THE APPLICATION OF ARTIFICIAL INTELLIGENCE TO OATA BASE MANAGEMENT

Rob GerrItsen
The Wharton School (*)
University of Pennsylvania
Philadelphia, Pennsylvania

Abstract

An application of Artificial IntelllgenceIs discussed: an automatic programming system that generates Information retrleval programs and data base structure designs for highly structured or network data bases. It Is claimed that these applications *are* unusual In that they have more practical value than Is usually the case, being of Immediate utility to commerclal data base management. The paper concludes wi th a short reflection on the problems associated with the representation and acquisition of knowledge for problem solvIng programs. In particular, representation of knowledge Is Important for efficiency of the programs, yet an efficient representation may not be "natural". It was difficult to separate representation from content/ and successful application required that the expert providing the knowledge also know how the program uses that knowledge.

CR categories: 3.5, 3.6U, 3.7, U.2, 5.23

Key words and phrases: application of Artificial Intelligence, automatic programming, procedure generation, information retrieval, network data base. Data Base Task Group, data base design, non-procedural languages.

Introduction

Most successful Artificial Intelligence programs exercise their problem solving ability In non-realistic environments or apply it to a game playing task, but th Is state of affalrs does not constitute a basis for criticism of Al. As many of the proponents of Al have maIntaIned, It Is necessary for Al to cut Its teeth on toy problems, games, and In otherwise limited environments before It can proceed to fulf111 Its high1y touted promIses. Perhaps the only fair criticism is that this teeth cutting Is taking longer than envisioned by early workers In the field.

Al is entering an age where practical and objective application Is feasible. This Is not to say that there have not been very valuable spin-offs from Al research that have contributed to advances In fields such as computer science, linguistics/ and psychology. However, there have not been many direct applications of Al technologies (with perhaps the exception of pattern recognition) to problem areas outside the research laboratory.

My background is not primarily In Al, but lies more in the area of commercla? data processing, data base management, and Information retrieval. My experience In these areas predates by several years my Introductory exposure to Al. This fact Is relevant because the problems, whlch will be descrI bed shortly, existed for me long before I dIscovered the technology to solve them. As such, my clrcumstances provIded an amount of practical experience not typically possessed by the Al researcher who Is lookIng for problems to adapt to his technology.

The application described here Is a practleal one. The problem solvIng systems that have been developed can supplant or augment some of the tasks currently being performed by programmers and data base administrators.

Data Management Application

The subject of this paper Is the successful application of Al, particularly automatic programming technology, to the problems of Information retrieval and data base design. This has resulted In a system<l,2,3> that generates Information retr1eval programs and data structure declarations.

These two problem solving tasks have been around for a long time, but growing commercial usage of larger and more complex network data bases has suddenly Increased their significance. CAs used here, the term "network data base" refers to a data base In which a data I tern may be linked to many other data Items. Our subject is not data bases on computer networks.) There has been an attempt within the United States and Canada to develop a standard for network data bases through CODASYL which spawned the Data Base Task Group (DBTG) involving both industry and academia. This group has produced a spec If i cat Ion for a general purpose network data base system<U>. Although not yet accepted as a standard. It is believed by many that the DBTG specification will be so adopted, perhaps

* A major portion of the work described herein was done while the author was at Carnegie-Mel Ion UnIvers!ty, PIttsburgh, Pennsylvania.

by default since many computer manufacturers and software firms are now selling Implementations of the DBTG specification.

Both applications described herein function In the data base environment as specified by the DBTG. That is, data structures designed by the system conform to DBTG specifications, and the Information retrieval programs generated by It are COBOL Procedure Divisions containing Data Manipulation Language (DML), as defined by the DBTG.

Programming for Network Data Bases

Introduction of the network type of data base brings with It a new level of complexity for the business programmer. Earley<5> points out that Instead of the usual logical-physlcal dichotomy that a programmer traditionally deals with, there Is a logical(a)-logleal(b)-physlcal trichotomy facing the network data base programmer. The loglcal(a) level corresponds to data relationships as perceived by the data user. The loglcal(b) level concerns the actual 1inks between the data In the data base.

Bachman<6> recognizes this complexity in the 1973 ACM Turing Award Lecture where he likens the programmer's problem (In the network data base environment) to that of a navigator. The programmer Is given the user's query requirements; his task Is to find the proper access path to the data through the myriad of connective links In the data base.

