Proc. Natl. Acad. Sci. USA
Vol. 77, No. 11, pp. 6264–6268, November 1980
Applied Mathematical Sciences

Computers—Non-numerical computation*

HERBERT A. SIMON

Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213

Contributed by Herbert A. Simon, July 18, 1980

The question posed by my title is: If not numbers, then what? The computer was invented as a number cruncher but, in order to perform numerical tasks efficiently, it had to be given important non-numerical capabilities. I will indicate why this had to be done and what the consequences have been.

Numbers—more precisely, numerals—are patterns of varied shapes. We distinguish the numeral 2 from the numeral 3 by noting these differences in shape. Of course, to a computer, the shapes are not what they are to us. If you open the cover of a computer, you will not find a pattern inside in the shape of the numeral 3. Nevertheless, the computer must be able to make the same kind of discrimination—among electromagnetic patterns, say—as we make when we recognize a 3. The computer can make tests to determine whether two patterns or symbols presented to it are equal symbols or unequal symbols—that is, whether they are both instances of the numeral 2 or are instances of different numerals, of the letter "e" or different letters, of the same English word or different English words.

It is essential that the computer be able to test equality and inequality of symbols, and the symbols that the computer compares and manipulates can denote anything—letters, words, numbers. A computer is an example of a physical symbol system, a system capable of manipulating symbols that are represented in some physical medium, whether it be steel or silicon or neuronal tissue.

COMPUTER CAPABILITIES

A computer can perform a few basic operations (1). It can read symbols from external sources and register them internally. It can write symbols, putting them out onto display devices. It can store symbols for indefinite periods. It can copy symbols from one storage location to another. As I mentioned, it can compare symbols and then—a very important capability—it can proceed along one path if it finds the symbols compared to be the same, and another path if they are different. This last is the *branch* operation familiar to all of us who have worked with computers.

If a symbol stored in the computer represents an instruction instead of data, the computer must be able to understand that symbol and to interpret it. Whatever the computer programming language and whatever the electromagnetic mode of memory storage, when we give the computer the instruction "add 3 and 3," we expect it to answer "6." This means that it has to understand the meaning of the instruction word "add," however that word may be symbolized and in whatever programming language.

Early in the history of the development of the modern computer, the advantage was seen of storing the program in the computer memory in just the same way as data were stored. Instead of following the model of the Jacquard loom, in which the program controlling its operations is recorded on a long cardboard strip outside the machine, the designers of the computer saw that greater speed could be achieved with an internally stored program. It was this insight that unintendedly gave the modern computer its universality and, in particular, its capabilities for non-numerical computation.

Except for important differences in speed and memory capacity outlined by Oliver (2), the modern computer (which means any computer built in the past 25 years) is not fundamentally different from the early computers that were designed at the University of Pennsylvania or by John von Neumann and his associates. If you store the program in exactly the same way as you store the data, then program and data inside the computer become interchangeable. You can operate on data as though they were program, and you can operate on and change programs as though they were data. This has important consequences.

Provided a computer meets the specifications I have outlined—the ability to perform basic symbol manipulations and to store its programs in memory—its hardware specifications are irrelevant for our purposes. In principle, you can do exactly the same things with the computer regardless of whether it is assembled from relays, tubes, germanium diodes, transistors, chips, or, some of us think, neurons.

SYMBOL SYSTEM HYPOTHESIS

The remainder of my remarks are based on a hypothesis usually described as the "physical symbol system hypothesis." The hypothesis is that a physical system can exhibit intelligence if, and only if, it is a properly programmed physical symbol system (3). By physical symbol system, I mean a system that has the basic capabilities, just described, of a modern computer.

