Technical Report CS73007-R

THE DYNAMIC CREATION AND MODIFICATION OF HEURISTICS IN A LEARNING PROGRAM[1]

Billy G. Claybrook
and
Richard E. Nance

May 1973

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

[1]The research reported here was undertaken while both authors were associated with the Computer Science/Operations Research Center at Southern Methodist University.

ABSTRACT

POLYFACT is a learning program that attempts to factor multivariable polynomials. The program has been successful in factoring polynomials (in simplified form) with a maximum of 84 terms, each term consisting of as many as five variables and a maximum degree of 67. The complexity of this learning task placed unusual requirements on the representation of heuristics. By using the first-order predicate calculus notation, we enable the creation and modification of heuristics dynamically during program execution. Constraints on the creation process are implemented in a series of tables by which one can alter the flexibility given to the program. Execution of heuristics begins with a translation of the predicate calculus representation to a reverse Polish string, followed by the interpretive evaluation of the Polish string. A general procedure for developing and implementing the predicate calculus representation is suggested.

C.R. Categories:

Keywords: multivariable polynomial factorization, representation of heuristics, predicate calculus, learning program

I. INTRODUCTION

This paper describes a new method for representing heuristics in a learning program. This method has been implemented in the program, POLYFACT [1], that determines the symbolic factorization of multivariable polynomials with integer coefficients. We use a slight variation of the first-order predicate calculus for both the external and internal representation of the heuristics. The notation is essentially that of the predicate calculus with only a minor difference involving domain specification for the assignment of values. This method of representation meets the need for representing complex heuristics facilitating dynamic interpretation and modification during program execution.

The effectiveness of learning schemes implemented in learning programs depends on the representation of heuristics chosen. We believe the representation described in this paper affords the programmer the capability to implement more powerful learning schemes than previous techniques.

The primary purpose of this paper is to describe the representation of heuristics in first-order predicate calculus notation. In Section II we discuss other representations of heuristics and provide a brief description of the first-order predicate calculus. We provide the motivation for the development of this representation by briefly discussing the factorization scheme implemented in POLYFACT in Section III. Section IV discusses the requirements of heuristics, describes the first-order predicate calculus representation, and the creation, modification, and execution of heuristics.

II. ELEMENTS OF THE PROBLEM-SOLVING TASK

The complex learning task confronting POLYFACT initially prompts the

consideration of the type of learning to be incorporated within the program. A second consideration involves the heuristics by which learning is implemented. With the type of heuristics determined, the representation of heuristics becomes a prominent consideration. We consider each of these, i.e. the type of learning, the heuristics by which learning is effected, and the representation of heuristics, to be elements of the problem-solving task.

Learning

Programs that learn must be capable of modifying themselves usually by altering the heuristics that guide their actions. Learning involves the need to change the mode of attack on a problem by modifying present heuristics or creating new ones. Learning also involves the selection of particular heuristics applicable to certain problem states or situations and an ordering is usually associated with these heuristics.

Learning programs should have heuristics that are: (1) easily modifiable, and (2) sufficiently powerful to represent complex actions. The learning schemes implemented in many learning programs [2], [3], [4] are not purposed toward extending the potential of the program beyond that provided by the designer. For this reason these programs do not employ a representation of heuristics that allows program modification beyond the normal adjustment of weights in a linear evaluation function.

In many cases learning has been studied in a simple environment so that more attention can be paid to the learning schemes than the problem environment. It is not clear that learning schemes developed for use in simple environments can be extended successfully to more complex environments. We

believe that a complex environment contributes to a more realistic approach to studying learning. The implementation of learning in POLYFACT is described in Section III.

Representation of Heuristics

Previous Representation Schemes

The representation of heuristics should permit their use in several problem environments since much effort goes into developing heuristics and methods for executing them. Previous programs have incorporated heuristics generally in one of three forms: (1) an integral part of the program code, (2) linear evaluation functions whose coefficients are modified through learning, or (3) a production language like that used by Waterman [5]. (Waterman provides an excellent discussion on the automatic learning of heuristics.)

Heuristics in the first form usually are so interwoven with the program code that it is difficult to identify the heuristics much less manipulate them. Heuristics in the form of linear evaluation functions are typical of heuristic search programs that attempt to determine solutions by searching graphs. Samuel's checker program [2] remains the most successful learning program using this form of learning. The third form is utilized by Waterman [5] who has provided the impetus for developing machine learning of heuristics by devising a representation of heuristics in a production language. These productions are similar to those used to describe phrase structure grammars and can be dynamically created, modified, and executed.

First-Order Predicate Calculus

The reader desiring a comprehensive description of the first-order predicate calculus should consult other sources, e.g. Korfhage [6] or Mendelson [7]. We attempt only to furnish a brief introduction to the principal characteristics, beginning with a few definitions.

The symbols, constants, and variables require some interpretation. The individual variables and the individual constants represent elements of a domain or set. The constants represent specific elements whereas the variables do not. The symbols (α) and $(\exists\alpha)$, where α can be any individual variable, are called respectively the universal and existential quantifiers, and are the formal equivalents of "all" and "some".

A term is defined as follows:

(1) Individual constants and individual variables are terms.

(2) If f_j^n is a function and $t_1, t_2, \cdots, t_n$ are terms, then $f_j^n(t_1, t_2, \cdots, t_n)$ is a term.

(3) The only terms are those formed by (1) and (2).

A propositional variable is a variable that has either a true or false value. A string is an atomic formula if it is either

(1) A propositional variable standing alone, or

(2) A string of the form $F_j^n(t_1, t_2, \cdots, t_n)$, where F_j^n is a predicate and $t_1, t_2, \cdots t_n$ are terms.

A well-formed formula (wff) is defined as follows:

(1) An atomic formula is a wff.

(2) If A is a wff and α is an individual variable, then (α)A and $(\exists\alpha)$A are wffs.