The programmer's task should be reduced. After all/ much Its complexity Is caused by an artlfact 1ntroduced by the data network, the data 1lnks. It should be noted here that others In the field, prlmarlly Codd<7,8,9>, have proposed relational models of data bases whereby they hope to avoid this artifact. It is not clear, however, how this can be done without exactly the kind of problem solving program that is the subject of this paper.

The technology used to reduce this task comes from work in the area of automatic programming. A compitier developed by Buchanan and Luckham(10,11> was used to create the Information Retrieval Program Generator.

Programming Rules

The Information Retrieval Program Generator is compiled from a set of rules. These rules are stated in a formal ism that closely corresponds to the formalism developed by Hoare<12> to describe the logic of programs. Each such rule describes some type of program construct, be It a loop, assignment statement, a DML statement, or other program construct*

One type of rule describes the conditions under which a program statement may be used/ and the effect of Its use. For example, the rule for a "GET" statement says that It Is not possible to GET a record unless Its location In the data base Is known to the data management system, and that the object record will be located In working storage following execution of the GET.

This rule Is formally stated (to the compller) as:

(CURRENT RI) I IGET RI|| (INCORE RI)

In this rule RI Is a variable (for record name) and (CURRENT RI) and (INCORE RI) are predicates. Henceforth we use the convention that RI/ R2/ XI, II, etc., are varlable names.

Other rules do not bear such close resemblance to a description one might find in a programming manual as does the rule for GET. These other rules capture my own programming know!edge, IncludIng heurIs!tles, that I had prevlously acquired and developed as a professional programmer.

One such rule Is the rule that describes the top level composition of a program:

(OPENED A1)A(CURRENT R1)A(LINKED RI R2)A
(CLOSED A1)A(ST0P XI)
--> (PROGRAM XI)

This rule is different than the rule for GET: it does not define a program statement, rather It Is an Implication stated with predicates. The order of the predicates Is obviously Important.

A semantic Interpretation of this rule Is that a program consists of five blocks that : (1) open the required files of the data base, (2) locate the first record of Interest, (3) follow the access path from this record to all other records of interest, (U) close the files opened above, and (5) terminate execution.

Of these five blocks, the second and especially the third, are the most complex. They may Involve many further levels of refinement, revealing Iterations, alternations and compositions. The third block will include the processing of the records on the access path to display values, calculate statistics, etc.

Translation to Micro-Planner Theorems

The rules are translated by the Buchanan-Luckham compiler to Micro-Planner<13,14> theorems that constitute the Information Retrieval Program Generator.

For example, the GET rule Is complled to:

```
(DEFPROP GET
  (THCONSE (R1) (INCORE (THV R1)
    (THGOAL (CURRENT (THV R1)))
    (THSET (CAR (THV ANS>)
      (CONS (CONS (QUOTE GET)
                  (LIST (THV RI)))
        (EVAL (CAR (THV ANS)))))
    (THASSERT (INCORE (THV RI))))
THEOREM)
```

The theorem presented here Is quite condensed from the one actually generated, which also contains bookkeeping, tracing, uniqueness, and other special functions. Note that the pre-condition of the rule, (CURRENT RI), Is a goal In the theorem. Similarly, the post-condition, (INCORE RI), Is defined as the consequence of the theorem and will be asserted when the theorem Is used. When the theorem Is used 'GET R1' will be catenated onto the ANSwer.

Program construction proceeds In a sub-goallng fashion. For example. If UNCORE PATIENT) Is a goal then the GET theorem Is tried, and (CURRENT PATIENT) becomes a sub-goal. A sub-goal may Invoke other theorems In turn, generatlng further sub-goals. This process generates a goal tree with leaf nodes which are true or false In a set of assertions known as the "state".

The Initial state Is the only input to the program generation process and Is made up of two major subsets of assertions. One such subset describes the structure of the data base, the other describes the query for which program generation Is required.

