# CHALLENGE TO ARTIFICIAL INTELLIGENCE: PROGRAMMING PROBLEMS TO BE SOLVED

J. E. Sammet

IBM Corporation
Cambridge, Mass.
U. S. A.

Abstract

This paper is in the nature of a challenge to artificial intelligence experts. It suggests that the techniques of artificial intelligence should be applied to some realistic problems which exist in the programming and data processing fields. After a brief review of the little related existing work which has been done, the characteristics of programming problems which make them suitable for the application of artificial intelligence techniques are given. Specific illustrations of problems are provided under the broad categories of data structure and organization, program structure and organization, improvements and corrections of programs, and language.

Descriptors

artificial intelligence
applications
programming
heuristic techniques

## I. INTRODUCTION

It has been over 15 years since computers were first used for anything resembling "artificial intelligence". The pioneering work of Newell, Shaw, and Simon on proving theorems in the propositional calculus is so well known as not to need discussion for the people knowledgeable in the field of artificial intelligence. Similarly, the early work of Samuel in checker playing is also well known. The total amount of work which has been done in the field of artificial intelligence, as represented by bibliographies, papers, conferences, etc., is quite large, even if a narrow definition of artificial intelligence is used. However, there is a major anachronism and irony in all of this, which is the subject of this paper.

The techniques of artificial intelligence have seldom been used to improve the use of computers, i.e., the programming process, even though primitive attempts were made as early as 1958. Even worse, there are vast numbers of problems, even from the limited view of systems programming, which could benefit from the application of artificial intelligence techniques. Put another way, it seems about time that the workers in this field begin to choose as vehicles for the exploration of artificial intelligence techniques some problems whose solution might really be of use - at least to other computer scientists and to the data processing field. Playing checkers or chess on a computer is an interesting tour de force, and when such programs win from their developers then the event is even more significant. Some of the early motivations (e. g., well structured problem, attention getting) which led to the choice of these and other problems are still existent but can now also be applied to some realistic programming problems.

This should not be interpreted as saying that no useful problems have been attacked in the name of, or in the spirit of, using artificial intelligence tools. On the contrary, the assembly line balancing program of Tonge (22), the formal integration systems of Slagle (18) and Moses (16), the chemistry work in Heuristic DENDRAL (3), project scheduling (10), and even the solution of algebra word problems in STUDENT (2) all represent direct or slightly indirect realistic practical problems which have been or are being addressed. However, the thrust of this paper is to concentrate on problems which arise in, or from the programming and data processing fields, and the related issue of communication between the human and the computer.

It is also important to recognize that there really are areas in which programming technology and problems have grown so complex that straightforward techniques and algorithms are inadequate to deal with them. As one illustration, the difficulty in planning a combined software-hardware configuration for a new installation, or even for a specific application, has almost gotten out of hand. The proliferation of central processor numbers and speeds, memory sizes, peripheral and storage devices, combined with the variability inherent from operating systems which allow multi-processors, multiprogramming, on-line systems and real time systems all simultaneously, make it difficult to determine adequate (let alone the best) configurations and methods of scheduling. Even the measurement of throughput and other criteria for performance is extremely difficult. While this whole problem area is probably not yet amenable to aid from the artificial intelligence field, it is at least a specific indication of the complexity of the programming field today and the type of problem which can eventually

benefit from heuristic techniques.

## II. RELATED EXISTING WORK

The earliest work dealing with a realistic programming problem seems to be that of Friedberg (7, 8) in 1958. He assumed the existence of a computer, and tried to develop a "learning machine" which would program the computer to yield an acceptable output for each input, e. g., complement the input bit. The approach was by trial and error, with feedback given on the correctness of the result in each case. Other attempts to develop "learning programs" were made by Campaigne (4) and Arnold (1). The latter dealt specifically with finding a program for a new computer which is equivalent (with respect to input and output) to a program on another computer; he used a "modified British Museum" approach. Kilburn, Grimsdale and Sumner (11) developed a program which could produce programs for a computer with 4 arithmetic and 2 copy instructions; the program was considered acceptable if it produced a sequence of numbers satisfying some predefined criteria (e. g., weak convergence).

