5

Early Heuristic Programs

5.1 The Logic Theorist and Heuristic Search

Just prior to the Dartmouth workshop, Newell, Shaw, and Simon had programmed a version of LT on a computer at the RAND Corporation called the JOHNNIAC (named in honor of John von Neumann). Later papers[1] described how it proved some of the theorems in symbolic logic that were proved by Russell and Whitehead in Volume I of their classic work, *Principia Mathematica*.[2] LT worked by performing transformations on Russell and Whitehead's five axioms of propositional logic, represented for the computer by "symbol structures," until a structure was produced that corresponded to the theorem to be proved. Because there are so many different transformations that could be performed, finding the appropriate ones for proving the given theorem involves what computer science people call a "search process."

To describe how LT and other symbolic AI programs work, I need to explain first what is meant by a "symbol structure" and what is meant by "transforming" them. In a computer, symbols can be combined in lists, such as (A, 7, Q). Symbols and lists of symbols are the simplest kinds of symbol structures. More complex structures are composed of lists of lists of symbols, such as ((B, 3), (A, 7, Q)), and lists of lists of lists of symbols, and so on. Because such lists of lists can be quite complex, they are called "structures." Computer programs can be written that transform symbol structures into other symbol structures. For example, with a suitable program the structure "(the sum of seven and five)" could be transformed into the structure "(7 + 5)," which could further be transformed into the symbol "12."

Transforming structures of symbols and searching for an appropriate problem-solving sequence of transformations lies at the heart of Newell and Simon's ideas about mechanizing intelligence. In a later paper (the one they gave on the occasion of their receiving the prestigious Turing Award), they summarized the process as follows:[3]

The solutions to problems are represented as symbol structures. A physical symbol system exercises its intelligence in problem solving by search – that is, by generating and progressively modifying symbol structures until it produces a solution structure.

. . .

To state a problem is to designate (1) a test for a class of symbol structures (solutions of the problem), and (2) a generator of symbol structures (potential solutions). To solve a problem is to generate a structure, using (2), that satisfies the test of (1).

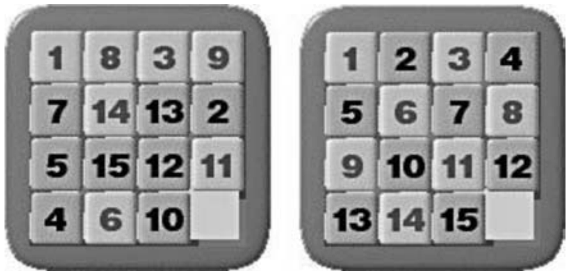

Figure 5.1. Start (left) and goal (right) configurations of a fifteen-puzzle problem.

Understanding in detail how LT itself used symbol structures and their transformations to prove theorems would require some mathematical and logical background. The process is easier to explain by using one of AI's favorite "toy problems" – the "fifteen-puzzle." (See Fig. 5.1.) The fifteen-puzzle is one of several types of sliding-block puzzles. The problem is to transform an array of tiles from an initial configuration into a "goal" configuration by a succession of moves of a tile into an adjacent empty cell.

I'll use a simpler version of the puzzle – one that uses a 3×3 array of eight sliding tiles instead of the 4×4 array. (AI researchers have experimented with programs for solving larger versions of the puzzle also, such as 5×5 and 6×6.)

Suppose we wanted to move the tiles from their configuration on the left to the one on the right as illustrated in Fig. 5.2.

Following the Newell and Simon approach, we must first represent tile positions for the computer by symbol structures that the computer can deal with. I will represent the starting position by the following structure, which is a list of three sublists:

((2, 8, 3), (1, 6, 4), (7, B, 5)).

The first sublist, namely, (2, 8, 3), names the occupants of the first row of the puzzle array, and so on. B stands for the empty cell in the middle of the third row.

In the same fashion, the goal configuration is represented by the following structure:

((1, 2, 3), (8, B, 4), (7, 6, 5)).

Next, we have to show how a computer can transform structures of the kind we have set up in a way that corresponds to the allowed moves of the puzzle. Note that when a tile is moved, it swaps places with the blank cell; that is, the blank cell moves too. The blank cell can either move within its row or can change rows.

Corresponding to these moves of the blank cell, when a tile moves within its row, B swaps places with the number either to its left in its list (if there is one) or to its

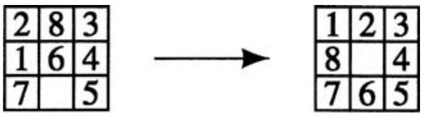

Figure 5.2. The eight-puzzle.

right (if there is one). A computer can easily make either of these transformations. When the blank cell moves up or down, B swaps places with the number in the corresponding position in the list to the left (if there is one) or in the list to the right (if there is one). These transformations can also be made quite easily by a computer program.

Using the Newell and Simon approach, we start with the symbol structure representing the starting configuration of the eight-puzzle and apply allowed transformations until a goal is reached. There are three transformations of the starting symbol structure. These produce the following structures:

((2, 8, 3), (1, 6, 4), (B, 7, 5)),

((2, 8, 3), (1, 6, 4), (7, 5, B)),

and

((2, 8, 3), (1, B, 4), (7, 6, 5)).

