:term from O except A2:t2.

T1: Suppose P1(O1) and P2(O2) are in the data base for P1*P2 > 0, where -1*P1(O1) and P2(O2) can be unified (fully unified if P2 < 0) by a substitution S without instantiating any variables in object O1. Then,

T1A: If |P1| <= |P2|, then replace P1(O1) by P1(O) where O is the residual produced by the unification.

T1B: If P1(O1) and P2(O2) do not have the same independent ancestor and 0 < MAX(P1,P2) < 1, then produce P(C) where P = P1 + P2 - P1*P2 and C is the object formed from all attribute:terms common to both S(O1) and S(O2).

T2: If PI(01) and P2(02) are <u>fully</u> unified by the substitution S, where P1 < 0 and P2 > 0, then produce the iolution triple (W,S(OI),S) where W equal! MIN(|P1|,P2). W *is* called the <u>value</u> of the triple. This iolution triple will be targeted for some node N, as described in section 4, and then will be compared with previous solution triples from node N.

We say that solution triple (A,B,.C) <u>subsumes</u> solution triple (DfE,F) if, treating C and F as property lists, we can fully unify -1(C) and F without instantiating any variables of substitution F.

Each previous solution triple (A,B,C) of node N first is examined to see if it is subsumed by the current triple (W,S(01),S). If (W,S(01),S) subsumes (A,B,C) where A < 1 and these two triples do not have the same independent ancestor, then A in the triple (A,B,C) is replaced by A + W - A*W and the revised triple (A,B,C) inherits the independent ancestors of (W,S(01),S). After all previous solutions have been examined in this manner, they then are reexamined to see if any subsume the current triple.

If the reexamination shows that (D,E,F) subsumes (W,S(01),S) where W < 1 and these two triples do not have the same independent ancestor, then W in the current triple (W,S(01),S) is replaced by D + W - D*W and the current triple inherits the independent ancestors of (D,E,F). After all previous solution triples have been reexamined in this manner, the current triple is added to the list of solutions for node N if it is not subsumed by a previous solution whose value was as great as the updated value W of the current triple (else the current triple *is* discarded).

T3: If PI(01) and P2(02) can be <u>partially</u> unified, where P1 < 0 and P2 > 0, then produce as output P(0) where P - -1*Min(|Pi|,P2) and 0 is the residual of the unification.

T4: Suppose PI(01)—>B and P2(02) are in the data base for PI*P2 > 0. Let P« MIN(|P1|,|P2|). If substitution S <u>fully</u> unifies -1*P1(01) and P2(02), then produce as output P(S(B)). If P2 > 0 and substitution S <u>partially</u> unifies -1*P1(01) and P2(02), then produce as output P(0)—>S(B) where 0 is the residual created by the unification.

The matcher will examine the attribute:terms from the negative object in order of occurrence (except that application of an attribute:term from the negative object is delayed if the term involves an executable function symbol). When given a choice, it will instantiate a variable from the positive object rather than the negative object, except when a full unification is required. If a full unification is not required, then once an attribute (other than NAME) from the negative object has one of its variables constrained, the instantiation of variables associated with subsequent attributes of the negative object will not be allowed. For example, unification of [NAME:J, AI:x, A2:y, A3:x, A4:b] with -1[NAME:J, AI:w, A2:d, A3:c, AA:w] for variables w,x,y and constants b,c,d,J would set x - w, y - d but would not instantiate either x - c or w - b since setting either x = c or w - b would necessitate instantiation of variable w from the negative object and variable w already was constrained to equal x by the previous attribute AI. Instead, it would produce the residual -1[NAME:J, A3:c, A4:W] which later could be unified with [NAME:J, AI:x, A2:y, A3:x, AA:b] for x - c and w - b.