The term "intelligence" in the hypothesis also needs definition. Intelligence, say psychologists, is what intelligence tests measure. I think we can improve on that definition. A system is intelligent if, when given problems that we and other human beings regard as difficult, it is able to discover solutions for those problems. Clearly, intelligence is not a unitary thing. A system, including ourselves, might be able to solve problems of one kind and completely unable to solve problems of another kind. Hence, a physical symbol system might exhibit intelligence in one problem domain and not in another. But if the physical symbol system hypothesis is correct, this limitation is a matter of programming and is not intrinsic.

The hypothesis is an empirical hypothesis. It is an assertion of what an actual physical system can or cannot do. The usual way in which we test such hypotheses is by designing and building the systems in question, presenting them with tasks whose performance would illustrate various kinds of intelligence, and measuring the speed and quality of this performance. The hypothesis does not postulate that if a system exhibits intelligence in a particular area the processes and programs that made the performance possible are unique. There may be a number of equivalent ways of solving the same problem. Some of them might require extensive manipulation

* Presented on 21 April 1980 at the Annual Meeting of the National Academy of Sciences of the United States of America.

of symbols whereas others, which we would regard as the more elegant and clever, would require only a modest amount of processing.

The physical symbol system hypothesis might be called the basic hypothesis of artificial intelligence research. It underlies all efforts to induce computers to do intelligent things, whether they accomplish this with the help of clever tricks or mainly by the use of brute computational force.

There is a stronger form of the hypothesis: a physical symbol system not only can exhibit intelligence but also can do so by adopting the same methods that human beings adopt in solving problems in the same problem domains—that is, that one can program a computer to simulate human cognitive processes. Thus, non-numerical computation has two main branches. The first, usually called "artificial intelligence," is aimed at programming computers to behave intelligently. The second, usually called "cognitive simulation," is aimed at achieving artificial intelligence in a humanoid way, by simulating human methods.

PROGRAMMING LANGUAGES

Schwartz (4) pointed out how critical it is for us, in making use of the new capabilities of the ever more powerful computers that become available to us, to deal effectively with the programming problem. To the degree that the research program of artificial intelligence is successful, we will find ways, as we already have to some extent, of shifting the burden of programming to the computer itself. Some examples of how this has been done have already been mentioned in connection with the topic of automated design. Research in artificial intelligence and non-numerical computation has also addressed the programming problem.

It was recognized very early, simultaneously with the development of the first high-level programming languages, that if you are to get a computer to do tasks that require it to explore in unpredictable directions, you must develop software techniques for organizing memories to accommodate the unpredictable or at least the unpredicted. A chess-playing computer, for example, must be able to store in memory the tree of positions it generates by searching out moves and countermoves. Because the shape of that search tree cannot be anticipated because certain branches may grow very large, the usual techniques for organizing computer memory by preassigning specific blocks to specific foreseen uses will not work. So a technique had to be developed in the middle 1950s, at the same time that FORTRAN-like languages were being developed, to handle irregularly shaped search trees. Because the languages and storage schemes had to permit search in different directions without knowing in advance what the shape of the tree was going to be, you could not preallocate memory. It is a little like the problem the Lord had in designing our own memories. He did not know whether we were going to speak Hebrew or Greek, and therefore He could not allocate particular parts of memory to the one language and other parts to the other. Memory had to be sufficiently flexible so that it could hold either one of these languages, or Chinese or some other.

The particular solution that was found for this problem for artificial intelligence purposes, but which also found other important applications in computing, is called "list processing" (1, 5). It is a software technique that can be used with standard computer hardware. The first list-processing languages were a series called IPL (Information Processing Language); the mostly widely used today is a language called LISP.

A list-processing language stores information in memory in ordered sets—lists—that need not be located sequentially in the hardware memory but may be distributed about in a wholly arbitrary way. The ordering is defined by storing with each symbol a pointer that designates the memory location of the next symbol on the list. The programming language contains instructions for finding the symbol that is next on a list to a given symbol, inserting a symbol in a list in an arbitrary location, finding a symbol in a list, and the like.