In 1963, Simon (17) used an approach similar to the techniques of GPS to develop a heuristic compiler which constructed an IPL-V program from stated input and output requirements. While some success was achieved in simple cases, this work has apparently not been developed further.

The DEDUCOM system (19) was primarily concerned with question answering, but did consider the problem of writing simple programs, and solved a small portion of a problem dealt with by Simon's heuristic compiler.

The problem of writing programs from stated inputs and outputs has been considered more recently. Green (9) and Waldinger (23, 24) deal with this problem, and provide simple but practical illustrations of their techniques, which are closely related to theorem proving and formal logic. In a further development, (14) describes a theorem-proving approach to automatic program synthesis.

The problem of writing programs from stated inputs and outputs is of course closely related to the concept of proving the validity of existing programs; much work has been done in this area lately, as shown by London's bibliography (12).

A conceptually different approach is used by Fikes in his REF-ARF system (6). His concern is with the development of an input language which can be used for stating a wide variety of problems to be solved, and then with the effectiveness of methods for solving those problems. Although most of the problems he solves are unrelated to programming, one example is shown of changing the values of computer registers given their existing contents and certain machine instructions.

The PILOT system of Teitelman (20) makes automatic corrections to certain errors in LISP programs, while still allowing the user to override the automatic facility.

Although the use of a computer to do formal integration is somewhat different from the types of problems with which this paper deals, the development of SAINT by Slagle (18) deserves mention. What makes this particularly interesting, and relevant for the future, is the later development of SIN by Moses (16) in which he was able to replace some of the heuristic work done by SAINT with algorithms which provided great improvements in speed. This implies that in at least some problems, if appropriate heuristic techniques can be developed, then perhaps after further study they can be (partially) replaced by algorithms which are actually more efficient.

## III. PROGRAMMING PROBLEMS NEEDING SOLVING

This section discusses a number of programming problems which need solution and which seem amenable to solution, or at least improvement, by various techniques common in artificial intelligence. It should not be assumed that all programming problems fall in this category. Section III. 1 indicates the characteristics which programming problems should (and should not) have to make them suitable for attack by artificial intelligence techniques. Section III. 2 then discusses some specific problems, divided into four main areas.

### III. 1 Characteristics of Programming Problems Which Make Them Suitable for AI Techniques

Although the phrase "artificial intelligence techniques" appears frequently in this paper, a definition of this phrase is deliberately being omitted. The reason is to avoid an argument on the definition or scope of artificial intelligence and its techniques. Intuitively what is being postulated is the situation in which a number of possible solutions for a problem (which itself may be large or small) are available, and at least one of these solutions is desired. In most cases, the "solution" is

required to be optimal according to some criteria. The facet of AI techniques which involve "self-improving" facilities in the programs is also needed. It is immaterial whether the AI approach is via "generality" (e.g., GPS) or "expertise" (e. g., SIN). It is inherent in this formulation that for some problems the solutions cannot be found at all, or cannot be found in a "reasonable" length of time. This has an effect in terms of limiting the class of problems which should be attacked.

The two basic characteristics which programming problems should have to make them suitable for the application of artificial intelligence techniques are as follows:

(1) The problem must be structured well enough so that a method for obtaining one or more basic solutions is known or can be developed by people knowledgeable about the problem (e. g., design of large data files).

(2) The problem should have a very large number of potential or feasible solutions (which may vary with time), but without a clear or easy or practical way of determining the best one (e. g., file layouts, scanning algorithms in compilers).

In addition to having these required characteristics, problems with one or more of the following elements are suitable:

(1) Individual cases or users should have individual treatment to achieve the best results (e. g., error checking of programs or data).

(2) Reorganization of the program or the system can improve efficiency or reduce errors, but this can't be determined until the program is developed and then it is too late to rewrite (e. g., any case where sequencing of computations has been specified but is not necessarily the most efficient).