None of these represents the goal configuration, so we continue to apply transformations to each of these and so on until a structure representing the goal is reached. We (and the computer) can keep track of the transformations made by arranging them in a treelike structure such as the one shown in Fig. 5.3. (The arrowheads on both ends of the lines representing the transformations indicate that each transformation is reversible.)

This version of the eight-puzzle is relatively simple, so not many transformations have to be tried before the goal is reached. Typically though (especially in larger versions of the puzzle), the computer would be swamped by all of the possible transformations – so much so that it would never generate a goal expression. To constrain what was later called "the combinatorial explosion" of transformations, Newell and Simon suggested using "heuristics" to generate only those transformations guessed as likely to be on the path to a solution.

In one of their papers about LT, they wrote "A process that *may* solve a problem, but offers no guarantees of doing so, is called a *heuristic* for that problem." Rather than blindly striking out in all directions in a search for a proof, LT used search guided by heuristics, or "heuristic search." Usually, as was the case with LT, there is no guarantee that heuristic search will be successful, but when it is successful (and that is quite often) it eliminates much otherwise fruitless search effort.

The search for a solution to an eight-puzzle problem involves growing the tree of symbol structures by applying transformations to the "leaves" of the tree and thus extending it. To limit the growth of the tree, we should use heuristics to apply transformations only to those leaves thought to be on the way to a solution. One such heuristic might be to apply a transformation to that leaf with the smallest number of tiles out of position compared to the goal configuration. Because sliding tile problems have been thoroughly studied, there are a number of heuristics that have proved useful – ones much better than the simple number-of-tiles-out-of-position one I have just suggested.

Using heuristics keyed to the problem being solved became a major theme in artificial intelligence, giving rise to what is called "heuristic programming." Perhaps

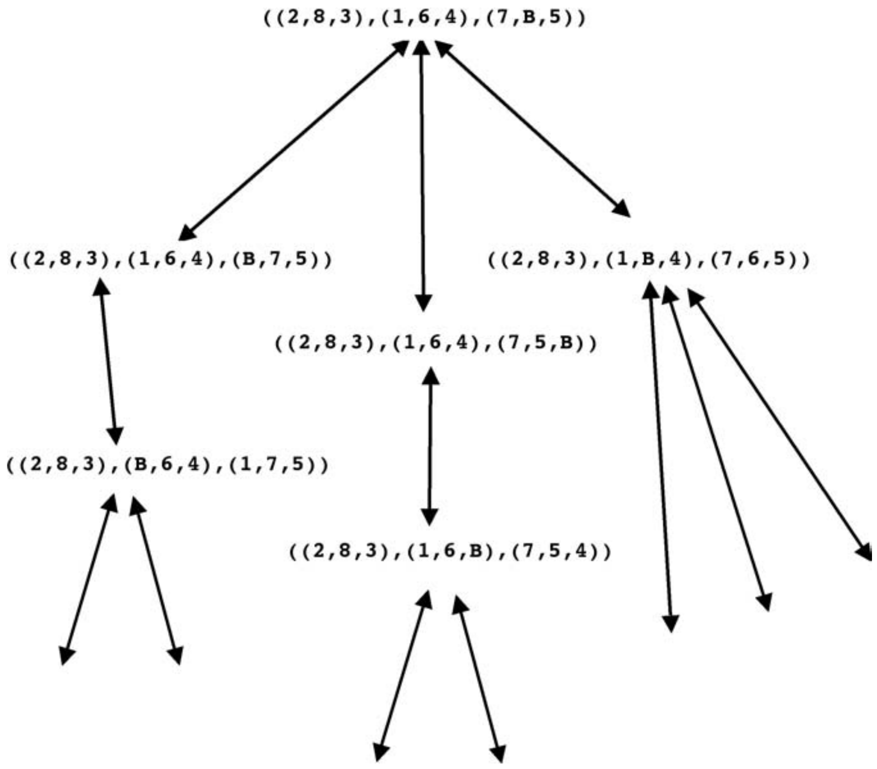
((2,8,3),(1,6,4),(7,B,5))
((2,8,3),(1,6,4),(B,7,5))
((2,8,3),(1,B,4),(7,6,5))
((2,8,3),(1,6,4),(7,5,B))
((2,8,3),(B,6,4),(1,7,5))
((2,8,3),(1,6,B),(7,5,4))

Figure 5.3. A search tree.

the idea of heuristic search was already "in the air" around the time of the Dartmouth workshop. It was implicit in earlier work by Claude Shannon. In March 1950, Shannon, an avid chess player, published a paper proposing ideas for programming a computer to play chess.[4] In his paper, Shannon distinguished between what he called "type A" and "type B" strategies. Type A strategies examine every possible combination of moves, whereas type B strategies use specialized knowledge of chess to focus on lines of play thought to be the most productive. The type B strategies depended on what Newell and Simon later called heuristics. And Minsky is quoted as saying "…I had already considered the idea of heuristic search obvious and natural, so that the Logic Theorist was not impressive to me."[5]

