Undergraduate Projects in the Application of Artificial Intelligence to Chemistry. I. Background

Hugh Cartwright

Physical and Theoretical Chemistry Laboratory, Oxford University, South Parks Road, Oxford OX1 3QZ, England, hugh.cartwright@chem.ox.ac.uk

Abstract: The potential of Artificial Intelligence (AI) in the development of intelligent machines is widely recognized. It is less widely appreciated that the methods which computer scientists use in their work on AI are also applicable to the solution of numerous problems in science. In many cases, AI methods are preferable to more conventional approaches, being superior in terms of time, quality of solution, or both. Most AI tools are comparatively simple to understand, despite their power, and computer programs to implement them can be written by anyone with average programming skills. This series of papers will demonstrate how AI methods are of value in science, why they work, and how they can be introduced into the syllabus as undergraduate research projects; suggestions of projects, illustrative programs and Java source code will be provided. This paper introduces the topic of AI and explains some of the ways in which an AI program differs from a conventional program.

Introduction

The evolution of science depends on the abilities and inspiration of scientists, but it is also crucially dependent upon the development of new experimental, theoretical, and computational tools. Advances in computational chemistry over the last three decades have been particularly marked, as growth in computer power has allowed scientists to undertake calculations that previously were too time-consuming.

A notable effect of this growth in processor speed has been to open up entirely new fields of analysis. Some of the most exciting new methods for tackling scientific problems lie within the field of Artificial Intelligence (AI). AI was once perceived to have as its limited (but nevertheless very challenging) goal the development of intelligent machines and agents. That goal remains, but gradually it has become apparent that the scope of AI methods is far broader than the mere production of clever robots [1–6]. AI is now recognized as constituting a powerful set of tools in its own right, capable of solving problems in such diverse areas as finance, pollution control, reactor simulation, food quality assessment, bus routing, and chemistry, to give just a few examples. As computer speed increases, AI tools very rapidly become more attractive, and it is not unreasonable to believe that, within a decade, software relying upon the principles of AI will be brought to bear upon many of the most intractable problems in science and will be of fundamental importance in the interpretation of scientific data.

In view of this, chemistry undergraduates whose interests extend to computing should have the opportunity to learn about AI and appreciate the impact it will have across science. (Indeed, one might argue that *all* scientists should have at least a passing acquaintance with the subject, such is its likely future effect on science.) To fully understand how AI can be used to interpret scientific data, hands-on experience is extremely valuable. The chemistry syllabus is very full, however, and although some chemistry departments already include an element of AI in their computing instruction, in many more cases undergraduate chemistry projects provide the most appropriate route by which students can gain substantive experience in the field. It is the aim of these papers to introduce some of the AI methods that have the greatest potential within chemistry, to outline their use, and to make practical suggestions of projects through which AI methods can be introduced into undergraduate research projects.

What is Artificial Intelligence?

It may seem odd that something as fundamental as the definition of Artificial Intelligence, a field in which tens of thousands of people work in industry and in universities, is not universally agreed, but such is the case. Some in the field regard the role of AI as the development of machines that, in all their intellectual behavior and properties, can rival humans; others argue that any computer programs that are differentiated from the deterministic form of programming, with which most scientists are familiar, by an ability to reason are examples of AI.

In these papers we shall adopt a broad definition of an AI computer program as being one that can *learn*. As we shall see in the examples developed in later papers, the manner in which a computer program learns is very varied. Some programs (“Expert Systems” [7], for example) learn by being spoon-fed information by a human expert, and in this way build a database of knowledge and rules that can eventually be used to provide advice. Other programs (such as “neural networks” [8], “genetic algorithms” [9–11] or “ant systems” [12, 13]) autonomously investigate their environment, taking note of its responses to their probing and so learn about the world in which they operate; still others (“self-ordering maps” [14]) quite remarkably can extract useful information from a database without any feedback at all to tell them whether their conclusions are correct. This wide range of behavior may seem confusing, but it is actually an asset, because the different types of program are suited to different types of scientific problems; consequently, as we shall see in subsequent papers,

```
PRINT " Give me values for a, b and c please."
INPUT A
INPUT B
INPUT C
IF (B*B >= 4*A*C) THEN
{
     X1= (-B+(SQR(B*B-4*A*C)))/(2*A)
     X2= (-B-(SQR(B*B-4*A*C)))/(2*A)
     PRINT " The roots are " X1 ; " and "; X2
}
ELSE
{
     PRINT " The roots are complex."
}
END
```

Figure 1. A fragment of a BASIC program for finding the roots of a quadratic equation.