(3) Individual modules or algorithms needed in the overall program are available but proper selection of the right one(s) is time-consuming and laborious and not obvious (e. g. , routines to access data, modules in a self-adjusting compiler).

It might be assumed that all programming problems fall into one or more of the above categories. This is not true, and there are certain characteristics which make a problem <u>unsuitable</u> for the application of current artificial intelligence techniques:

(1) Very broad problems which require intuition or vast experience to solve (e. g. , overall systems design for any very large program or application).

(2) Problems where the interaction of factors is not well defined (e. g. , language design).

(3) Problems where timing is critical and solutions must be reached in minutes or seconds (e. g., process control, traffic control).

(4) Problems where lives are at stake (e. g. , manned space flights, air traffic control).

## III. 2 <u>Specific Problems Suitable for Application of AI Techniques</u>

This section describes some specific programming problems which are amenable to solution or improvement through the use of AI techniques. This is a representative - but by no means an exhaustive - list of such problems.

The problems have been divided into four main areas: (1) data structure and organization, (2) program structure and organization, (3) improvement and correction of programs, and (4) language. It will be noted that in one guise or another the issue of language keeps cropping up. This is not merely because of personal predilections of the author, but because language is the means by which people communicate with each other and with the computer. If an idea is in the mind of a person, he needs a language (however feeble or inarticulate either the language or the idea is) to communicate it.

### III. 2.1 <u>Data Structure and Organization</u>

One of the major practical problems facing any organization is the handling of large quantities of data, commonly referred to as "data bases". This data can range from highly structured information such as personnel information such as personnel information (e. g. , name, address, social security number, job identification, salary, education, etc.) to more amorphous or frequently changing information (e. g. , the location of parts, finished goods, or delivery trucks, and the financial status of each of these). Furthermore, in todays environment where teleprocessing equipment and terminals are common, people in one part of a large compary want immediate - or at least rapid - access to this information. In the cases of the structured data this is not too difficult to do efficiently, but in the less well, or non-structured data, it is virtually impossible.

The problem involved is not how to find a way to structure the data. This is done in each case by the systems analysts, data base managers, programmers, and anyone else involved. However, the techniques they use are generally ad hoc and based on experience, intuition, or often just doing what seems easiest, even though analytic techniques are becoming available. The large numbers of access methods for data make it clear that the way in which data is prepared logically and then stored physically in a computer is not a stereotyped activity and requires careful consideration. The complexity of the problem is partially illustrated by the need for using a File Organization Evaluation Model (FOREM) to do a simulation of a parametric study of file design (see (13)). Thus, the number of possibilities for a file design is so large that even though the analytic techniques for doing a thorough study are available, the people and machine time required may be prohibitive. This is a case in which heuristic techniques might fruitfully be used to reduce the solution space so as to permit existing analytic techniques to be applied to a smaller (and hence more practical) number of cases.

In addition to the above problem, the artificial intelligence field should find ways of developing a system (i.e., program) which itself will determine the "best" way to store the data, depending upon its potential usage. The real key to this is the phrase "depending upon its potential usage". Information which in practise is used only in a batch environment, or by a very limited number of people at terminals, can be handled today. It is when the combinations of possibilities get very large that the difficulties set in. The problem has to be broken down into (a) specifying the ways in which the data is to be used, (b) describing the data, (c) specifying the constraints (which often means the objectives), and then allowing a program to produce the optimum data layout for the objectives specified. Furthermore, this program should include "learning" facilities so that the data can be automatically reorganized, based on practical experience by the users, or changes in the constraints or objectives. For example, a large company might set up a data management system based on the assumption that on-line access to the data would be equally required from many places in the country. Actual experience might show that only a few locations used the on-line facility, and a program with built-in learning facilities could monitor the usage and automatically readjust the physical data organization to produce greater efficiency. Similarly, the logical data structure could be changed based on usage.