It was recognized quite early in AI that the way a problem is set up, its "representation," is critical to its solution. One example of how a representation affects problem solving is due to John McCarthy and is called the "mutilated checkerboard" problem.[6] Here's the problem: "Two diagonally opposite corner squares are removed from a checkerboard. Is it possible to cover the remaining squares with dominoes?" (A domino is a rectangular tile that covers two adjacent squares.) A naive way of searching for a solution would be to try to place dominoes in all possible ways over the checkerboard. But, if one uses the information that a checkerboard consists of 32 squares of one color and 32 of another color, and that the opposite corner squares are of the same color, then one realizes that the mutilated board consists of 30 squares of one color and 32 of another. Because a domino covers two squares of opposite colors, there is no way that a set of them can cover the remaining colors. McCarthy was interested in whether or not people could come up with "creative" ways to formulate the puzzle so that it could be solved by computers using methods based on logical deduction.

Another classic puzzle that has been used to study the effects of different representations is the "missionary and cannibals" problem: Three cannibals and three missionaries must cross a river. Their boat can only hold two people. If the cannibals outnumber the missionaries, on either side of the river, the missionaries on that side perish. Each missionary and each cannibal can row the boat. How can all six get across the river safely? Most people have no trouble formulating this puzzle as a search problem, and the solution is relatively easy. But it does require making one rather nonintuitive step. The computer scientist and AI researcher Saul Amarel (1928–2002) wrote a much-referenced paper analyzing this puzzle and various extended versions of it in which there can be various numbers of missionaries and cannibals.[7] (The extended versions don't appear to be so easy.) After moving from one representation to another, Amarel finally developed a representation for a generalized version of the problem whose solution required virtually no search. AI researchers are still studying how best to represent problems and, most importantly, how to get AI systems to come up with their own representations.

5.2 Proving Theorems in Geometry

Nathan Rochester returned to IBM after the Dartmouth workshop excited about discussions he had had with Marvin Minsky about Minsky's ideas for a possible computer program for proving theorems in geometry. He described these ideas to a new IBM employee, Herb Gelernter (1929–). Gelernter soon began a research project to develop a geometry-theorem-proving machine. He presented a paper on the first version of his program at a conference in Paris in June 1959,[8] acknowledging that

> [t]he research project itself is a consequence of the Dartmouth Summer Research Project on Artificial Intelligence held in 1956, during which M. L. Minsky pointed out the potential utility of the diagram to a geometry theorem-proving machine.

Gelernter's program exploited two important ideas. One was the explicit use of subgoals (sometimes called "reasoning backward" or "divide and conquer"), and the other was the use of a diagram to close off futile search paths.

The strategy taught in high school for proving a theorem in geometry involves finding some subsidiary geometric facts from which, if true, the theorem would follow immediately. For example, to prove that two angles are equal, it suffices to show that they are corresponding angles of two "congruent" triangles. (A triangle is congruent to another if it can be translated and rotated, possibly even flipped over, in such a way that it matches the other exactly.) So now, the original problem is transformed into the problem of showing that two triangles are congruent. One way (among others) to show that two triangles are congruent is to show that two

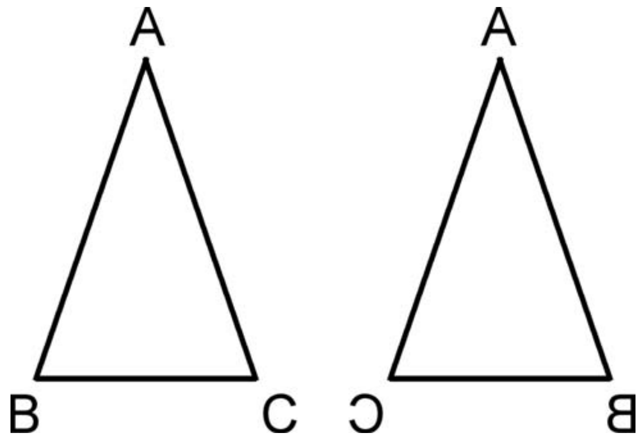

Figure 5.4. A triangle with two equal sides (left) and its flipped-over version (right).

corresponding sides and the enclosed angle of the two triangles all have the same sizes. This backward reasoning process ends when what remains to be shown is among the premises of the theorem.

Readers familiar with geometry will be able to follow the illustrative example shown in Fig. 5.4. There, on the left-hand side, we are given triangle ABC with side AB equal to side AC and must prove that angle ABC is equal to angle ACB. The triangle on the right side is a flipped-over version of triangle ABC.

Here is how the proof goes: If we could prove that triangle ABC is congruent to triangle AƆᗺ, then the theorem would follow because the two angles are corresponding angles of the two triangles. These two triangles can be proved congruent if we could establish that side AB (of triangle ABC) is equal to side AƆ (of triangle AƆᗺ) and that side AC (of triangle ABC) is equal to side Aᗺ (of triangle AƆᗺ) and that angle A (of triangle ABC) is equal to angle A (of triangle AƆᗺ). But the premises state that side AB is equal to side AC, and these lengths don't change in the flipped-over triangle. Similarly, angle A is equal to its flipped-over version – so we have our proof.