(3) If A and B are wffs, then $\sim(A)$, $(A)\supset(B)$, $(A)\wedge(B)$, $(A\vee B)$, and $(A)\equiv(B)$ are wffs.

(4) The only wffs are those obtainable by finitely many applications of (1), (2), and (3).

The expression to which the quantifier is applied is called the scope of the quantifier. The occurrence of an individual variable x is bound if it is an occurrence of (x), $(\exists x)$, or within the scope of a quantifier (x) or $(\exists x)$. Any other occurrence of a variable is a free occurrence. An occurrence of an individual variable is bound by the innermost quantifier on that variable within whose scope the particular occurrence lies.

III. OVERVIEW OF POLYFACT

Our discussion of POLYFACT is limited to the aspects of this learning program that relate to the requirements for heuristics and the consequent motivation for using the predicate calculus as a representation form. A complete description of POLYFACT is given in the dissertation of one of the authors (Claybrook [1]). The primary objectives in the development of POLYFACT were:

(1) to design a multivariable polynomial factoring program that could be used as a vehicle in a complex learning environment,

(2) to develop a powerful representation for heuristics permitting dynamic creation and modification,

(3) to show that learning through the dynamic modification of heuristics can be used successfully in a complex environment to increase the efficiency of the program, and

(4) to demonstrate the use of a classification scheme enabling the program to extend itself to newly classified polynomials and furnishing a mechanism for implementing localized learning.

We are primarily interested in discussing (2) above, but we emphasize that

the other objectives were responsible for causing the development of the representation discussed in the remainder of this paper.

The factorization scheme [8] implemented in POLYFACT relies on the fact that a reducible polynomial can be written as the product of two factors, one with M-terms and one with N-terms. During a factorization attempt the M-term factor is sought, and the N-term factor is determined by division of the M-term factor into the subject polynomial. Then both factors are saved and later reduced.

POLYFACT attempts to minimize the amount of searching for the M-term factor by: (1) building a model for each polynomial, (2) using learning for term selection to initiate the factorization process, and (3) using learning to select term possibilities in the M-term factor. The model-builder is not described in this paper since it has no direct bearing on the representation of heuristics, but its importance to the factorization scheme in POLYFACT is considerable.

Classification of Polynomials

POLYFACT classifies polynomials according to certain features that each exhibits. Through classification the capability exists for applying specific heuristics to a designated polynomial. Two types of features are used in classification: surface features and hidden features. Surface features are those features that can be determined by visible examination of the subject polynomial. Hidden features are those features not immediately visible to either a human or a pattern recognition program. The hidden features are detected during a factorization attempt, i.e. during the factorization of a polynomial characteristics are discovered that are not obvious from the initial examination.

The detection of hidden features during a factorization attempt usually results in a reclassification of the polynomial unless the current factorization attempt is successful. The reclassification process is a powerful one since it provides the capability to apply different sets of heuristics to a single polynomial during its factorization.

POLYFACT also uses the classification mechanism as a means for extending itself to factor newly classified polynomials. It does this by borrowing heuristics from a previously classified polynomial with similar features. Each classified polynomial has its own independent set of heuristics. Thus, learning can be associated independently with each class of polynomials. One problem generally associated with localized learning via a classification scheme is the amount of memory required to store all the individual sets of heuristics. We have solved this problem by developing a representation of heuristics that can be easily encoded into a form requiring little memory and then decoded for expansion into the predicate calculus notation prior to execution. This idea of encoding and subsequent decoding requires the heuristics to be composed of distinguishable components that can be easily manipulated.

Learning in POLYFACT

We have stated that the amount of searching for the M-term factor is reduced by using learning to aid in the selection of a term to initiate the factorization process and to select term possibilities for the M-term factor. The primary objective in term selection is to choose a term that leads to a small search space. The heuristic associated with directing learning in term selection utilizes the presumption that the term exhibiting the fewest number of possibilities leads to the minimum search space.

The learning associated with term selection is as follows. After a factorization attempt is complete, the number of possibilities in each term of the polynomial is determined. The features of the term(s) with minimum number of possibilities have their frequency count(s) increased. Then heuristics are constructed dynamically (and ordered) to reflect the importance of the features in selecting a term T to initiate the factorization process, i.e. if a feature has the highest frequency count, then all terms that do not have this feature are removed from consideration for T. The order of the heuristics for term selection can vary during program exeuction since POLYFACT adapts to the sequence of input polynomials.

The possibilities that can be selected as terms in the M-term factor are ranked according to their apparent merit in determining the correct M-term factor, and during a factorization attempt the highest ranked possibilities are selected. The features for possibility selection determine the rank of each possibility. After a polynomial has been factored, each of the terms in the M-term factor is examined to determine its set of characteristic features. A binary vector is created with nonzero entries indicating the features present. Then a heuristic is constructed in first-order predicate calculus notation using as predicates those features characterizing the M-term factor.

To facilitate the construction of this set of heuristics, a matrix is maintained providing a history of the features of terms that have appeared in factors of previous polynomials. After the vector of features has been created for a term, it is compared with each row in the matrix to determine if the vector is already present. If so, the frequency count for the matching row is incremented. If the vector is not in the matrix, it is added and

the corresponding heuristic is created.

When these heuristics are used to rank the possibilities, a binary vector is created with nonzero entries appearing in the vector positions corresponding to the predicates in the satisfied antecedent. When an antecedent is satisfied the vector (created by the satisfied antecedent) is compared to each row in the history matrix. If the vector is in the matrix, the rank of the possibility is the frequency count, kept as an augmented column entry in the matched row; otherwise, the rank is zero.

The possibility selection heuristics are maintained in complete predicate calculus notation and also in the encoded matrix form. The matrix form is convenient for determining the need for modifications to the heuristics. The term selection heuristics are kept in an encoded form for all classes of polynomials and then expanded prior to execution.