A second major problem is so old it is a shame that it has not been attacked before. This involves the classical decision as to when to store information, and when to compute it. In the early days of computing, it was thought that tables for trigonometric routines should be stored. It was rapidly ascertained that computers had insufficient storage for all those numbers and the program as well, so the value of a particular trig function had to be calculated as needed. Now, with modern day computers whose storage capacities are orders of magnitude greater, it may be time to reevaluate this classical problem and solution. Again, this decision could be made by a program from specification of the given problem and the available equipment, and adjusted as necessary based on experience.

### III. 2. 2 Program Structure and Organization

All programming languages in use today tell the computer what to do and in what sequence to do it. They vary considerably in the amount of information supplied to the computer, and the level of detail in that sequencing. This is represented by the conclusion reached by thiB author that the definition of a "nonprocedural language" is relative to the state of the art of languages and compilers. The problem to be dealt with here is to allow both major and minor decisions of program organization to be made by an "intelligent" system. In the case of higher level languages and their compilers, many decisions are already made by the latter. For example, detailed code sequences, allocation of memory, and manipulation of registers are all decided by the compilers, but in a fairly rigid way. Several things need to be done by the compilers.

The compilers should be able to accept a much higher level of language and decide how to structure the program. For example, in the problem "CALCULATE THE SQUARE ROOT OF THE PRIME NUMBERS FROM 3 TO 99 AND PRINT IN TWO COLUMNS", there are essentially two main organizational approaches. One is to take each odd number in turn from 3 to 99, immediately test it for primeness, and then immediately calculate and print the square root for each prime number. The other major organization is to first test all the odd numbers and create a list of prime numbers, then take each prime number and produce the list of numbers to be printed, and finally to do the printing. It is not the least bit obvious which is the more

efficient organization since it depends entirely on the computer configuration. (This is a greatly simplified version of a very practical problem encountered by this author in which the first program structure was used whereas the second would have reduced the running time by a factor of 100. ) In cases like these the programmer should not have to (or be allowed to) specify the sequencing except where needed logically. Even when - or if - we get compilers which are capable of accepting the sentence cited above, it is unlikely that the compiler will do anything other than use one or the other of these organizational approaches, i. e. , the compiler will almost surely have a single built-in method whereas it should have heuristics to determine the best.

Two major programming efforts for systems programmers are compilers and operating systems. In both cases, a major design objective is modularity, i, e. , the program should be designed in small units each of which can be replaced without affecting others. Compilers are generally designed to achieve one major objective, e. g. , speed of compilation, speed of object code, minimum storage for object code, maximum error checking, etc. Seldom does the user have a choice. What needs to be tackled from the artificial intelligence view is to create a self-organizing compiler, i. e. , provide many modules in a compiler to do the same task with each using different techniques, and bring them together in an "intelligent" fashion, depending on the needs of the particular user and program to be compiled. Probably no two compilations would be done the same way, if this capability were available. The heuristic techniques can be applied in two ways. One is to select, as indicated above, the "best set" of modules for a particular compilation. The information needed for this selection would include constraints and priorities from the user, the compiler's knowledge of the current operating environment, history about that particular user, etc. An alternate use of heuristics would involve a "quick and dirty" scan of the source code combined with whatever information of the type above was available, and then production of the most effective compiler. This represents a compiler generator of a new type.

(The use of artificial intelligence to select the right modules from compilers should not be confused with the classical programming problem of developing module libraries, which has not yet been solved and for which AI is unlikely to be of much assistance. The problems in developing module libraries lie primarily in finding methods of describing the module so anyone can know which to select, and specifying interfaces which would allow modules developed by different people for different purposes to be pulled together in one program. )

In considering an operating system, the use of AI techniques would enable automatic reassignment of data in an installation to differing storage devices, with self-improvement of the system based on continuous changes in the individual programs, and experience from the general job stream. Furthermore, heuristic analysis of the job mix would enable frequent reorganization of the operating system to achieve the best performance for the individual installation.

Of the two major systems programming activities cited above, the application of AI techniques to compilers will be far easier initially because the compilers - although large - are an order of magnitude smaller than the operating systems, and are much better understood at this point in time. However, as an illustration of one small example of improvement in an operating system, the MULTICS system at Project MAC (5) is experimenting with inclusion of an algorithm for doing predictive paging, i. e. , guess which pages will be needed next based on the history of the particular program, and bring them in; their experience shows a slight gain in performance from doing this.

