

GAME PLAYING

Every game of skill is susceptible of being played by an automaton.

-Charles Babbage

(1791-1871), English mathematician, philosopher, inventor and mechanical engineer

OVERVIEW

Games hold an inexplicable fascination for many people, and the notion that computers might play games has existed at least as long as computers. Charles Babbage, the nineteenth-century computer architect, thought about programming his Analytical Engine to play chess and later of building a machine to play tic-tac-toe [Bowden, 1953]. Two of the pioneers of the science of information and computing contributed to the fledgling computer game-playing literature. Claude Shannon [1950] wrote a paper in which he described mechanisms that could be used in a program to play chess. A few years later, Alan Turing described a chess-playing program, although he never built it. (For a description, see Bowden [1953].) By the early 1960s, Arthur Samuel had succeeded in building the first significant, operational game-playing program. His program played checkers and, in addition to simply playing the game, could learn from its mistakes and improve its performance [Samuel, 1963].

There were two reasons that games appeared to be a good domain in which to explore machine intelligence:

- They provide a structured task in which it is very easy to measure success or failure.
- They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position.

The first of these reasons remains valid and accounts for continued interest in the area of game playing by machine. Unfortunately, the second is not true for any but the simplest games. For example, consider chess.

- The average branching factor is around 35.
- In an average game, each player might make 50 moves.
- So in order to examine the complete game tree, we would have to examine 35^{100} positions.

Thus it is clear that a program that simply does a straightforward search of the game tree will not be able to select even its first move during the lifetime of its opponent. Some kind of heuristic search procedure is necessary.

One way of looking at all the search procedures we have discussed is that they are essentially generate-and-test procedures in which the testing is done after varying amounts of work by the generator. At one extreme, the generator generates entire proposed solutions, which the tester then evaluates. At the other extreme, the generator generates individual moves in the search space, each of which is then evaluated by the tester and the most promising one is chosen. Looked at this way, it is clear that to improve the effectiveness of a search-based problem-solving program two things can be done:

- Improve the generate procedure so that only good moves (or paths) are generated .
- Improve the test procedure so that the best moves (or paths) will be recognized and explored first.

In game-playing programs, it is particularly important that both these things be done. Consider again the problem of playing chess. On the average, there are about 35 legal moves available at each turn. If we use a simple legal-move generator, then the test procedure (which probably uses some combination of search and a heuristic evaluation function) will have to look at each of them. Because the test procedure must look at so many possibilities, it must be fast. So it probably cannot do a very accurate job.

Suppose, on the other hand, that instead of a legal-move generator, we use a *plausible-move generator* in which only some small number of promising moves are generated. As the number of legal moves available increases, it becomes increasingly important to apply heuristics to select only those that have some kind of promise. (So, for example, it is extremely important in programs that play the game of go [Benson et al., 1979].) With a more selective move generator, the test procedure can afford to spend more time evaluating each of the moves it is given so it can produce a more reliable result. Thus by incorporating heuristic knowledge into both the generator and the tester, the performance of the overall system can be improved.

Of course, in game playing, as in other problem domains, search is not the only available technique. In some games, there are at least some times when more direct techniques are appropriate. For example, in chess, both openings and endgames are often highly stylized, so they are best played by table lookup into a database of stored patterns. To play an entire game then, we need to combine search-oriented and nonsearch-oriented techniques.

The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached. In the context of game-playing programs, a goal state is one in which we win. Unfortunately, for interesting games such as chess, it is not usually possible, even with a good plausible-move generator, to search until a goal state is found. The depth of the resulting tree (or graph) and its branching factor are too great. In the amount of time available, it is usually possible to search a tree only ten or twenty moves (called *ply* in the game-playing literature) deep. Then, in order to choose the best move, the resulting board positions must be compared to discover which is most advantageous.