Analysis of a Factorization Attempt

Regardless of failure or success, the results of each factorization attempt are analyzed. During this analysis, POLYFACT determines if the heuristics for term and possibility selection require modification and whether or not the polynomial warrants reclassification in the case of failure. The learning associated with term and possibility selection can be ended after an appropriate training period if the user desires. Then heuristics are not modified after a factorization attempt until the learning indicator is reset.

As one can determine from the above discussion, the demands on the heuristics in POLYFACT necessitate a representation that can be easily manipulated and modified during program execution. The predicate calculus

notation satisfies these requirements and we believe that this representation allows the implementation of learning schemes more complex than described in the literature.

Nature of the Heuristics in POLYFACT

The creation and modification of all the heuristics in predicate calculus notation are directed by the learning programs in POLYFACT. The set of heuristics determined by the classification scheme for a particular polynomial guides the actual factorization attempt. Each set of heuristics consists of several subsets, with each subset having a specific function to perform. These subsets are responsible for:

(1) selecting a term T to initiate a factorization attempt,

(2) creating the set S of all possibilities in the term T,

(3) ranking the possibilities in the set S according to their probable merit in creating the M-term factor,

(4) selecting a possibility P from S, where P is the first term in the M-term factor,

(5) creating the set S' of possibilities used to complete the M-term factor,

(6) ranking the possibilities in S', and

(7) creating the remainder of the M-term factor (terms 2 through M) by the selection of terms from S'.

We make no attempt here to describe the details associated with creating the sets of possibilities referred to above. Instead, we refer the reader to [1] or [8].

IV. HEURISTIC REPRESENTATION USING THE PREDICATE CALCULUS

Requirements of Heuristics

The representation of the heuristics is probably the key to the success of any heuristic program (with or without learning). Representation of heuristics in a complex problem-solving environment prompts several considerations:

(1) The heuristics must be capable of representing complex actions.

(2) The creation, modification, and execution of heuristics should be relatively simple tasks.

(3) An appealing property of a representation scheme is that it conserve storage.

(4) The representation should allow at least a partial solution of the credit-assignment problem [5].

(5) The heuristics should be modular, i.e. the representation should allow the construction of heuristics from distinguishable components.

(6) The representation should allow heuristics to be referenced as individuals or as members of designated sets of heuristics.

(7) The heuristics should permit dynamic manipulation during program execution.

(8) The final consideration is the flexibility of the representation, i.e. the ability to interchange the components that comprise the heuristics in the event that heuristics are changed by the designer.

Some of the above considerations for the representation of heuristics require discussion and/or justification. The first consideration is motivated by the realization that many of the actions performed in learning programs such as POLYFACT require a comprehensive analysis of the problem situation, i.e. several criteria must be considered, often simultaneously, to assure that all prior conditions are satisfied before performing an action.

The second consideration does not necessarily imply the decisions related to the actual creation and modification of heuristics to be simple; however, once these decisions are reached for particular heuristics, the procedures

should be mechanical in nature and relatively simple to execute.

The third and fifth considerations are complementary. If the heuristics are modular, they can be represented in an encoded form to conserve storage. This is especially important in heuristic programs that use classification mechanisms for implementing localized learning.

The fourth consideration, solution of the credit assignment problem, is included because of its importance to learning mechanisms. Representation of heuristics in a learning program must enable the assigning of credit for success among the many heuristics of potential use in solving a particular problem.

In (6) above we are suggesting that each heuristic should be an entity with an individual identifier. The practice of associating an identifier with each heuristic is useful when learning is implemented in different parts of the program. The heuristics peculiar to each area can be referenced and executed as required by a standard execution program.

Of the eight considerations stated above, the final four, i.e. considerations numbered 5 - 8, constitute significant requirements beyond those usually placed on the representation of heuristics. These four requirements are essential for the dynamic creation and modification of heuristics during program execution. In attempting to satisfy these requirements, we have developed the predicate calculus representation described below.

First-Order Predicate Calculus Representation

In treating the creation, modification, and execution of heuristics, we provide a general description of the first-order predicate calculus representation. While we believe the representation to offer distinct general

advantages, we recognize that other applications might foster details that we do not consider. Consequently, we limit the description of implementation details to only the few taken from POLYFACT to illustrate certain ideas.

General Form

The notation is identical to that of first-order predicate calculus except for a minor difference involving domain specification for the assignment of values. In the implementation of the predicate calculus notation, a heuristic can have one of two general forms:

(1) NAME ($DOMAIN_1$) ($DOMAIN_2$)$\cdots$($DOMAIN_k$) (($ANTECEDENT_1$ C $CONSEQUENT_1$) 0$\cdots$0 ($ANTECEDENT_n$ C $CONSEQUENT_n$)) \$, or

(2) NAME (($ANTECEDENT_1$ C $CONSEQUENT_1$) 0 ($ANTECEDENT_2$ C $CONSEQUENT_2$) 0$\cdots$0 ($ANTECEDENT_n$ C $CONSEQUENT_n$))\$.

In either of the above forms, the same antecedent or consequent can occur several times; but the same antecedent-consequent pair should occur but once. The first form has a non-null domain; whereas, the second has a null domain. One of the functions of the non-null domain is to specify an ordered set from which the values for the variable (indicated in the domain field) are taken. Some of the variables in the antecedent-consequent pairs can be free, i.e. their values are specified elsewhere. Each bound variable must appear as an argument in at least one antecedent or consequent, i.e. each variable specified in a domain must appear as an argument in at least one of a predicate, a function, or a consequent. The domain as defined in this paper corresponds to the quantifiers in predicate calculus notation; however, in predicate calculus notation the domain is not included as a part of the quantifier. The order of domain precedence is identical to that of the quantifiers.