In a general sense, an "understanding" of <u>any</u> program would permit its reorganization in a manner most effective for the equipment (hardware and software) available. In a concrete situation involving parallel processors, rearrangement of the program by heuristic techniques would eliminate the need for special language features to denote parallelism. (Numerous proposals have been made for the latter; sre for example (21) and its bibliography)

### III. 2. 3 <u>Improvement and Correction of Programs</u>

For any program which solves a specific problem, it should be possible for another program to improve the first one, i.e., it should be possible to have the system rewrite the problem program for better efficiency. As an example of this, consider the earlier statement "CALCULATE THE SQUARE ROOT OF THE PRIME NUMBERS FROM 3 TO 99 AND PRINT IN TWO COLUMNS". If this is coded (in a current language) in either way indicated

earlier, then the system should be able to extract the meaning and intent and determine whether the alternative method of coding is better. As indicated earlier, optimization of programs for a parallel processor could be achieved this way. (A discussion of this issue, together with an illustration involving algorithms for Fibonacci numbers, is given by Minsky in his Turing lecture (15). )

The problem of finding and correcting errors in a program is in its infancy, although the finding *of* errors is orders of magnitude ahead of the correcting process. Artificial intelligence techniques need to be applied to the problem of finding what the error really is because then more can be done about fixing it. There are too many cases in which a person gets a compiler diagnostic that says "the parentheses are mismatched" when what really happened was that an illegal data name was used several lines earlier and that error cascaded into the symptom described to the user. It is probably true that the system has to really understand what the programmer had in mind before it can start truly identifying the errors, let alone fixing them. It is regrettable that the human can look at very many spelling errorrs and each them withoot any dificultu [sic] whereas the compiler gets confused by AD instead of ADD. Correcting these obvious (to the human) errors certainly requires artificial intelligence techniques of the highest order. The work of Teitelman (20) shows that much can be done, but it would be useful to many more people if this were done in a COBOL compiler.

Another problem which is considered here (although it also belongs under the heading of program structure and organization) is the one of program translation. For many years we have had systems that translate a program in one language (regardless of whether higher level or assembly) to another language, but only partially. That is, the translation can take place, and even with reasonable efficiency, up to some point (which differs in every case) which really depends upon knowing what the program is intended to do. Artificial intelligence techniques could be used here to great advantage by developing self-improving facilities based on intent of the program or programmer. In a practical situation the need is to translate a set of programs in a specific installation or written by a particular person. Since there are programming styles, a translator with self-improving facilities could "learn" or "be taught" which styles were being used and apply that knowledge to providing more efficient or more complete translations.

III.2.4 Language

In some sense it iB the area of language in which the most work has been done, at least by the criteria which includes the development of question answering systems as part of artificial intelligence. If we consider this author's long range goal of having each person able to communicate with a computer in the same way that he communicates with another person (1. e. , by natural language), then clearly this problem cannot be achieved without significant application of, and even advances in, artificial intelligence techniques. There is a fundamental difference between question answering systems with a limited discourse and true conversation, even though the conversation may be similarly limited in subject matter. A question answering system is easier to prepare because the structure of a question on a specific subject is somewhat limited, whereas ordinary conversation (even on a specific subject) is not. For example, if the subject is "Transportation by Plane", the question "WHAT PLANES LEAVE FROM BOSTON TO LONDON AFTER 5 PM?" is realistic whereas the question "DOES THE PLANE GO FASTER THAN A SHIP?" is unrealistic. However, the sentence "THE ADVANTAGE OF TRAVELING BY PLANE IS THAT IT IS MUCH FASTER THAN A SHIP" is reasonable in a conversation about transportation by plane. Thus, to enable realistic conversation with a computer will require greater depth and complexity of linguistic, heuristic and self-improving techniques.