Before continuing my description of Gelernter's program, a short historical digression is in order. The geometry theorem just proved is famous – being the fifth proposition in Book I of Euclid's *Elements*. Because Euclid's proof of the proposition was a difficult problem for beginners it became known as the *pons asinorum* or "fools bridge." The proof given here is simpler than Euclid's – a version of it was given by Pappus of Alexandria (circa 290–350 CE).

Minsky's "hand simulation" of a program for proving theorems in geometry, discussed at Dartmouth, came up with this very proof (omitting what I think is the helpful step of flipping the triangle over). Minsky wrote[9]

> In 1956 I wrote two memos about a hand-simulated program for proving theorems in geometry. In the first memo, the procedure found the simple proof that if a triangle has two equal sides then the corresponding angles are equal. It did this by noticing that triangle ABC was congruent to triangle CBA because of "side-angle-side." What was interesting is that this was found after a very short search – because, after all, there weren't many things to do. You might say the program was too stupid to do what a person might do, that is, think, "Oh, those are both the same triangle. Surely no good could come from giving it two different names." (The

program has a collection of heuristic methods for proving Euclid-Like theorems, and one was that "if you want to prove two angles are equal, show that they're corresponding parts of congruent triangles." Then it also had several ways to demonstrate congruence. There wasn't much more in that first simulation.) But I can't find that memo anywhere.

As Minsky said, this is a very easy problem for a computer. Gelernter's program proved much more difficult theorems, and for these his use of a diagram was essential. The program did not literally draw and look at a diagram. Instead, as Gelernter wrote,

[The program is] supplied with the diagram in the form of a list of possible coordinates for the points named in the theorem. This point list is accompanied by another list specifying the points joined by segments. Coordinates are chosen to reflect the greatest possible generality in the figures.

So, for example, the points named in the problem about proving two angles equal are the vertices of the triangle ABC, namely, points A and B and C. Coordinates for each of these points are chosen, and care is taken to make sure that these coordinates do not happen to satisfy any special unnamed properties.

Gelernter's program worked by setting up subgoals and subsubgoals such as those I used in the example just given. It then searched for a chain of these ending in subgoals that could be established directly from the premises. Before any subgoal was selected by the program to be worked on however, it was first tested to see whether it held in the diagram. If it did hold, it might possibly be provable and could therefore be considered as a possible route to a proof. But, if it did not hold in the diagram, it could not possibly be true. Thus, it could be eliminated from further consideration, thereby "pruning" the search tree and saving what would certainly be fruitless effort. Later work in AI would also exploit "semantic" information of this sort.

We can see similarities between the strategies used in the geometry program and those used by humans when we solve problems. It is common for us to work backward – transforming a hard problem into subproblems and those into subsubproblems and so on until finally the problems are trivial. When a subproblem has many parts, we know that we must solve all of them. We also recognize when a proposed subproblem is patently impossible and thus can reject it. The next program I describe was based explicitly on what its authors thought were human problem-solving strategies.

5.3 The General Problem Solver

At the same 1959 Paris conference where Gelernter presented his program, Allen Newell, J. C. Shaw, and Herb Simon gave a paper describing their recent work on mechanizing problem solving.[10] Their program, which they called the "General Problem Solver (GPS)," was an embodiment of their ideas about how humans solve problems. Indeed, they claimed that the program itself was a theory of human problem-solving behavior. Newell and Simon were among those who were just as interested (perhaps even more interested) in explaining the intelligent behavior of humans as they were in building intelligent machines. They wrote "It is often argued that a careful line must be drawn between the attempt to *accomplish* with machines the same tasks that humans perform, and the attempt to *simulate* the processes humans actually use to accomplish these tasks. . . . GPS maximally confuses the two approaches – with mutual benefit."[11]

GPS was an outgrowth of their earlier work on the Logic Theorist in that it was based on manipulating symbol structures (which they believed humans did also). But GPS had an important additional mechanism among its symbol-manipulating strategies. Like Gelernter's geometry program, GPS transformed problems into subproblems, and so on. GPS's innovation was to compute a "difference" between a problem to be solved (represented as a symbol structure) and what was already known or given (also represented as a symbol structure). The program then attempted to reduce this difference by applying some symbol-manipulating "operator" (known to be relevant to this difference) to the initial symbol structure. Newell and Simon called this strategy "means–ends analysis." (Note the similarity to feedback control systems, which continuously attempt to reduce the difference between a current setting and a desired setting.) To do so, it would have to show that the operator's applicability condition was satisfied – a subproblem. The program then started up another version of itself to work on this subproblem, looking for a difference and so on.