Each antecedent is a single predicate or a logical combination of predicates

connected by conjunction ('A' = AND) and/or disjunction ('O' = OR) operators. Each predicate is a logical function and can be referenced with arguments that are constants, variables, or functions. 'C' is the conditional operator, and the consequent is always the name of a routine (or procedure) that is executed when the corresponding antecedent is satisfied.

To illustrate the representation of heuristics in the predicate calculus notation, we use an example taken from the term selection heuristics in POLYFACT:

H1.1 (E T IN IPTRSO) ((N H1(G11(T), MINDEG) C FIX123)) $

This heuristic consists of the components:

H1.1	is the NAME of the heuristic,
(E T IN IPTRSO)	is the DOMAIN of the heuristic,
N	is the negation operator,
H1	is a predicate that is 'TRUE' if G11(T) equals MINDEG,
G11	is a function whose value is the degree of term T,
T	is a bound variable,
MINDEG	is a constant function, i.e. a function whose value is constant during the execution of the heuristic,
C	is the conditional operator, and
FIX123	is a CONSEQUENT.

Internal Structure of Heuristics in POLYFACT

The heuristic with name H1.1 resets the use flag (the use flag is used to indicate membership in a set) of each term T with degree exceeding MINDEG in the set IPTRSO. The value of MINDEG is the degree of the minimum degreed term in IPTRSO. IPTRSO is the set of terms in the input polynomial to POLYFACT.

Each individual heuristic in POLYFACT is represented internally as a right-linked list. The value of each cell in the list is an alphanumeric string of characters representing a token in the heuristic. The tokens in the heuristic are individual entities such as 'H1.1', 'C', 'E', 'T', 'IN', 'IPTRSO', etc. The internal representation of heuristic H1.1 is shown in Fig. 1.

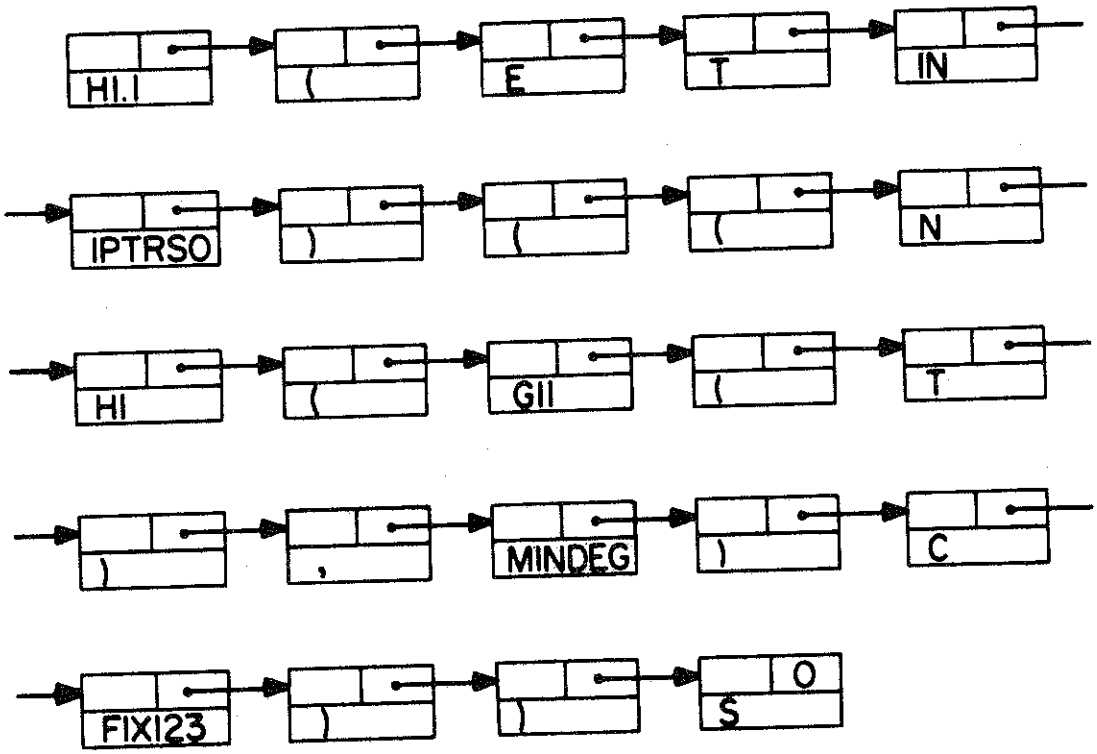
The heuristics corresponding to a particular classification in POLYFACT are similar in representation to H1.1. All heuristics in a set are linked together as a left-linked list with the subsets comprising the total set delimited by a cell with value '?'. Generally an entire subset of heuristics is executed; however, by using the name of a heuristic and a pointer to the total set of heuristics, an individual heuristic can be referenced, i.e. any single heuristic can be executed provided its name is known.

"Insert Fig. 1 here"

Creation and Modification

We have emphasized that the learning process in POLYFACT is through the dynamic creation and modification of heuristics. Tables are used to define constraints on the creation of heuristics. Although these constraints could be null, giving the learning program total responsibility for the creation decisions, some limits seem appropriate for most applications. In both the creation and modification of heuristics, tables serve to specify relationships between predicates, consequents, domains, etc. The data in these tables can be read from input cards, thereby adding a note of flexibility to their creation.

Fig. 1. Internal representation of heuristic H1.1


HI.I
(
E
T
IN
IPTRSO
)
(
(
N
HI
(
GII
(
T
)
,
MINDEG
)
C
FIX123
)
)
O
S

The consequent-predicate table gives the correspondence between each consequent and the predicates that can be used to form an antecedent-consequent pair. The consequent-domain type table specifies the correspondence between each consequent and the sets from which values for a variable are selected. Each bound variable must be an argument in a predicate (within an antecedent) or consequent. Normally, we can expect the variable in a domain field to be used as an argument in specific antecedent-consequent pairs designated in this table.