## IV. SUMMARY AND CONCLUSIONS

This paper has attempted to present a challenge to the workers in the field of artificial intelligence to apply their techniques to realistic problems in the programming field, since virtually nothing practical has been done in this area to date. Characteristics for the types of programming problems which seem amenable to this approach were given. Some specific illustrations were provided under the broad headings of data structure and organization, program structure and organization, improvements and corrections of programs, and language.

## References

(1) Arnold, R. F. "A Compiler Capable of Learning", Proc. Western Joint Comp. Conf. , San Francisco, March, 1959.

(2) Bobrow, D. G. "A Question-Answering System for High School Algebra Word Problems", Proc. FJCC, Vol. 26, Part 1, 1964.

(3) Buchanan, B. G. , Sutherland, G. L. , and Feigenbaum, E. A. "Rediscovering Some Problems of Artificial Intelligence in the Context of Organic Chemistry", in Machine Intelligence 5, American Elsevier Publ. Co. , 1970.

(4) Campaigne, H. "Some Experiments in Machine Learning", Proc. Western Joint Comp. Conf. , San Francisco, March, 1959.

(5) Corbato, F. J. et al . Six papers in session "A New Remote Accessed Man-Machine System", Proc. FJCC. , 1965.

(6) Fikes, R. E. "REF-ARF: A System for Solving Problems Stated as Procedures Artificial Intelligence Journal, Vol. 1, Nos. 1/2, Spring 1970.

(7) Friedberg, R. M. "A Learning Machine, Part I", IBM J. Research & Development Vol. 2, Jan., 1958.

(8) Friedberg, R. M. , Dunham, B. , and North, H.J. "A Learning Machine, Part II", IBM J. Research & Development, Vol. 3, July, 1959.

(9) Green, C. "Application of Theorem Proving to Problem Solving", in (2 5).

(10) Huesmann, L.R. "A Study of Heuristic Learning Methods for Optimization Tasks Requiring a Sequence of Decisions", Proc. SJCC, 1970.

(11) Kilburn, T. , Grimsdale, R. L. , and Sumner, F. H. (1959). "Experiments in Machine Learning and Thinking" Proc. International Conf. on Information Processing, UNESCO, Paris, Butterworths, London.

(12) London, R. L. "Bibliography on Proving the Correctness of Computer Programs", in Machine Intelligence 5, Edinburgh University Press, 1970.

(13) Lum, V. Y. , Ling, H. , and Senko, M. E. "Analysis of a Complex Data Management Access Method by Simulation Modeling", Proc. SJCC, 1970.

(14) Manna, Z. , and Waldinger, R.J. "Toward Automatic Program Synthesis", CACM, Vol. 14, No. 3, March, 1971

(15) Minsky, M. "Form and Content in Computer Science", JACM, Vol. 17, No. 2, April, 1970.

(16) Moses, J. "Symbolic Integration", MIT, Project MAC, Cambridge, Mass. , MAC-TR-36 (Ph. D. Thesis), 1967.

(17) Simon, H. "Experiments with a Heuristic Compiler", JACM, Vol. 10, No. 4, October, 1963.

(18) Slagle, J. R. "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus", JACM, Vol. 10, No. 4, October, 1963.

(19) Slagle, J. R. "Experiments with a Deductive Question-Answering Program", CACM, Vol. 8, No. 12, December, 1965.

(20) Teitelman, W. "Toward a Programming Laboratory", in (25).

(21) Tesler, L. G. , and Enea, H. J. "A Language Design for Concurrent Processes", Proc. SJCC , 1968.

(22) Tonge, F. M. "Summary of a Heuristic Line Balancing Procedure", in Computers and Thought, McGraw-Hill, 1963.

(23) Waldinger, R.J. "Constructing Programs Automatically Using Theorem Proving", Ph.D. Thesis, Carnegie-Mellon University, 1969.

(24) Waldinger, R. J. , and Lee, R. C. T. "PROW: A Step Toward Automatic Program Writing", in (25).

(25) Walker, D. E. , and Norton, L. M. (Eds). Proc. International Joint Conf. on Artificial Intelligence , Washington, D. C. , 1969.