For example, suppose the goal is to have Sammy at school when Sammy is known to be at home.[12] GPS computes a "difference," namely, Sammy is in the wrong place, and it finds an operator relevant to reducing this difference, namely, driving Sammy to school. To drive Sammy to school requires that the car be in working order. To make the problem interesting, we'll suppose that the car's battery is dead, so GPS can't apply the drive-car operator because that operator requires a working battery. Getting a working battery is a subproblem to which GPS can apply a version of itself. This "lower" version of GPS computes a difference, namely, the need for a working battery, and it finds an operator, namely, calling a mechanic to come and install a new battery. To call a mechanic requires having a phone number (and let us suppose we have it), so GPS applies the call-mechanic operator, resulting in the mechanic coming to install a new battery. The lower version of GPS has successfully solved its problem, so the superordinate GPS can now resume – noting that the condition for drive-car, namely, having a working battery, is satisfied. So GPS applies this operator, Sammy gets to school, and the original problem is solved. (This example illustrates the general workings of GPS. A real one using actual symbol structures, differences, and operators with their conditions and so on would be cumbersome but not more revealing.)

When GPS works on subproblems by starting up a new version of itself, it uses a very important idea in computer science (and in mathematics) called "recursion." You might be familiar with the idea that computer programmers organize complex programs hierarchically. That is, main programs fire up subprograms, which might fire-up subsubprograms, and so on. When a main program "calls" a subprogram, the main program suspends itself until the subprogram completes what it is supposed to do (possibly handing back data to the main program), and then the main program resumes work. In AI (and in other applications also), it is common to have a main program call a version of itself – taking care that the new version works on a simpler problem so as to avoid endless repetition and "looping." Having a program call itself is called "recursion."

Do people use subprograms and recursion in their own thinking? Quite possibly, but their ability to recall how to resume what some higher level thought process was doing when that process starts up a chain of lower level processes is certainly limited. I don't believe that GPS attempted to mimic this limitation of human thinking.

Newell and Simon believed that the methods used by GPS could be used to solve a wide variety of different problems, thus giving rise to the term "general." To apply it to a specific problem, a "table of differences" for that problem would have to be supplied. The table would list all the possible differences that might arise and match them to operators, which, for that problem, would reduce the corresponding differences. GPS was, in fact, applied to a number of different logical problems and puzzles[13] and inspired later work in both artificial intelligence and in cognitive science. Its longevity as a problem-solving program itself and as a theory of human problem solving was short, however, and lives on only through its various descendants (about which more will be discussed later).

Heuristic search procedures were used in a number of AI programs developed in the early 1960s. For example, another one of Minsky's Ph.D. students, James Slagle, programmed a system called SAINT that could solve calculus problems, suitably represented as symbol structures. It solved 52 of 54 problems taken from MIT freshman calculus final examinations.[14] Much use of heuristics was used in programs that could play board games, a subject to which I now turn.

5.4 Game-Playing Programs

I have already mentioned some of the early work of Shannon and of Newell, Shaw, and Simon on programs for playing chess. Playing excellent chess requires intelligence. In fact, Newell, Shaw, and Simon wrote that if "one could devise a successful chess machine, one would seem to have penetrated to the core of human intellectual endeavor."[15]

Thinking about programs to play chess goes back at least to Babbage. According to Murray Campbell, an IBM researcher who helped design a world-champion chess-playing program (which I'll mention later), Babbage's 1845 book, *The Life of a Philosopher*, contains the first documented discussion of programming a computer to play chess.[16] Konrad Zuse, the German designer and builder of the Z1 and Z3 computers, used his programming language called Plankalkül to design a chess-playing program in the early 1940s.

In 1946 Turing mentioned the idea of a computer showing "intelligence," with chess-playing as a paradigm.[17] In 1948, Turing and his former undergraduate colleague, D. G. Champernowne, began writing a chess program. In 1952, lacking a computer powerful enough to execute the program, Turing played a game in which he simulated the computer, taking about half an hour per move. (The game was recorded. You can see it at http://www.chessgames.com/perl/chessgame?gid=1356927.) The program lost to a colleague of Turing, Alick Glennie; however, it is said that the program won a game against Champernowne's wife.[18]

After these early programs, work on computer chess programs continued, with off-again–on-again effort, throughout the next several decades. According to John McCarthy, Alexander Kronrod, a Russian AI researcher, said "Chess is the

Figure 5.5. Arthur Samuel. (Photograph courtesy of Donna Hussain, Samuel's daughter.)

Drosophila of AI" – meaning that it serves, better than more open-ended intellectual tasks do, as a useful laboratory specimen for research. As Minsky said, "It is not that the games and mathematical problems are chosen because they are clear and simple; rather it is that they give us, for the smallest initial structures, the greatest complexity, so that one can engage some really formidable situations after a relatively minimal diversion into programming."[19] Chess presents very difficult problems for AI, and it was not until the mid-1960s that the first competent chess programs appeared. I'll return to discuss these in a subsequent chapter.

More dramatic early success, however, was achieved on the simpler game of checkers (or draughts as the game is known in British English). Arthur Samuel (Fig. 5.5) began thinking about programming a computer to play checkers in the late 1940s at the University of Illinois where he was a Professor of Electrical Engineering. In 1949, he joined IBM's Poughkeepsie Laboratory and completed his first operating checkers program in 1952 on IBM's 701 computer. The program was recoded for the IBM 704 in 1954. According to John McCarthy,[20] "Thomas J. Watson Sr., the founder and President of IBM, remarked that the demonstration [of Samuel's program] would raise the price of IBM stock 15 points. It did."