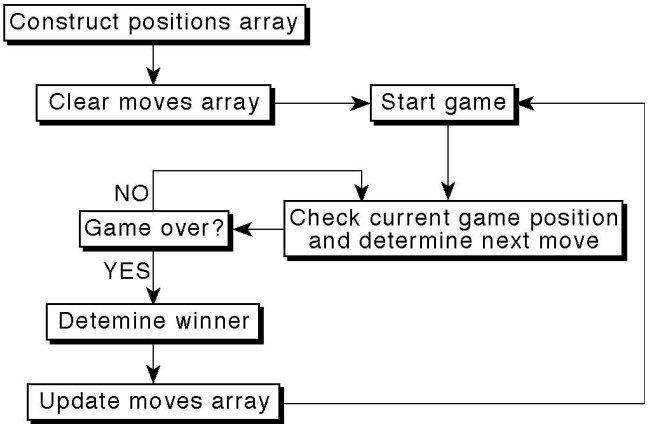

Figure 2. Flow chart for a simple tic-tac-toe-playing program.

a huge variety of scientific problems are amenable to attack using AI programs.

How Can a Computer Program Learn?

A computer must be told precisely what it is to do before it can accomplish anything; these instructions are encapsulated as the computer program. But, if a program holds all the information that is necessary for the computer to function, and if that program is fully defined in advance by the programmer, what freedom exists for the computer to learn? This is an important question, which we will address using a simple example.

The snippet of code given in Figure 1 constitutes a program whose purpose is to determine the (real) roots of the quadratic equation $ax^2 + bx + c = 0$. It can do this for an infinite variety of combinations of the coefficients a, b and c, but every invocation of the program is fresh, and the program learns nothing from experience, no matter how frequently it is run. It is no more or less proficient at finding the roots of quadratic equations on its hundredth use than it was on its first. This must be the case, because there is nothing further for the program to learn; even on its first use it already has all the information it needs to perform its simple task perfectly.

AI programs are quite different from the simple program shown above—they *do* learn from experience. By implication, therefore, they must start life with only limited knowledge (or none at all) about the problem they are to tackle and must gradually construct an understanding of the problem. This can be done in various ways, for example, by interacting with an expert, with the user, or with the environment. To illustrate, we present an example that, though it is trivial enough that it falls only on the margins of AI, does provide some insight into how learning can occur.

Computer Learning—A Simple Example

Figure 2 shows a schematic for a computer program that can learn to play tic-tac-toe (noughts-and-crosses) at a modest level. This program is tackling a nonscientific task, but the principles extrapolate readily to practical scientific problems.

Tic-tac-toe is a pencil-and-paper game with which almost every child (and adult) is familiar. Players take it in turns to add O's or X's to a 9-box grid with the aim of being the first to complete a line of zeros or crosses along a horizontal, diagonal, or vertical row. If a computer is to play this game it needs to be able to accomplish several tasks:

i) recognize the state of the board at any stage of the game;

ii) determine a legal move to play;

iii) recognize when the game is over.

If the computer can perform these steps and no more, however, it will never be much of a player. To be able to win consistently it needs, in step (ii), to make not just legal `moves`, but to make good `moves`. There are two quite different ways in which it might manage this. The computer could store every conceivable position that might arise in the game, together with the "best" move to make in each situation, as defined in advance by the programmer; it could then proceed by always following this predetermined "best" route. This strategy might succeed, but it relies upon the programmer knowing in advance the best possible move for every game position. An alternative strategy is for the computer itself to learn what the best move is for each game position. Let us see how this can be done.

We first consider how the computer might deal with tasks i–iii.

i) Recognize the state of the board. A simple way in which the computer could recognize the state of the game is for it to

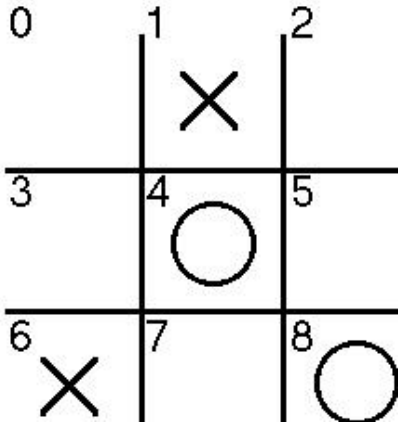

Figure 3. An early stage in a typical game.