The domain type-variable-set table defines the variable-set pair associated with a domain type. The domain is determined by the variable and the set from which the values of the variable are taken. The purpose of this table is to prevent heuristics with a given type of domain from using predicates and consequents associated with another type of domain. In addition this table might prevent the creation of heuristics which have a certain mixture of domains.

These three tables are used by the routines that create and modify heuristics. The implementation of these tables for the creation of heuristics can be simplified by having entries that are pointers to lists containing the variables, predicates, consequents, etc. in symbolic form rather than storing the symbolic names of these tokens in the tables.

Associated with the translation from infix predicate calculus notation to Polish postfix notation, which is described in subsequent paragraphs, is a predefined symbol table. This symbol table has an entry for each token that can appear in a heuristic. Three attributes are associated with each token: (1) token type, (2) number of arguments, if any, associated with the token, and (3) an index. The token type identifies the token during the

translation process, i.e. it indicates whether a token is an operator or a variable. The function of each of these other two attributes is discussed later. An example of the predefined symbol table is given in Table I.

"Insert Table I Here"

In Table I, the token N (negation symbol) has type 1, a single argument and an index value of 1. G11 is a function and has type 2 (all functions are of type 2). All predicates are of type 3, and all formal parameters such as T,F,IPTRSO, etc. are of type 17.

In addition to the tables several parameters provide information for the creation routines:

ISBTYP = 1 : add a predicate to a heuristic

ISBTYP = 2 : add an antecedent-consequent pair to a heuristic

NEWSET = 1 : a new set of heuristics

NEWSET = 0 : an old set of heuristics

INUM = 0 : the heuristic is a new one

INUM $\neq$ 0 : gives the number of the antecedent-consequent pair to which a new predicate is to be added, or it gives the position in a heuristic for the addition of a new antecedent-consequent pair

IVDEX : pointer to a variable

ISDEX : pointer to a set in the domain field

ICDEX : pointer to the consequent

IANTC : pointer to the antecedent

NAME : name of the heuristic to be modified

TABLE I

EXAMPLE OF SYMBOL TABLE ORGANIZATION

TOKEN	TYPE	NUMBER OF ARGUMENTS	INDEX
N	1	1	1
A	4	2	2
O	5	2	3
C	6	2	4
(	7	0	0
)	8	0	0
$	9	0	0
G11	2	1	5

Technical Report CS73007-R

THE DYNAMIC CREATION AND MODIFICATION OF HEURISTICS IN A LEARNING PROGRAM[1]

Billy G. Claybrook
and
Richard E. Nance

May 1973

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

[1]The research reported here was undertaken while both authors were associated with the Computer Science/Operations Research Center at Southern Methodist University.

IPHRT : pointer to the heuristic with name NAME

IHEURT : pointer to the set of heuristics created.

Whenever a consequent or domain is required in the construction of a heuristic, the particular entity is formed (as in the case for domains) or simply copied (as is the case for consequents) from the lists of tokens, thereby never allowing any of the original tokens to be destroyed. All of the parameters described above are used by the creation routines. Note that IANTC is a pointer to the antecedent and not a pointer to a predicate. The antecedent is determined before the actual creation of a heuristic is performed since it can be a logical combination of predicates and is usually determined by the learning programs.

Each heuristic is represented in fully parenthesized form so that no ambiguities can result as modifications are made to the heuristics. A heuristic can have the following form:

NAME (DOMAIN) ((ANTECEDENT$_1$ C CONSEQUENT$_1$) 0

(ANTECEDENT$_2$ C CONSEQUENT$_2$) 0 (ANTECEDENT$_3$ C CONSEQUENT$_3$)) $.

Initially this heuristic consists of one antecedent-consequent pair. Modification for this heuristic results in the addition of antecedent-consequent pairs. During the execution of this heuristic, execution is accomplished for the first consequent whose antecedent is satisfied. This execution process is similar to the COND statement in the LISP language [9].

The order of the antecedent-consequent pairs in a heuristic can be important, especially when more than one antecedent can be satisfied per execution. With multiple antecedents the parameter INUM becomes crucial, and some procedure is required to determine its value and to establish the reordering when antecedent-consequent pairs are added to a heuristic. To

increase the flexibility, the creation and modification programs can insert a heuristic between two already existing heuristics in a set, thereby changing the order of execution. When the insertion process is necessary in POLYFACT, the name of the predecessor for the heuristic must be specified. Insertion is accomplished easily in POLYFACT since only left-linked pointers must be altered.

Within POLYFACT the heuristics are created and modified dynamically during program execution. However, if the predicate calculus representation is used in a heuristic program not requiring this capability, then the heuristics can be created by reading the parameter values in the above list from input cards. An alternative procedure would be to input the heuristics in the first-order predicate calculus representation.

We remind the reader that any symbol in the predefined symbol table can be used as an entity in a heuristic. For example, the formal arguments in a predicate can be different from one use to the next. In most instances the user requires some method of controlling which entities are allowed to be selected during creation, and he can control this selection by the use of the tables described earlier. The modular form of the representation allows the designer to increase the entities available for heuristic construction by adding them to the input stream.

Execution of Heuristics

Each time a heuristic is executed, it is translated into a reverse Polish string and then executed interpretively. Interpretive execution of the heuristics in POLYFACT is necessary since they are modified dynamically. However, the execution routine is quite straightforward.

Whenever a heuristic or subset of heuristics is to be executed, a pointer to this set is passed to the execution routine. The heuristics that contain non-null domains select elements from the sets given in the domains. In the selection of elements from a set, a heuristic can consider all elements in the set, and the elements can be selected serially or randomly. In this case, the domains have the form:

(E T IN IPTRSO),

where the E indicates that all elements (T) in the set IPTRSO are selected during the execution of the particular heuristic. A heuristic with a non-null domain can also consider elements from a set until an antecedent is satisfied. The corresponding consequent is then executed and activation of this heuristic is terminated. This type of domain is represented as:

(EA T IN IPTRSO).