[Apparently, Samuel was not the first to write a checkers-playing program. According to the *Encyclopedia Brittanica, Online*, "The earliest successful AI program was written in 1951 by Christopher Strachey, later director of the Programming Research Group at the University of Oxford. Strachey's checkers (draughts) program ran on the Ferranti Mark I computer at the University of Manchester, England. By the summer of 1952 this program could play a complete game of checkers at a reasonable speed."][21]

Samuel's main interest in programming a computer to play checkers was to explore how to get a computer to learn. Recognizing the "time consuming and costly procedure[s]" involved in programming, Samuel wrote "Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort."[22] Samuel's efforts were among the first in what was to become a very important part of artificial intelligence, namely, "machine learning." His first program that incorporated learning was completed in 1955 and demonstrated on television on February 24, 1956.

Before describing his learning methods, I'll describe in general how Samuel's program chose moves. The technique is quite similar to how moves were chosen in the eight-puzzle I described earlier. Except now, provision must be made for the fact that the opponent chooses moves also. Again, a tree of symbolic expressions, representing board positions, is constructed. Starting with the initial configuration, all possible moves by the program (under the assumption that the program moves first) are considered. The result is all the possible resulting board configurations branching out from the starting configuration. Then, from each of these, all possible moves of the opponent are considered – resulting in more branches, and so on.

If such a tree could be constructed for an entire game, a winning move could be computed by examination of the tree. Unfortunately, it has been estimated that there are about 5×10^{20} possible checkers positions. A leading expert in programming computers to play games, Jonathan Schaeffer, was able to "solve" checkers (showing that optimal play by both players results in a draw) by time-consuming analysis of around 10^{14} positions. He wrote me that "This was the result of numerous enhancements aimed at focussing the search at the parts of the search space where we were most likely to find what we needed."[23] I'll describe his work in more detail later.

Samuel's program then could necessarily construct only a part of the tree – that is, it could look only a few moves ahead. How far ahead it looked, along various of its branches, depended on a number of factors that need not concern us here. (They involved such matters as whether or not an immediate capture was possible.) Looking ahead about three moves was typical, although some branches might be explored (sparsely) to a depth of as many as ten moves. A diagram from Samuel's paper, shown in Fig. 5.6, gives the general idea. Samuel said that the "actual branchings are much more numerous."

So, how is the program to choose a move from such an incomplete tree? This problem is faced by all game-playing programs, and they all use methods that involve computing a score for the positions at the tips, or "leaves," of the tree (that is, the leaves of the incomplete tree generated by the program) and then "migrating" this score back up to the positions resulting from moves from the current position. First, I will describe how to compute the score, then how to migrate it back, and then how Samuel used learning methods to improve performance.

Samuel's program first computed the points to be awarded to positions at the leaves of the tree based on their overall "goodness" from the point of view of the program. Among the features contributing points were the relative piece advantage (with kings being worth more than ordinary pieces), the overall "mobility" (freedom to move) of the program's pieces, and center control. (The program had access to 38 such features but only used the 16 best of these at any one time.) The points contributed by each feature were then multiplied by a "weight" (reflecting the relative importance of its corresponding feature), and the result was summed to give an overall score for a position.