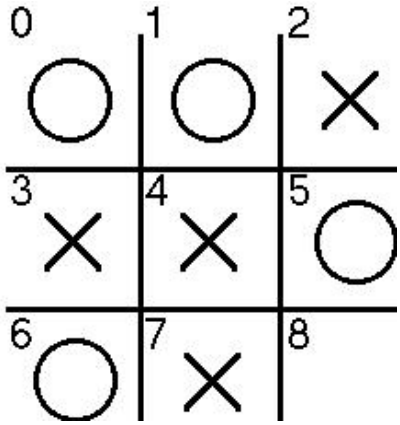

Figure 4. A game about to end in a draw.

have access to an array (we shall call it the "`grid_state`" array), which contains an entry corresponding to every possible position that may arise in the game. Any method might be used to store these positions. In this example, we will number the boxes consecutively in the grid, and then in `grid_state` denote a box that contains an O by a zero, a box that contains an X by a one, and an empty box with a nine. According to this encoding, if the grid boxes are numbered in the fashion shown in Figures 3 and 4, the game position shown in Figure 3 would be stored as 919909109, the game position shown in Figure 4 would be encoded as 001110019, and the starting position, 999999999.

ii) Determine a suitable move to make. Once the computer has determined the current state of the game, it must chose a move. A "dumb" program, with no understanding whatsoever, would make any move provided that it was legal, picking an empty square at random. A "pretrained" program or an "intelligent" program, however, would have to select a move according to some algorithm. For this purpose, we will assume that the computer can consult a second array which we shall call "`moves.`" For any selected state of the grid this contains the number of the box that the computer should now chose as its move in order to maximize its chance of winning the game. In the pretrained program, this move is set in advance; in the intelligent program it will be determined by experience.

iii) Recognize when the game is over. It is a simple matter for the computer to determine after any move whether the game is complete, by inspecting the `grid_state` array to determine whether either player has completed a row, or the board is full.

The game then proceeds as follows:

- The board is cleared, and the computer checks the `grid_state` array to find the current position. It then reads the moves array to find out where to place its cross.
- The player responds to the computer's move by placing an O in an unoccupied box.
- The computer determines the new state of the grid from the `grid_state` array, checks that the game is not over, and again consults the corresponding entry in the moves array to find out what its next move should be.

These steps are repeated until either one player wins, or until no blank boxes remain.

Once the game is over, both dumb and pretrained programs are immediately ready for their next game, because the way they play is completely unaffected by experience. An intelligent program, however, must learn from the game in which it has just been involved. All the computer's knowledge is encapsulated in the moves array—this array is in essence the "memory" of the computer program—so it is this array which must be modified if the computer is to learn. We can accomplish this if we include a second entry in the moves array, which is a weight; this weight is a measure of how likely it is that, having made the move to which the weight corresponds, the computer will win the game.

When the program is run for the very first time, every entry in the moves array is set at random—the computer knows nothing, so any arbitrary move is as good as any other. Every weight is set arbitrarily to five.

At the end of each game, the entries in the moves array are adjusted to reflect the successes and failures of the computer. If the computer has won the game, the plays it made were productive. It needs to be encouraged to make the same moves in the future when it faces the same situations, so the weights of all `moves` it made during the game are increased by (the arbitrary value of) one. If the computer has lost the game, its `moves` were poor. Accordingly, the weights of all `moves` it made during the game are decreased (again by one). For some `moves`, this decrement may reduce the associated weight to less than zero; if this happens a new legal move is chosen at random, placed in the moves array, and given a weight of zero.

How does the program learn as a result of this adjustment of weights? Initially, the computer knows nothing, because all its `moves` were chosen at random. Gradually, however, when by chance it makes the right `moves` and as a consequence wins a game, the `moves` that lead to that success are rewarded by an increase in their weights, so they will be selected again when the computer encounters an identical board position in future games. Unproductive moves, which have lead to a loss, are penalized. Soon the weights of such moves fall below zero, at which point they are replaced by alternative moves selected at random.

Over time, the weights of productive moves are reinforced, whereas those moves that lead to the computer losing the game are replaced by other randomly-selected `moves`, which, if valuable, are rewarded. In this fashion, the computer discards poor moves and reinforces sound ones—it is learning how to win.

This is a particularly simple example, but it does illustrate how computers can develop knowledge. Two aspects of this learning are particularly worthy of note:

i) The computer is learning responses, not rules. For example, the computer cannot derive the rule

> If my opponent has two zeros in a row, I must position my cross so as to block the row off, otherwise, I shall lose the game.

Although it cannot discover this *rule*, it can learn the correct *action to take to implement the rule*. That is, it learns to make the right move in a particular position, but does not understand why it should do it. (Some forms of neural net are an exception to this; much attention is now being paid to forms of neural nets that can derive rules from data. These form a method of great potential in chemistry, and will be discussed in a later paper).

ii) We cannot predict how the computer will learn, because the evolution of the weights, which form the program's memory, depends upon the moves chosen by its opponent and also by the randomly chosen contents of the moves array when the program first starts operation. This random element is a characteristic of many types of AI programs, and indeed it is crucial to the operation of some [10].