We refer to the execution routine as procedure EVAL. EVAL is responsible for determining when the execution of a subset of heuristics is to cease. It is also responsible for identifying the sets within each domain field and monitoring the order in which elements are selected from these sets. For example, assume that the heuristic currently executing is:

H1.3 (EA F IN IFPTR1) (EA T IN IPTRSO) ((H1(L11(T,F),TF) C SUB1))$

The execution of heuristic H1.3 ceases the first time an antecedent is satisfied, i.e. the consequent SUB1 is executed at most once. The execution of the heuristic formed by replacing EA by E in the innermost domain in heuristic H1.3 can cause multiple executions of the consequent SUB1. Note that the nested domains give the same effect as nested loops, i.e. elements are selected from the inner domain more rapidly than the outer domain.

The precedence functions in Fig. 2 are used in the translation algorithm.

x	F(x)	G(x)
FUNCTION (OR SUBROUTINE)	15	15
PREDICATE	14	14
N	12	13
A	11	11
O	10	10
C	9	9
(	8	7
)	7	7
$	6	6

Fig. 2. Precedence functions for parsing algorithm

The translation algorithm in Fig. 3 is similar to that of Graham [10]. The σ and γ symbols in Fig. 3 are the current token and top of stack token, respectively.

"Insert Fig. 2 here"

"Insert Fig. 3 here"

The translation process is carried out in the following manner:

(1) The translation algorithm processes each token in the heuristic in a left to right manner. A pointer to the corresponding token in the predefined symbol table is determined.

(2) The algorithm uses the precedence functions in Fig. 2 to determine whether to output the token (actually the symbol table pointer replaces the token in the reverse string) to the reverse string or place it on a stack.

(3) When all tokens have been scanned and the stack is empty, the Polish string has been formed and each token is represented by its symbol table entry pointer.

EVAL changes the contents of each cell in the string to obtain the token type, number of arguments, and the index. Once each cell in the string contains this information, EVAL is ready to begin execution of the heuristic. Even if a heuristic is executed several times, e.g. a heuristic with domain (E T IN IPTRSO), the translation process is performed only once.

We use the following heuristic to illustrate the translation process:

H1.2 (E T IN IPTRSO) ((N H1(G11(T),MINDEG) A N H1(G21(T),MINVAR)C FIX123)) $.