The description of one more rule will give us a sufficient number of rules for an example. A rule for FINDing records:

(HASHKEY R1 K1)A(DESIRE K1 V1)A
(CONTAINS KI V1) I IFIND RI|| (CURRENT RI)

This rule Is similar to the rule for GET; It defines a program statement. Again, It corresponds closely to a paragraph In a programming manual: To FIND a record, first initialize the key for that record with the value Identifying the desired record. Following the execution of FIND, the record is current to the data management system. Current Is a DBTG term meaning that a record's exact location In the data base is presently know to the data management system,

Example

Assume that a data base definition specifies that STUDENT records *are* (hash)keyed on SNUM. If a query asks for Information regarding the student having number (SNUM) 126, then two of the assertions in the initial state will be:

```
(HASHKEY STUDENT SNUM)
(DESIRE SNUM 126).
```

The first assertion belongs to the set defining the data base, the second to the set defining the query.

Micro-Planner is given the goal (PROGRAM EXAMPLE), and program generation commences. The first sub-goal, (OPENED A1), Is satisfied with a set of rules not illustrated here resulting In an initial block of procedure to open the flles.

The next sub-goal is (CURRENT RI). Since the rule for FIND has this as a consequence,it Is tried. The first two sub-goals In this rule are immedlately true in the state, and the variables RI, KI, and V1 get bound to STUDENT, SNUM, and 126, respect!vely. The third sub-goal Is satisfied with the assignment statement rule (not lllustrated here), and an assignment statement is Inserted In the program. The pre-condition now being true, 'FIND STUDENT' is put into the program.

At this stage the program appears as :

```
(Block to open files)
MOVE 126 TO SNUM.
FIND STUDENT.
```

The state (set of assertions) has also been altered and now contains an additional assertion, (CURRENT STUDENT), because of the FIND rule. This may affect subsequent procedure generation. For example. if the query requires that values from the STUDENT record be In working-storage, the system will not generate any procedure for the (CURRENT RI) sub-goal In the GET rule.

Designing Network Data Structures

Another problem attendant to data base management Is data base structure design, that is, the design of appropriate links between data elements so that data relationships can be properly captured and reconstructed. Since all such relationships must be Implicitly contained In the set of all retrieval requests. It should be possible to derive a data base structure from such a set<3>.

The Automatic Data Structure DesIgner operates in exactly this fashion. It generates a data structure design that Is satisfactory for a set of queries. The resulting structure Is not optimal but has the characteristic that data redundancy Is minimized (and as a corrolary, the number of links are maximized).

The design program has two major tasks: to design a network structure that can accomodate all of the hierarchical relationships referenced in the queries, and to determine the best location In the structure for each data I tern.

Detection of a hierarchical relationship in a query is not difficult. The system simply looks for quantifiers or summarizing commands such as "all," "any," "average," "total," etc. For example, a query asking for a class list (all values of student-name when class = "history") suggests a hierarchical relationship between class and student-name. Similarly, average grade for student-name = "Anderson", suggests that there may be many values for grade for a given value of student-name.

A designer has a problem if, for example. In addition to the class list discussed above, a second query asks for a student transcript (all values of class when student-name ="Anderson"). The two queries' views of the data base are Inverted with respect to each other.

Inciuded in the Automat!c Design System are rules that recognize such situations. These rules will insure that a special linking record Is included in the data base design. Through this record will pass two linked lists, one for each hierarchy.

The second principal task of the Designer Is to determine the location of data iterns (attributes) In the hierarchical structure. Querles may differ in their views of attribute associations. For example, hospital name may be viewed as a patient attribute In one query (where he's being treated), as a doctor attribute (where he works), or as a hospital attribute in other queries. If hospital-patient and hospital-doctor hierarchies (several patients and several doctors per hospital) have already been constructed by the system, then it will assign the attribute In question to the hospital record. This permits usage of hospital name as a unique attribute of hospital, doctor, and patient.

Using Earley's terminology, we can say that the Automatic Data Structure Designer finds a logical(b) arrangement that accomodates all loglcal(a) relationships. As a matter of fact, translation from loglcal(a) to logical(b) also characterizes the system's activities as a programmer. Further translation to the physlcal level is not the task of these programs. That activity is performed by the data management system, e.g. an Implementation of the DBTG specification.

As was the case with the Information Retrieval Program Generator, the Automatic Data Structure Designer Is defined with a set of rules that are translated by a compller to a program of Micro-Planner theorems. However, in this case the resulting system Is not a code generator, it is a declarative generator.