Up to the present, this kind of flexibility in the organization of computer memories has been accomplished through list-processing languages which have to be painfully interpreted or translated by the computer, instruction by instruction, into its underlying machine instructions. Although some of the efficiency that is lost by this interpretive procedure can be recovered by compiling the program in advance of execution, even with compilation, the use of list-processing languages reduces by an order of magnitude the speed of a computation (that is, of any computation that could readily be carried out without using such techniques). At the present time, several groups are designing computers that will have list-processing capabilities built into hardware, with a consequent major gain in speed of execution.

In considerable part, the idea of organizing memories as list structures was borrowed from our knowledge of the organization of human memory, in particular the fact that human memory is associative. Memory can be thought of as an assemblage of lists or a collection of nodes with links connecting them. The human memory is also a little like an encyclopedia with a very large body of text, in order to find the text that you want to read, you have to use an elaborate index. No matter where items are located, you can find them if appropriate index entries are associated with them. List-processing techniques have been developed for generating such indexes automatically so that the programmer does not have to pay explicit attention to where information is stored (6). It is like a looseleaf encyclopedia in which new items can be added anywhere at any time.

These are some of the ways in which programming methods developed for use in non-numerical computation have enabled us to borrow some of the organizational techniques that are used to store and process symbols in the human brain. Because these techniques are implementable in software, they depend on neither the hardware of a particular computer nor the neurophysiology of the brain. That is fortunate, because we know very little about the latter as it relates to the implementation of associative storage or processes.

PRODUCTION SYSTEMS

Another class of technical programming issues that had to be addressed before non-numerical computation could go very far included those concerned with controlling the path of computation—i.e., how the system can decide what it will do next. Here, the early ideas were quite parallel with those that were adopted for algebraic computation in the FORTRAN-like languages. One of these was the idea of being able to define subroutines. For example, in a program for filling out an income tax return, one subroutine might do some additions and subtractions to calculate adjusted gross income. Then the system should be able to incorporate this subroutine and others into large and larger structures until the whole hierarchy of closed subroutines is capable of computing the tax. The idea of writing programs as such hierarchies (sometimes referred to as "structured programming") received much of its early impetus from the research on list-processing languages.

The idea of subroutine hierarchies is widely used, but it turns out not to provide all the flexibility that is needed for the convenient programming of artificial intelligence systems. In particular, it creates too rigid a control over the course of the computation, so that inappropriate actions are taken when events occur that were not foreseen or prepared for. An alternative programming organization, production systems (1), is now becoming more and more popular in artificial intelligence.

On the theoretical side, the idea of production systems predates the invention of the modern computer, going back to the logicians Post, Church, Turing, and Markov. The idea is that you can build a perfectly general programming language ("perfectly general" means one in which anything can be programmed that can be programmed in another computer language) in the form of lists of homogeneous instructions called "productions."

In a production system, all of the instructions are in one basic form: a set of conditions (C) followed by actions (A), C → A. The rule of operation is that, whenever the conditions of a production are satisfied, the actions of the production are executed. Various rules may be introduced to resolve the conflict that arises when the conditions of two or more productions are satisfied simultaneously. One rule imposes a prespecified priority ranking among the productions; another requires that productions with stronger conditions be executed before productions with weaker conditions. These details will not concern us here.

A production system for driving an automobile might contain a production, "if there is a red light → stop." Another might specify, "if there is a green light → proceed." If the former production were given priority over the latter—the safe arrangement—a combination of red and green lights would bring the car to a halt.

Evidence is gradually accumulating that the control of human behavior is managed by something that resembles a production system. When they are carrying out cognitive tasks, people are sensitive to two kinds of things. They are sensitive to some aspects of the visual and auditory stimuli impinging on them—stimuli that might include the marks on a sheet of paper on which they are writing down numbers or words. Second, their behavior is responsive to goals. The behavior must be oriented to the present state of affairs as revealed through the senses and it must be goal-oriented if it is to be rational, purposive behavior.