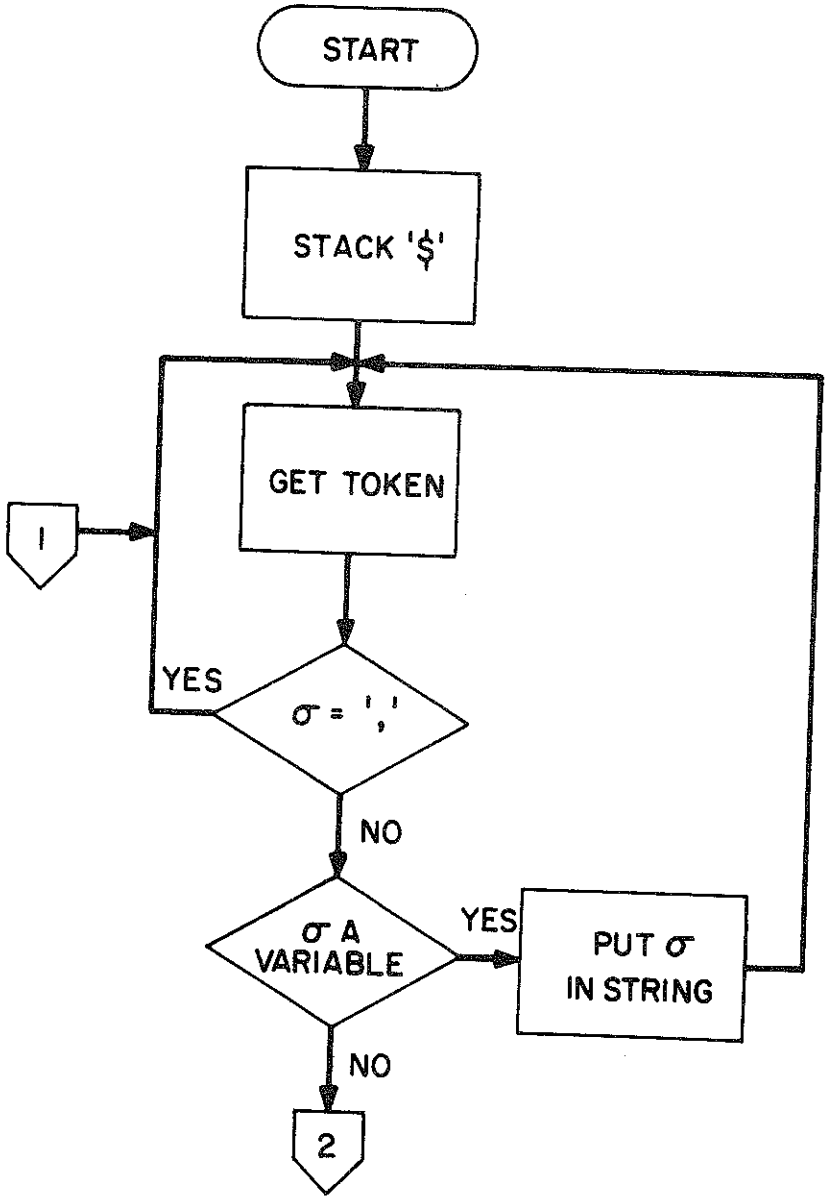

START
STACK '$'
GET TOKEN
1
YES
σ = ','
NO
σ A VARIABLE
YES
PUT σ IN STRING
NO
2

Fig. 3. Translation algorithm for first-order predicate calculus

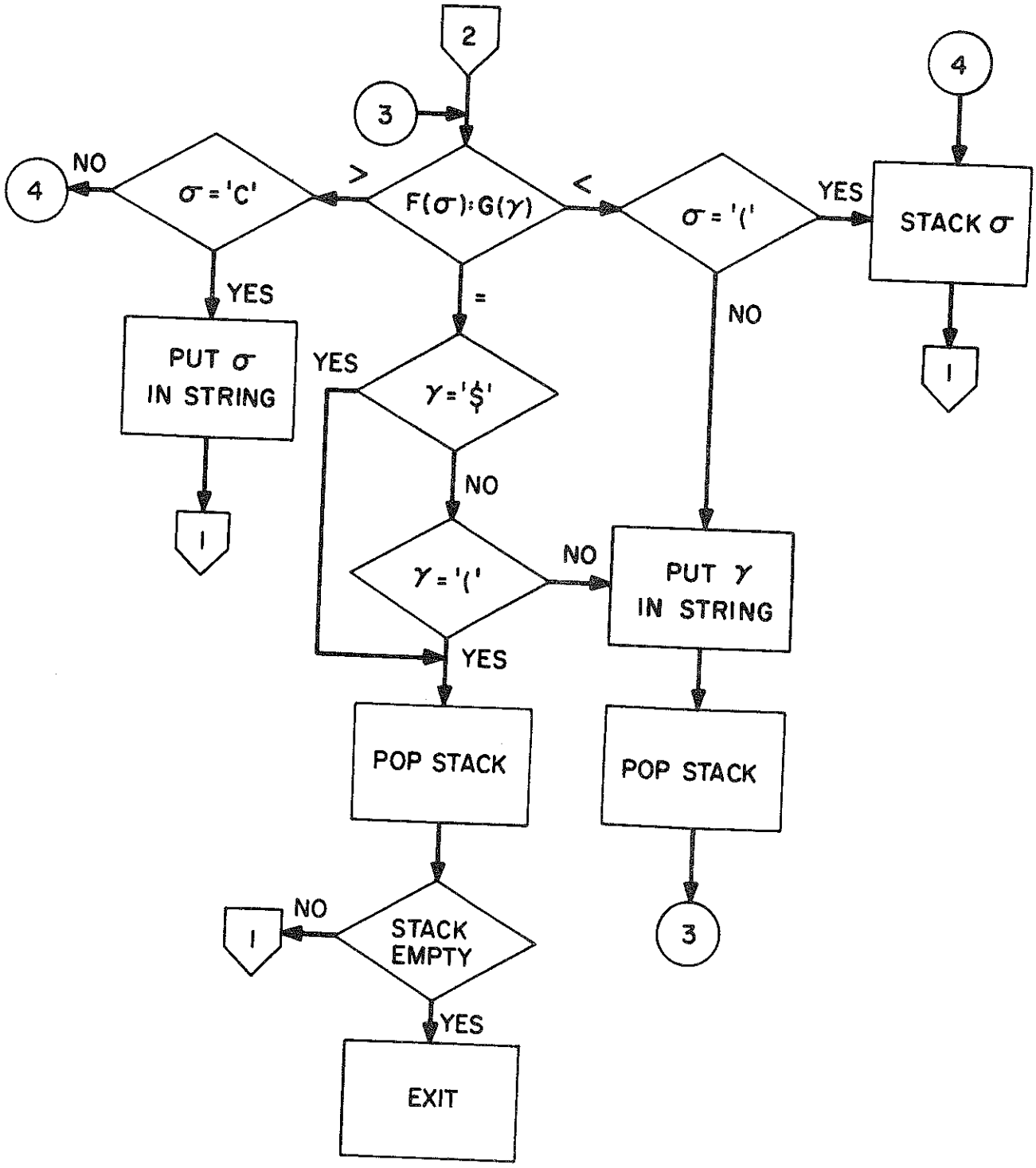

2
3
4
4
NO
σ = 'C'
>
F(σ):G(γ)
<
σ = '('
YES
STACK σ
YES
PUT σ
IN STRING
=
NO
1
YES
γ = '$'
NO
1
γ = '('
NO
PUT γ
IN STRING
YES
POP STACK
POP STACK
NO
1
STACK
EMPTY
3
YES
EXIT

Fig. 3. Continued

This heuristic is translated into the following ordered set:

{T,G11,MINDEG, H1,N,T,G21,MINVAR,H1,N,A,C,FIX123} .

One should note that the translation from infix to postfix notation is not strict; this is illustrated by the relative positions of C and FIX123. A strict translation to postfix notation would result in FIX123 C. Since we want to execute the consequent FIX123 only if the antecedent is true when the token C is encountered (indicating that the antecedent value has been determined), we check the value of the antecedent. An antecedent value of "TRUE" triggers execution of the consequent, otherwise execution is omitted. This small variant in postfix notation simplifies the execution process since EVAL always executes a function or subroutine when its name is mentioned in the reverse Polish string.

The final executable form of the Polish string above is shown in Fig. 4. If IP is a pointer to any cell in the list, then LLINK(IP) is the token type, LLINK(IP+1) is the number of arguments and RLINK(IP+1) is the index. Since POLYFACT is implemented in FORTRAN, the index is for a computed GOTO; however, generally speaking the index is an indicator where control in EVAL is to be transferred for some specified action to take place.