Representation of Knowledge

If a system Is to apply knowledge. It must of course first obtain such knowiedge. Untll generalized learning systems become available that *can* be adapted In a practical way for use within a particular application domain. It Is necessary to imbed knowledge of the application either by building a specific learner for that system or by providing the system directly with the knowledge It needs. The trouble with the *former* approach is that building the learner may turn out to be impractically difficult. The latter approach Is the one that was taken for the applications discussed here.

Complexity of Knowledge

The rule formalism provided a good framework for capturing network data base programmlng and deslgn knowledge. Even so, capturing the knowledge involved many trlal-and-error Iterations because much of It does not have a well understood structure. Furthermore, It Is possible to map the knowledge into a structured set of formal rules in several valid ways. These representations may differ with respect to efficiency.

However, some of the knowledge does have a well-understood structure, for example the knowledge regarding GET or FIND as discussed above. In general, the rules for single program statements like GET are similar to usage rules that might be found In a reference manual. it is the rules that control the next higher level of programming, the construction of program blocks, that are complex and not easlly derlved.

Such rules were further complicated by considerations of efficiency both for the generated programs and for the program generation process Itself.

As an example we Illustrate the two rules that control code generation for alternation. Alternation occurs In the generated program whenever the query specifies conditional processing. During program generation, the condition that Is to be tested In the generated program is contained in a list, FORMS, in dIsjunctIve form.

For example, the list <<<GT A B)(GT B C)) ((LT A 26))) means (A>B and B>D) or A<26.

The two rules are:

```
(NULL FORLIS)V
(CONJUNCT ACTION (CAR FORLIS) DUM)A
(DISJUNCT ACTION (CDR FORLIS))
    —> (DISJUNCT ACTION FORLIS)

((NULL C0NJ)A(ACT ACTION)A(SETQ DUM 0))V
(SETQ REL (CAAR CONJ))A
(SETQ ITM1 (CADAR CONJ))A
(SETQ ITM2 (CADDAR CONJ))A
(DETVAL ITM2)A(DETVAL ITMDA
(TEST (LIST REL ITM1 ITM2))A
(CONJUNCT ACTION (CDR CONJ) DUM)
    --> (CONJUNCT ACTION CONJ DUM)
```

Those readers who are programmers will agree that these rules bear little correspondence to their own programming knowledge, el ther vlewed IntrospectIvely, or as such knowledge might be represented In a programming text.

The first rule CDR's through FORLIS, each time picking off the CAR, which Is a conjunctive list of predicates. The CAR of FORLIS will be bound to CONJ In the second rule.

The second rule CDR's through CONJ until It becomes NULL, at which point the processing code can be generated (accomplished by invoking the ACT rule, not shown here).

Each time the second rule is used, code is generated to determine the values of the two i terns In the test described by the CAR of CONJ. Then the test itself is Inserted In the generated program.

For the example of FORLIS given above, the generated procedure has the follow!ng form:

```
(block to determine value of B)
(block to determine value of A)
IF A>B THEN
   BEGIN
   (block to determine value of C)
   IF B>C THEN
      BEGIN
      (block of conditional
      processIng)
      END
   ELSE PR0C2
   END
ELSE PR0C3

PR0C2:  (and PR0C3)
BEGIN
IF A<6 THEN
   BEGIN
   (block of conditional processing)
   END
END
```

The Buchanan-Luckham compller takes special care with rules that may generate alternations. Unbound variables become important, and it is for this reason that DUM is included iIn the second rule. For further detils see <1,10,11>.

Tests *are* separated in the generated procedure to increase runtime efficiency. in the example above, the value of C Is determined only If A>B. Since such a value determination may require a lot of processing, It is best to avoid It if possIble.

Impact of Represe ntation on Efficiency

Since the content of the rules determines the decision tree that controls the search space, the choice of mapplngs mentloned above can have a dramatic impact on the efficiency of the search for a program. If the use of a particular rule occurs far out in the branches of the tree, and If the rule Is a powerful discriminator, then It would be appropriate to consider an alternative rule structure that would permit earlier use of the rule in question.

It is also Important to direct the Information Retrieval Program Generator as much as possible. Consider the FIND rule previously discussed. This rule states that FIND has the effect of making a record current. There are In fact four other rules describing statements that have the same effect. These rules all differ in the way in which they make records current: some use polnters, another uti1lzes sequentlal access, etc.