By incorporating in a production both conditions that test for features of the external world and features that test for the presence of goals, we can bring about actions that are appropriate both to the situation in which a system finds itself and to the goals that it has stored in memory. Programs have been written for a wide range of task domains—solving problems in physics or geometry, making medical diagnoses, to mention a couple—that carry out successful problem-solving searches under the control of production systems.

MEANS/ENDS ANALYSIS

Having sketched some of the programming techniques that are used in artificial intelligence and cognitive simulation, let me turn to some specific examples of applications of these techniques. One interesting area of application, just beginning to find practical use in industry, is the automatic design of paths for chemical synthesis. Suppose we want to create a particular kind of molecule. We have available a supply of reagents and a body of chemical knowledge about possible reactions—their inputs, their outputs, and perhaps information about their yields and about the conditions they require. We incorporate this knowledge in productions that allow the following kind of monologue to be carried out. I want to synthesize this molecule; I have as my starting point these reagents. What is the difference between the molecule and the reagents? (There will usually be many such differences.) In the list of reactions (represented by productions), is there any that will take a subset of the reagents and produce a substance that is a little closer to my target molecule? Let us try that reaction. Now I have produced a new molecule and am a little closer to my goal. I will just continue this process recursively until I find a reaction path that leads from reagents to desired molecule.

If you examine this process a bit, you will see that it is close to what in human affairs we call "means/ends analysis" (7, 8). You have a goal. You have some potential means for reaching it. You analyze the relationship between your goal and what you have available. You find a difference between them. From your knowledge stored in memory, you find some kind of operator that will (or may) reduce the difference. You apply the operator, producing a new situation. You play the same game over again. Means/ends analysis provides a nearly complete specification for the organization of a computer program capable of engaging in purposeful search. The design calls for one additional essential element, a control structure to decide at each step what the system will try next. The systems that now do this kind of chemical synthesis, at a quite sophisticated level, illustrate a number of the broad principles that were discovered in the first decade of research on artificial intelligence.

HEURISTIC SEARCH

One of these broad principles is heuristic search—that is, highly selective search through immense spaces of possibilities where even the computing power and speed of the modern computer does not allow it to search everywhere or nearly everywhere.

It has been estimated that there are 10^{120} different possible games of chess. Whether the correct number be 10^{120} or 10^{100} or even only 10^{60}, even the kinds of computers that are promised for the 1990s will not search spaces of this size exhaustively. Search has to be highly selective, and we have to borrow heavily from the kinds of principles of selectivity that humans use in their search. All of the artificial intelligence programs that I know of that are at all successful in exhibiting intelligence in some task make heavy use of the principle of selective heuristic search.

For a serial system, human or computer, to search through a branching tree of possibilities, some rule must be imposed to determine where to search next. From a programming standpoint, the simplest rules are *breadth-first search* and *depth-first search* (1). In breadth-first search, all branches are explored one step ahead, then all twigs of those branches, and so on. In depth-first search, a branch is explored until the answer is found or the continuation seems unpromising. Then the search backs up to the nearest unexplored branch and continues again in depth. Wide experience shows that neither of these procedures competes effectively with *best-first search*. In best-first search, each branch that has not yet been explored is assigned a value, an estimate of the likelihood that it lies on an easy path to the goal. At each step, the search is continued from the unexplored tip that has the highest value. In this scheme, search can shift flexibly from one part to another of the search tree as the evaluations are altered by new information derived from the search.

These are not new principles. What is new is the demonstration that with the use of just these principles plus the basic capabilities of the computer one can do sophisticated things like chemical synthesis and medical diagnosis at a professional level.

LESSONS FROM CHESS

The game of chess has become a sort of *Drosophila* of artificial intelligence, a standard task domain in which experiments can be conducted and knowledge accumulated (9–11). Chess is a nice task for this purpose because, like real life, it does not have smooth mathematical structure. The knights make queer, jagged moves. The edges of the board create complex boundary conditions. Furthermore, an elaborate rating system exists for measuring the skill of human players and, hence, also for calibrating chess-playing programs. For this and other reasons, it is an excellent domain in which to test artificial intelligence techniques.