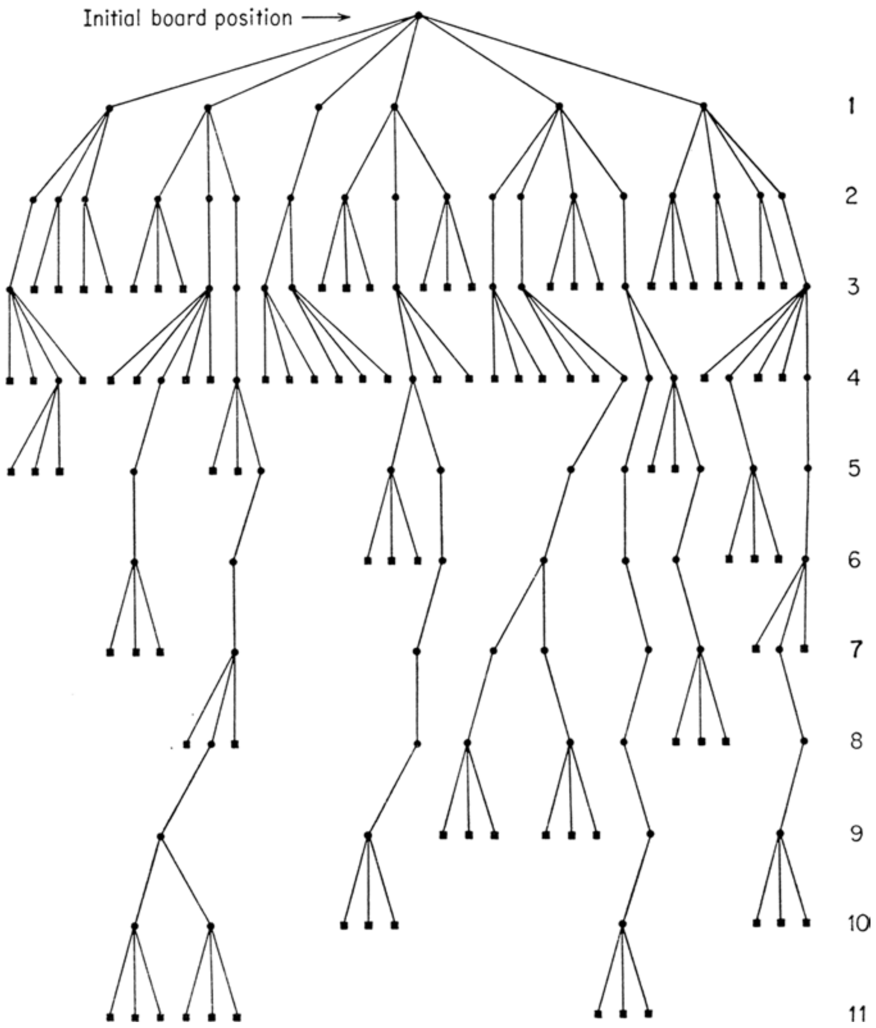

Figure 5.6. An illustrative checkers game tree. (From p. 74 of Edward A. Feigenbaum and Julian Feldman (eds.), *Computers and Thought*, New York: McGraw Hill, 1963.)

Starting with a position immediately above those at the tip of the tree, if it is a position for which it is the program's turn to move, we can assume that the program would want to move to that position with the highest score, so that highest score is migrated back to this "immediately above" position. If, however, it is a position from which it is the opponent's turn to move, we assume that the opponent would want to move to that position with the lowest score. In that case, the lowest score is migrated back to this immediately above position. This alternately "highest–lowest" migration strategy is continued back all the way up the tree and is called the "minimax" strategy.

[A simple modification of this strategy, called the "alpha–beta" procedure, is used to infer (correctly) from already-migrated scores that certain branches need not be examined at all – thus allowing other branches to be explored more deeply. Opinions differ about who first thought of this important modification. McCarthy and Newell and Simon all claim credit. Samuel told me he used it but that it was too obvious to write about.]

If one assumes that it is the program's turn to move from the current position, and that scores have already been migrated back to the positions just below it, the program would make its move to that position with the highest score. And then the game would continue with the opponent making a move, another stage of tree growth, score computation and migration, and so on until one side wins or loses.

One of the learning methods in Samuel's program adjusted the values of the weights used by the scoring system. (Recall that weight adjustments in Pandemonium and in neural networks were ways in which those systems learned.) The weights were adjusted so that the score of a board position (as computed by the sum of the weighted feature scores) moved closer to the value of its migrated score after finishing a search. For example, if the score of an initial position was computed (using the weights before adjustment) to be 22, and the migrated score of that position after search was 30, then the weights used to compute the score of the initial position were adjusted in a manner so that the new score (using the adjusted value of the weights) was made closer to 30, say 27. (This technique foreshadowed a very important learning method later articulated by Richard Sutton called "temporal-difference learning.") The idea here was that the migrated score, depending as it did on looking ahead in the game, was presumed to be a better estimate than the original score. The estimating procedure was thereby improved so that it produced values more consistent with the "look-ahead" score.

Samuel also used another method called "rote learning" in which the program saved various board positions and their migrated scores encountered during actual play. Then, at the end of a search, if a leaf position encountered was the same as one of these stored positions, its score was already known (and would not have to be computed using the weights and features). The known score, based as it was on a previous search, would presumably be a better indicator of position value than would be the computed score.

Samuel's program also benefitted from the use of "book games," which are records of the games of master checkers players. In commenting about Samuel's work, John McCarthy wrote that "checker players have many volumes of annotated games with the good moves distinguished from the bad ones. Samuel's learning program used *Lee's Guide to Checkers*[24] to adjust its criteria for choosing moves so that the program would choose those thought good by checker experts as often as possible."

Samuel's program played very good checkers and, in the summer of 1962, beat Robert Nealey, a blind checkers master from Connecticut. (You can see a game played between Mr. Nealey and Samuel's program at http://www.fierz.ch/samuel.htm.) But, according to Jonathan Schaeffer and Robert Lake, "In 1965, the program played four games each against Walter Hellman and Derek Oldbury (then playing a match for the World Championship), and lost all eight games."[25]