S1: Replace formula-frame [DEFINITION:F\/G, NODE:P, ...] by first creating two nodes NI and N2 where NI initially is [FORMULAS:NIL, SOLUTIONS-.NIL, PFORM:F\/G, PNODE:P, NEXT-CASE:G, SIBLING:N2] and N2 initially if [FORMULAS:NIL, SOLUTIONS:NIL, PFORM:F\/G, PN0DE:P, NEXT-CASE:NIL, SIBLING:NI]. PN0DE:P indicates that node P is the parent of this node and PFORM:F\/G indicates thF\/G s the formula which produced this node. Formula F is created and targeted for node NI. Any solution produced by way of formula F *is* added to the SOLUTIONS attribute of node NI. As soon as a single solution is found, the formula G will be created and targeted for node N2. Any variable appearing in both F and G is suitably tagged and referred to as a <u>labeled</u> variable. A variable, whose value is sought as an answer to the original question, *is* also tagged as a labeled variable. When attempting to unify a labeled variable with an ordinary variable, it will be the ordinary variable which gets instantiated. Both rule TI and the subsumption tests of rule T2 are not allowed to instantiate a labeled variable. Each formula-frame has an attribute called SUBSTITUTION whose value is the substitution that reflects all bindings of labeled variables along the path which led to the formula. In order for a two-input inference rule to be successful, it must be able to reconcile the substitutions inherited from each of its two inputs. The reconciliation *is* accomplished by unifying the values of corresponding variables in these substitutions and then merging the substitution elements to produce a consistent substitution S. For example, if the two substitutions were [u:z, v:f(53), w:37] and [u:y, v:f(z), x:A8] for variables u,v,w,x,y and z, then the reconciled substitution S would be [u:53, v:f(53), w:37, x:48, y:53, z:53]. The reconciled substitution S then would be merged with any bindings of additional labeled variables produced by the inference rule to form the value of attribute SUBSTITUTION associated with the output of the rule.

S2: If a new solution is added to node N, and SIBLING is not an attribute of node N, then this solution represents an answer to our original question. Otherwise, the solutions on the SIBLING node of N are examined. In particular, we use the reconciliation process described at the end of rule S1 in order to make the substitution associated with this new solution consistent with the substitution associated with a solution on the SIBLING node. If the reconciliation is successful for some substitution S and the two solutions had values WI and W2 respectively, then a new solution triple (W,H,S) would be produced, where W * MIN(W1,W2) and H *is* value of attribute PFORM of both N and its sibling M. At the PNODE of N and M, the new triple would undergo the examinations described in rule T2 and it it at this PNODE that the new solution triple (W,H,S) would be assigned. It is possible that the same solution from N might be reconciled with more than one solution on the sibling node. However, if (1) the reconciled solution (W,H,S) had a value W which was >= a threshold (determined from the statement of the problem) and (2) either substitution S did not instantiate any labeled variables or its instantiation of labeled variables occurred prior to the processing of N and its sibling, then this PNODE and all its descendants would be designated as INACTIVE and no more formulas involving these nodes would be processed. A node also would be designated as inactive once both its children were designated as inactive.

A. Control Structure

The inference engine begins with a global list called UNPROCESSED, consisting of formulas that have not yet been processed by any of the inference rules. One of the unprocessed formulas on this global list could be the denial of the theorem to be proved. Whenever an inference rule creates a new formula, it *is* sent immediately to the UNPROCESSED list. While global list UNPROCESSED is non-empty, the inference engine will remove formulas from UNPROCESSED and apply the replacement rules Rl through R20. The input of a replacement rule is discarded whenever the rule is successful. Any formula which survives all the replacement rules is applied to rule T1 and then (if it survives rule TlA) *is* applied to rule T2. Upon completion of rule T2, the formula would be transferred to a global list called SURVIVORS.

When a formula is transferred to SURVIVORS, it is also transferred simultaneously to the node associated with the formula (i.e., value of attribute NODE of its formula-frame). Each of the initial formulas on the UNPROCESSED list *is* targeted for a special node called TEMPORARY. The output of a replacement rule is targeted for the same node as the formula being replaced. However, for a formula to get sent to its node, it must first reach the SURVIVORS list by completing rule T2.