At present the best computer programs have expert or weak master competence. In rapid play (at 5 or 10 sec per move) they can defeat strong masters fairly often. But under tournament rules, when people have more time to think, they cannot; the comparative advantage of the computer dissipates under these circumstances. The strength of chess programs is improving steadily as a function both of improvements in the programs and increase in speed of the computers on which they are implemented. Within a few years they will be ready for grandmaster competition.

The research on chess programs, which has been going on for more than 20 years, illustrates the tradeoff between speed and cunning. The human being is heavily dependent on cunning—chess knowledge and heuristics that compensate for his inability to search large trees. The computer gains important advantages by making its wheels spin fast, although the strong programs do not rely on brute force alone but incorporate extensive knowledge that enables them to search selectively. They look at hundreds of thousands of alternative positions before making a move—not at 100^{100} or even 10^7.

We have good empirical evidence from the psychological laboratory that, when a chess grand master looks at a complicated middle-game position and ponders for perhaps 15 min before taking a move, he probably does not consider more than 100 branches on the tree of possible moves and countermoves. As a matter of fact, the evidence shows that he does not consider any more branches than does a duffer or a class A player. The only statistic that discriminates the grand master from weaker players is that he examines the relevant branches of the search tree while they commonly look at irrelevant branches and make the wrong move.

SIMULATION OF HUMAN THINKING

The aspect of non-numerical computing in which I am particularly interested is the use of the computer to simulate human thinking. The aim is to find out more about ourselves, about how we think, and, as possible practical outcomes, to help us to think better, to improve teaching and learning processes in our society, and to improve decision-making processes in business, governmental, and educational organizations.

Study of the thought processes used in solving college-level physics problems is proving to be a productive direction of research (12). A physics professor (assumed to be expert) is asked to solve a sophomore physics problem; a sophomore who is currently taking the physics course is also set to solving it. Then, computer programs are built to simulate the thought processes of the expert and of the novice. Comparison of the programs reveals the basic differences in the ways in which they approached the problems and in the pre-stored knowledge and skills they applied to them. The final step in the research, and this is still prospective, will be to build some adaptive production systems that will gradually evolve from novice toward expert status. By an adaptive production system I mean a program that has capabilities for modifying itself, including capabilities for constructing new productions and ingesting them.

What does a program need in order to solve physics problems even at the novice level? First, it has to have a language-processing component (13). Although we are far from being able to handle language in its full generality, there are programs today that, within limits, will handle the language of the problems at the end of the chapter in a physics textbook. They can parse the kinds of sentences that are found there, and they can be provided with appropriate vocabulary. So, as a first requirement, the physics program has to have a considerable capability for natural language processing.

Second, there has to be a semantic component (14). We know that when a good physicist solves a problem he does not just read the English and translate it directly into algebra. We know that he goes through an intermediate stage in which he creates what we would call a physical representation of the problem. This may take the form of a sketch on a piece of paper or a mental picture of the problem. A considerable part of the research on problem-solving in physics is directed at building, in computer memories, symbolic structures (schemata) that have the kinds of semantic information that is held in the physicist's mental picture. Of course, the problem-solving program also needs a mathematical component capable of manipulating algebraic expressions to set up and to solve the equations of the problem.

Space does not permit me to describe here how an adaptive production system can be superimposed on a problem-solving system of the kind I have just described, in order to enable it to learn and to improve its performance. Several such systems, for simple subdomains within physics, algebra, and geometry have already been constructed (15–17). One idea that has been quite successful has been to provide such a program with worked-out problem examples. Using means/ends analysis, the adaptive production system is able to determine what action was performed at each step of the example and what change was accomplished. This information is then converted into a new production, $C \rightarrow A$, in which the conditions C are inferred from changes produced by the action, and the action itself becomes A.