"Insert Fig. 4 here"

EVAL (in POLYFACT) has a statement number which corresponds to each index in the predefined symbol table. When the Polish string is executed, the index entry in each cell indicates the statement number to which control is transferred causing some action. The action may be: (1) stack an argument, (2) reference a predicate and stack the resulting logical value, (3) call

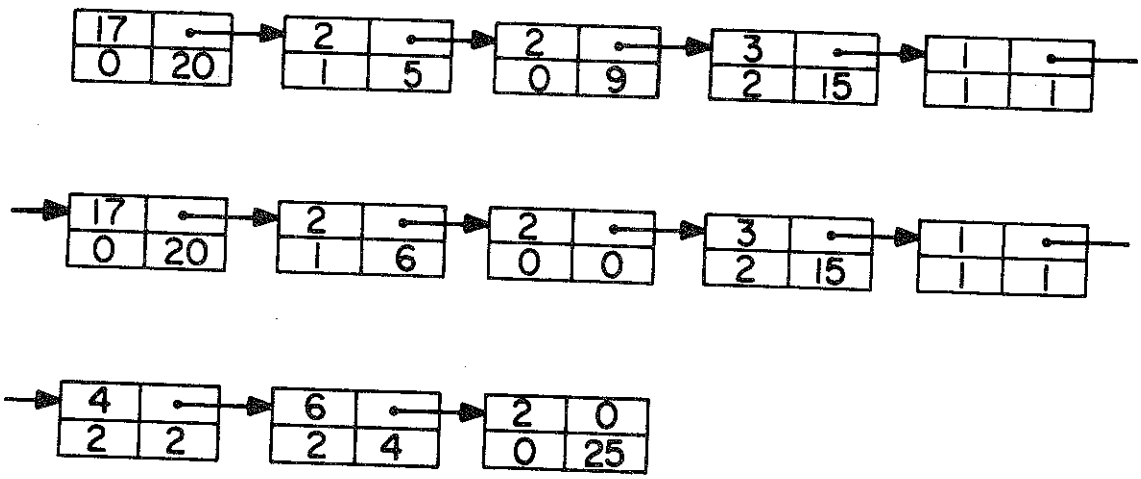

17
0
20
2
1
5
2
0
9
3
2
15
1
1
1
17
0
20
2
1
6
2
0
0
3
2
15
1
1
1
4
2
2
6
2
4
2
0
0
25

Fig. 4. Executable form of Polish string

a consequent type subroutine, or (4) perform a logical operation and stack the result. If a function, predicate, or subroutine has arguments, then the entry LLINK(IP+1) in the corresponding cell is nonzero. In this case before the routine is executed, the arguments are popped from the argument stack and placed in their respective positions in the parameter list of the routine. This process allows the arguments in a predicate or the functions in a heuristic to vary from one execution to the next.

V. SUMMARY

The task for which POLYFACT is designed is complex. Factorization of multivariable polynomials is an exceedingly difficult problem for humans. An integral part of the internal strategy of POLYFACT is the ability to create and to modify heuristics dynamically, i.e. during program execution. This ability places extraordinary requirements on the scheme for representing heuristics; yet the first-order predicate calculus notation has proved effective.

We believe the predicate calculus representation to be sufficiently powerful and flexible to function effectively in problem-solving environments other than the factorization of polynomials. For someone considering the use of the representation, we can sketch a step-wise description of our development (assuming the potential user has formulated his problem):

(1) Determine the heuristics for attacking the problem.

(2) Describe the heuristics formally, if possible, in first-order predicate calculus notation.

(3) From (2) determine the domains, variables, constants, functions, predicates, and consequents.

(4) Write the necessary programs for the functions, predicates, and consequents described in (3).

(5) Write an execution program for translating and executing the heuristics.

(6) Write the routines for creating and modifying the heuristics.

(7) Create the cross-reference tables or give the program the capability for learning the correspondences.

Specific aspects of the relationships between heuristic representation and other elements of the problem-solving task in POLYFACT might prove useful to others. In particular, the cross-reference tables to define constraints on the creation of heuristics seems an effective approach. One can apply these tables so as to effect quite sophisticated learning mechanisms. Also, one might want to consider a different interpretation for the termination of the execution of heuristics than the one we proposed in Section IV. For example, we could interpret the execution of a heuristic with domains (EA F IN IFPTR1) (E T IN IPTRS0) to cease the first time an antecedent is satisfied. The implementation of this interpretation is more difficult and less natural than the one we propose. Finally, we believe the predicate calculus representation enables a more comprehensive implementation of learning than is described in previous research.

REFERENCES

1. Claybrook, B.G. POLYFACT: A learning program that factors multivariable polynomials. Ph.D. Dissertation. Computer Science/Operations Research Center. Southern Methodist University, 1972.

2. Samuel, A.L. Some studies in machine learning using the game of checkers. In Feigenbam, E., and Feldman, J. (Eds.), Computers and Thought. McGraw-Hill, New York, 1963, pp. 71-105.

3. Slagle, J.R., and Farrell, C.D. Experiments in automatic learning for a multipurpose heuristic program. ACM Communications 14 (1971), 91-99.

4. Doran, James. An approach to automatic problem-solving. In Collins, N.L., and Michie, Donald (Eds.), Machine Intelligence 1. American Elsevier, 1967, 105-123.

5. Waterman, D.A. Generalization learning techniques for automating the learning of heuristics. Artificial Intelligence 1 (1970), 121-170.

6. Korhage, Robert R. Logic and Algorithms. John Wiley & Sons, New York, 1966.

7. Mendelson, Elliott. Introduction to Mathematical Logic. Van Nostrand Reinhold Company, New York, 1964.

8. Claybrook, B.G. A heuristic factorization scheme for multivariable polynomials. VPI & SU Technical Report CS-73002, 1973.

9. McCarthy, John, Abrahams, Paul W., Edwards, D.J., Hart, T.P., and Levin, M.I. LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1969.

10. Graham, Robert M. Bounded context translation. In Rosen, Saul(Ed.), Programming Systems and Languages. McGraw-Hill, New York, 1967.