The replacement rules were designed so that any formula which reaches the SURVIVORS list must be either a procedural formula, a disjunction, an elementary formula, or a production rule whose premise is an elementary formula. When the UNPROCESSED list is empty, the top ranked formula is removed from the SURVIVORS list and applied to the inference rules. If this top ranked formula *is* a procedural formula, then it is executed as a procedure (possibly sending some output formulas to the UNPROCESSED list) and then discarded. If the top ranked formula is a disjunction, then it *is* applied to splitting rule S1.

Otherwise, it is applied to the two-input rules T3 and T4, as it earlier had been applied to Tl and T2. In both instances, this formula is designated as the primary input. The attempt to apply the two-input inference rules to the primary input *is* called a cycle and proceeds without interruption. This attempt requires a search for secondary inputs since each of these rules requires two inputs. Once a primary input has completed its cycle for T3 and T4, it is used only as a secondary input. The search for a secondary input is confined to 1) the same node as the primary input, 2) nodes that are ancestors of the node associated with the primary input, and either 3) all nodes pointed to by the dictionary from key words in the primary input if the primary input is an elementary formula, or 4) all nodes pointed to by the dictionary from key words in the premise of the primary input if the primary input is a production rule. The output of a two-input inference rule (and this includes the solution triples of T2) is always targeted for the same node as the primary input.

5. Some Observations

The object-centered representation and the ability to unify arbitrarily nested property lists enables us to combine some of the expressiveness of semantic nets [6] with the deductive power of predicate logic. Since a certainty factor is computed automatically for the output(s) of each inference rule, useful information is thereby immediately made available to help control the search. Rules T1B and T2 allow us to combine independent sources of evidence, but instead of doing it via the backward chaining gathering of evidence employed by MYCIN, it is done in a more incremental manner.

The splitting rules insure that problems can be decomposed without sacrificing flexibility of control. The use of nodes as control elements helps to avoid the combinatorial explosion and backtracking that can result when a solution to a case forces a hopeless search of the subsequent case. It accomplishes this by generating solutions to the cases independently and resolving conflicts with the companion case only when a new solution is discovered. However, unlike [5], it does not attack each case in succession and then attempt to resolve conflicts afterward. It avoids a precommitment to any case by attacking the cases in parallel and resolving conflicts as they are found.

REFERENCES

1. Bledsoe, W.W., Boyer, Robert S., and Henneman, William H., "Computer proofs of limit theorems, Artificial Intelligence, vol. 2 (Spring 1972), pp. 27-60.

2. Clark, K.L. and McCabe, F.G., "PROLOG: a language for implementing expert systems," in Machine Intelligence, vol. 10, J.E. Hayes and Donald Michie (Eds.), New York:Halsted (1982), pp. 455-470.

3. Ernst, G.W. and Hookway, R.J., "Mechanical theorem-proving in the case verifier," in Machine Intelligence, vol. 10, J.E. Hayes and Donald Michie (Eds.), New York:Halsted (1982), pp. 123-144.

4. Nevins, Arthur J., "A human oriented logic for automatic theorem proving," J. Assoc. Comput. Machinery, vol. 21 (Oct. 1974), pp. 606-621.

5. Nevins, Arthur J., "A relaxation approach to splitting in an automatic theorem prover," Artificial Intelligence, vol. 6 (1975), pp. 25-39.

6. Quillian, M.R., "Semantic Memory," in Semantic Information Systems, M.L. Minsky (Ed.), Cambridge, Mass.: MIT Press, (1968), pp. 227-270.

7. Robinson, John A., "A machine-oriented logic based on the resolution principle," J. Assoc. Comput. Machinery, vol. 12 (Jan. 1965), pp. 23-41.

8. Shortliffe, Edward H. and Buchanan, Bruce G., "A model of inexact reasoning in medicine," Mathematical Biosciences, vol. 23 (1975), pp. 351-379.

9. Van Melle, William., "A domain-independent production-rule system for consultation programs," Sixth Int. Joint Conf. Artificial Intelligence (1979), pp. 923-925.