What Kinds of Problems Can AI Programs Tackle?

It will be obvious that few, if any, problems in chemistry resemble tic-tac-toe in form, nor are they generally so simple, so the relevance of this game to real scientific problems may not be obvious. It is not the game described above which is of importance, however, but the manner in which it illustrates that, relying not on explicit instruction but experience, a computer can learn. Real AI programs use more sophisticated methods to develop knowledge, and there is a role for such programs in chemistry, tackling problems for which the route to a solution is not self-evident. Later papers will discuss in more detail the kinds of tasks that AI programs can successfully undertake, but it would be appropriate here to mention a few examples of roles that AI programs may play.

The earliest AI programs in chemistry—indeed some of the earliest AI applications to any subject—were forms of expert system. In such programs, the knowledge of an expert in a field is encoded in such a way that the computer can reason in a manner similar to that which an expert might employ. It follows, therefore, that the program should reach the same conclusions as an expert would when presented with the same data. Early examples of expert systems involved the interpretation of mass spectral fragmentation patterns and the determination of the appropriate materials to use for protective gloves when handling dangerous chemicals.

More recently, neural networks have been used to analyze infrared spectra, genetic algorithms to determine the efficient operation of a chemical flowshop, and pheromone trail (ant system) algorithms to optimize synthetic routes.

These applications share an important characteristic: no "conventional" (deterministic) algorithm exists that can solve them exactly. This is almost invariably a characteristic of AI applications; where deterministic methods exist, they are often faster than AI methods; where they are absent, or too complicated to use, AI methods may provide a route to high-quality answers in a reasonable time.

Are AI Programs Difficult to Implement?

Full-scale AI applications may appear complicated, but their principles are generally straightforward. Indeed, it is remarkable that, even though the logic underlying genetic algorithms, neural networks or self-ordering maps is simple, the methods themselves are capable of solving complex problems. Later papers will present the principles of these techniques in detail so that novice users, familiar with programming, but not with AI, will be able to code working examples. These papers will outline for each method examples of how they can be used in chemistry, and will show how straightforward it is for undergraduates to prepare meaningful AI applications.

References

1. Ordonez, R.; Zumberge, J.; Spooner, J. T.; Passino, K. M. "Adaptive Fuzzy Control: Experiments and Comparative Advantages" *IEEE-FUZZ* **1997**, *5*(2)**,** 167.
2. Russo, M. "FuGeNeSys—A Fuzzy Genetic Neural System for Fuzzy Modelling" *IEEE-FUZZ* **1998**, *6*(3)**,** 373.
3. Goonatilake, S.; Khebbal, S. *Intelligent Hybrid Systems*; Wiley: Chichester, 1995.
4. Issott, D. The Operation of Resin Processes by Hybrid Genetic Algorithm-Fuzzy Logic Control. Chemistry Part II Thesis, Oxford University, U.K., 1999.
5. Becerra, V.; Roberts, P. D.; Griffiths, G. W. "Novel Developments in Process Optimization Using PredictiveControl" *J. Proc. Cont.* **1998**, *8*(2), 117.
6. Cartwright, H. M. *Applications of Artificial Intelligence in Chemistry*; Oxford University Press: Oxford, U.K., 1993.
7. Zhu, Q.; Stillman, M. J. "Expert Systems and Analytical Chemistry: Recent Progress in the ACexpert Project" *J. Chem. Inf. Comput. Sci.* **1996**, *36*, 497.
8. Porter, A. Neural Networks for Interpretation of Real-time Infrared Spectra of Pollutants in the Workplace. Chemistry Part II Thesis, Oxford University, U.K., 1999.
9. Keun, K. Quantitative Structure-Activity Analysis of N-Substituted Arenes using Genetic Algorithms. Chemistry Part II Thesis, Oxford University, U.K., 1997.
10. *Handbook of Genetic Algorithms*; Davis, L., Ed.; Van Nostrand Reinhold: New York, 1991.
11. Tuson, A. L. The Use of Genetic Algorithms to Optimise Flowshops of Unrestricted Topology. Chemistry Part II Thesis, Oxford University, U.K., 1994.
12. Hopkins, J. A. Pheromone Trail Algorithms. Chemistry Part II Thesis, Oxford University, U.K., 1995.
13. Cartwright, H. M.; Hopkins, J. A. Evolutionary Design of Synthetic Routes in Chemistry. In *Proceedings of the AISB Conference on Evolutionary Computing*, University of Sussex, U.K., 1996.
14. Saunders, J. Investigation of Structure-Biodegradability Relationships Using Self-Organizing Maps. Chemistry Part II Thesis, Oxford University, U.K., 1997.