Since there are five ways of making a record current/ Micro-Planner may try several Inappropriate rules to satisfy a CURRENT goal before the proper one Is attempted* This can be very expensive since an Inappropriate rule may not be *discarded* until a very large subtree has been evaluated.

Frequentiy, such searches can be eliminated because the programming context will determine which type of FIND Is needed. In the Iteration step of a loop, for example, we would not expect a direct FIND, rather we would expect a FIND for the next record on a list or the next record in sequence.

By changing the rules so that the system is directed to a specific rule whenever possible, we were obviously able to avoid a lot of unnecessary searching.

Che it is important to note that the rules were correct without these specific directions. The way in which the rules were used requlred add!tlonal knowledge about the application so that the rules could be used efficiently.

For a particular set of programs that the system generated. it tried (on average) about 94 rules per program. Of these (on average) only 13 rules were inappropriately tried, resulting in backtracking to use alternative rules.

Although we have no exact figures to compare with, the system was trying 10 to 20 times as many unnecessary rules prior to the addition of more specific direction. The difference Is multiplicatlve rather than addltive because the sub-tree associated with a single inappropriate rule may be quite large.

Conclusion

As a user of Al technology, I am somewhat dlsappolnted that It was not sufficient to directly transfer my knowledge to the machine. In Itself a difficult task. It was also necessary to observe how that knowledge was being used by the machine so that I could change and augment the representation for more efficient use of the knowledge.

These problems were also encountered while building the Automatic Data Structure Designer. Another dlfficulty arose as well. As the Designer took shape It became apparent that my design knowledge contained many ad hoc techniques that were not easily captured in a set of general rules. This realization led me to develop the rules for the more direct, general purpose algorithm that is embedded in the Designer.

Although the difficulties of transferring knowledge to the machine were somewhat frustrating, the process (and the machine as a mirror of myself) also led to new Insights Into the problem domain.

Acknowledgements

The assistance and encouragement of my thesis advisor. Jack R. Buchanan, is gratefully acknowledged, as are the suggestions of one of the referees.

This research was supported n part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract Fi44620-73-C-0074, and in part by the Office of Naval Research under contract M0001U-75-C-0W62. I am also Indebted to the William Larimer Mellon Fellowshlp, which provided personal funding durlng the course of this research.

References

1. Gerrltsen, Rob, "Understanding Data Structures," PhD Thesis, Carnegie-Mellon Unlverslty, Plttsburgh, Pennsylvania, 1975.
2. Gerrltsen, Rob, "The Relational and Network Models of Data Bases: Bridging the Gap," Proceedings 2nd USA-Japan Computer Conference, 1975.
3. Gerrltsen, RoE^ "TS Prel Imlnary System for the Design of DBTG Data Structures," will appear In CACM, October 1975.

i*. CODASYL, CODASYL Data Base Task Group Aprl1 71 Report, avallable from ACM, New York City.

5. Earley, J., "On the Semantics of Data Structures," in RustIn, R. (Ed), Haia____Base Systems. Prentice Hall, 1972.
6. Bachman, C. W., "The Programmer as Navigator," CACM 16. 11, November 1973, pp653-658.
7. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," CACM 13, 6, June 1970, pp377-387.
8. Codd, E. F. and C. J. Date, "Interactive Support for Non-programmers: the Relational and Network Approaches," IBM Research RJ-1U00, June 6, 1971*.
9. Date, C. J. and E. F. Codd, "The Relational and Network Approaches: Comparl son of the ApplFeat lon Programming Interfaces," Proceedings 197tt ACM-SIGFIDET Workshop.
10. Buchanan, J. R. and D. C. Luckham, "On Automat!ng the Construction of Programs," Stanford Al Project Memo, Stanford Unlvers!ty, Californla, 197U.
11. Buchanan, J. R., "A Study In

Automatic Programming," PhD Thesis, Stanford ` California, 1974.
12. Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," CACM12, 10, October 1969, pp576-580.
13. Hewl tt, C, "Descrl ptlon and Theoretical Analysis of Planner," PhD Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1971.
14. Sussman, Gerald J. and Terry Wlnograd, "Micro-Planner Reference Manual," Project MAC Report, Massachusetts Institute of Technology, CambrIdge, Massachusetts, 197 2.