Notes

1. A. Newell and H. A. Simon, "The Logic Theory Machine: A Complex Information Processing System," *Proceedings IRE Transactions on Information Theory*, Vol. IT-2, pp. 61–79, September 1956, and A. Newell, J. C. Shaw, and H. A. Simon, "Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristics," *Proceedings of the 1957 Western Joint Computer Conference*, Institute of Radio Engineers, pp. 218–230, 1957. [81]
2. Alfred North Whitehead and Bertrand Russell, *Principia Mathematica*, Vol. 1, Cambridge: Cambridge University Press, 1910. [81]
3. Allen Newell and Herbert A. Simon, "Computer Science as Empirical Inquiry: Symbols and Search," *Communications of the ACM*, Vol. 19, No. 3, pp. 113–126, March 1976. [81]
4. Claude E. Shannon, "Programming a Computer for Playing Chess," *Philosophical Magazine*, Ser. 7, Vol. 41, No. 314, March 1950. Text available online at http://www.pi.infn.it/~carosi/chess/shannon.txt. (The paper was first presented in March 1950 at the National Institute for Radio Engineers Convention in New York.) [84]
5. Pamela McCorduck, *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*, p. 106, San Francisco: W. H. Freeman and Co., 1979. [84]
6. John McCarthy, "A Tough Nut for Theorem Provers," Stanford Artificial Intelligence Project Memo No. 16, July 17, 1964; available online at http://www-formal.stanford.edu/jmc/toughnut.pdf. [84]
7. Saul Amarel, "On Representations of Problems of Reasoning About Actions," in Donald Michie (ed.), *Machine Intelligence 3*, pp. 131–171, Edinburgh: Edinburgh University Press, 1968. [85]
8. Herbert Gelernter, "Realization of a Geometry-Theorem Proving Machine," *Proceedings of the International Conference on Information Processing*", pp. 273–282, Paris: UNESCO House, Munich: R. Oldenbourg, and London: Butterworths, 1960. Also in Edward A. Feigenbaum and Julian Feldman (eds.), *Computers and Thought*, pp. 134–152, New York: McGraw Hill, 1963. [85]
9. From http://www.math.niu.edu/~rusin/known-math/99/minsky. [86]
10. Allen Newell, J. C. Shaw, and Herbert A. Simon, "Report on a General Problem-Solving Program," *Proceedings of the International Conference on Information Processing*, pp. 256–264, Paris: UNESCO House, Munich: R. Oldenbourg, and London: Butterworths, 1960. [87]
11. For more about GPS as a theory and explanation for human problem solving, see Allen Newell and Herbert Simon, "GPS, a Program That Simulates Human Thought," in H. Billings (ed.), *Lernende Automaten*, pp. 109–124, Munich: R. Oldenbourg KG, 1961. Reprinted in *Computers and Thought*, pp. 279–293. [88]
12. I adapt an example from http://www.math.grinnell.edu/~stone/events/scheme-workshop/gps.html. [88]
13. See George Ernst and Allen Newell, *GPS: A Case Study in Generality and Problem Solving*, New York: Academic Press, 1969. [89]
14. James R. Slagle, "A Heuristic Program That Solves Symbolic Integration Problems in Freshman Calculus," Ph.D. dissertation, MIT, May 1961. For an article about SAINT, see James R. Slagle, "A Heuristic Program That Solves Symbolic Integration Problems in Freshman Calculus," *Journal of the ACM*, Vol. 10, No. 4, pp. 507–520, October 1963. [89]
15. Allen Newell, J. Shaw, and Herbert Simon, "Chess-Playing Programs and the Problem of Complexity," *IBM Journal of Research and Development*, Vol. 2, pp. 320–335, October 1958. [89]

16. Chapter 5 of *Hal's Legacy: 2001's Computer as Dream and Reality*, David G. Stork (ed.), Cambridge, MA: MIT Press, 1996. See the Web site at http://mitpress.mit.edu/e-books/Hal/chap5/five3.html. [89]
17. Andrew Hodges, "Alan Turing and the Turing Test," in *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*, Robert Epstein, Gary Roberts, and Grace Beber (ed.), Dordrecht, The Netherlands: Kluwer, 2009. See A. M. Turing, "Proposed Electronic Calculator," report for National Physical Laboratory, 1946, in *A. M. Turing's ACE Report of 1946 and Other Papers*, B. E. Carpenter and R. W. Doran (eds.), Cambridge, MA: MIT Press, 1986. [89]
18. http://en.wikipedia.org/wiki/Alan_Turing. [89]
19. Marvin Minsky (ed.), "Introduction," *Semantic Information Processing*, p. 12, Cambridge, MA: MIT Press, 1968. [90]
20. From a Web retrospective at http://www-db.stanford.edu/pub/voy/museum/samuel.html. [90]
21. See Christopher Strachey, "Logical or Non-mathematical Programmes," *Proceedings of the 1952 ACM National Meeting (Toronto)*, pp. 46–49, 1952. [90]
22. Arthur L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 210–229, 1959. Reprinted in Edward A. Feigenbaum and Julian Feldman (eds.), *Computers and Thought*, p. 71, New York: McGraw Hill, 1963. [91]
23. E-mail of February 14, 2009. [91]
24. John W. Dawson, *Lee's Guide to the Game of Draughts*, Revised Edition, London: E. Marlborough, 1947. [93]
25. Jonathan Schaeffer and Robert Lake, "Solving the Game of Checkers," *Games of No Chance*, pp. 119–133, MSRI Publications, Vol. 29, 1996. (Available online at http://www.msri.org/communications/books/Book29/files/schaeffer.pdf.) [93]