CONCLUSION

I have been able to mention only a tiny sample of the advances in artificial intelligence, cognitive simulation, and non-numerical computing in general that are going on today. At this time there are programs in practical use that analyze mass spectrograms automatically and identify the molecules that produced the spectrogram. Several programs have been developed for medical diagnosis which, although not yet used on a routine basis, have already reached a good clinical level of competence in internal medicine (18) and in microbial diseases (19). Programs now under development are capable of discovering lawful patterns in bodies of empirical data. The examples are almost endless.

I have presented here a personal view of a field that is burgeoning very rapidly, especially as application areas begin to branch off. There are specialists now, for example, who are spending all of their time on medical diagnosis programs; other specialists devote their time to mass spectrography programs, and so on.

In the kinds of non-numerical computation I have been illustrating, the symbols that the computer manipulates are rarely interpretable as numbers. They are more often interpretable as English words and phrases or as abstract representations that are used in many fields in problem solving. Non-numerical computation—artificial intelligence and cognitive simulation—may turn out to be the most significant of all the consequences of the invention of the computer, not merely or perhaps not even because of the additional computing power it gives us but because it gives us deep insights into the nature of intelligence and hence into the operation of the human mind. With the aid of this new device, the computer, used in the way that I have been describing, we are learning to obey that ancient injunction that was said to have been inscribed at Delphi, "Know thyself."

1. Newell, A. & Simon, H. A. (1972) *Human Problem Solving* (Prentice-Hall, Englewood Cliffs, NJ).
2. Oliver, B. M. (1980) *Proc. Natl. Acad. Sci. USA* **77**, 6260–6261.
3. Newell, A. & Simon, H. A. (1976) *Communications of the ACM* **19**, 113–126.
4. Schwartz, J. T. (1980) *Proc. Natl. Acad. Sci. USA* **77**, 6262–6263.
5. Knuth, D. E. (1968) *The Art of Computer Programming* (Addison-Wesley, Reading, MA), Vol. 1, pp. 228–464.
6. Feigenbaum, E. A. (1963) in *Computers and Thoughts*, eds. Feigenbaum, E. A. & Feldman, J. (McGraw-Hill, New York), pp. 297–309.
7. Ernst, G. W. & Newell, A. (1969) *GPS: A Case Study in Generality and Problem Solving* (Academic, New York).
8. Corey, E. J. & Wipke, W. T. (1969) *Science* **166**, 178–192.
9. de Groot, A. (1978) *Thought and Choice in Chess* (Mouton, New York).
10. Chase, W. G. & Simon, H. A. (1973) in *Visual Information Processing,* ed. Chase, W. G. (Academic, New York), pp. 215–282.
11. Simon, H. A. & Chase, W. G. (1973) *Am. Sci.* **61**, 394–403.
12. Larkin, J., McDermott, J., Simon, D. P. & Simon, H. A. (1980) *Science* **208**, 1335–1342.
13. Winograd, T. (1972) *Understanding Natural Language* (Academic, New York).
14. Novak, G. S. (1977) *Proceedings Fifth International Joint Conference on Artificial Intelligence* (IJCAI, Boston), pp. 286–291.
15. Waterman, D. A. (1970) *Artif. Intell.* **1**, 121–170.
16. Neves, D. (1978) *Proceedings Second Conference of the Canadian Society for Computational Studies of Intelligence* (Univ. Toronto, Toronto, Ont.), pp. 191–195.
17. Anzai, Y. & Simon, H. A. (1979) *Psychol. Rev.* **86**, 124–140.
18. Pople, H. (1977) *Proceedings Fifth International Joint Conference on Artificial Intelligence* (IJCAI, Boston), pp. 1030–1037.
19. Shortliffe, E. H. (1976) *MYCIN: Computer-based Medical Consultation* (Am. Elsevier, New York).