

# Adversarial Search

Dana S. Nau, University of Maryland

*Adversarial search*, or *game-tree search*, is a technique for analyzing an adversarial game in order to try to determine who can win the game and what moves the players should make in order to win. Adversarial search is one of the oldest topics in Artificial Intelligence. The original ideas for adversarial search were developed by Shannon in 1950 and independently by Turing in 1951, in the context of the game of chess—and their ideas still form the basis for the techniques used today. Computer programs based on adversarial search techniques are now as good as or better than humans in several popular board games and card games (see Table 1).

Table 1. Popular board games that computer programs can play as well or better than humans.

| Game       | Best programs   | Ability                   | Primary techniques                |
|------------|---|---------------------------|-----------------------------------|
| Checkers   | Chinook (Schaeffer, 1998)                             | Better than best humans   | Game-tree search                  |
| Othello    | Hannibal (Geoffroy, 2000) and Logistello (Buro, 2000) | Better than best humans   | Neural networks, game-tree search |
| Scrabble   | Maven and CrossWise (Alexander, 2001)                 | Comparable to best humans | Simulation                        |
| Chess      | Deep Blue (IBM, 1997)                                 | Comparable to best humans | Game-tree search                  |
| Backgammon | TD-gammon (Tesauro, 1995)                             | Comparable to best humans | Neural networks                   |

## Game trees

Adversarial search is based on the notion of a *game tree*, a mathematical structure that represents the positions that might result from every possible series of moves in a game. Game trees can be used to model any game that satisfies the following restrictions:

- The game is played by two players, who make moves sequentially rather than simultaneously. This excludes most popular video games, and also the kinds of games studied in the branch of economics known as classical game theory.
- The game is *finite*. At each turn a player can choose from only a finite number of possible moves, and the game is guaranteed to end within some finite number of moves.
- The game is *zero-sum*; i.e., the amount that one player wins is precisely equal to the amount that the other player loses.
- The game contains no elements of chance (although we will later discuss how to incorporate certain kinds of chance elements into game-tree searching).

Figure 1 shows a simple example of a game tree for a game between two players called Max and Min. Square nodes represent positions where it is Max's move, and round nodes represent positions where it is Min's move. The leaf nodes represent positions where the game has ended, and the numbers below them represent the payoffs for Max (recall that Min's payoff is always the negative of Max's payoff).

## The minimax algorithm

At the node  $n_4$ , of Figure 1, Max's best move is to go to  $n_8$  to get the payoff of 5. At the node  $n_5$ , Max's best move is to go to  $n_{10}$  to get the payoff of 8. If we know that Max will always make the best move, then this means that at the node  $n_2$ , Min's best move is to go to  $n_4$  so that Max's payoff will be 5 rather than 8. Generalizing this analysis, it follows that if both players play perfectly from some node  $n$  onward, then the payoff for Max is the *minimax value* of  $n$ , which is the value computed by the following algorithm:<sup>1</sup>

```
procedure minimax( $n$ )
  if  $n$  is a leaf node then return the payoff for Max at  $n$ ;
  compute the children  $n_1, \dots, n_k$  of  $n$ ;
  for  $i = 1, \dots, k$ , let  $m_i = \text{minimax}(n_i)$ ;
  if it is Max's turn to move at  $n$  then return  $\max(m_1, m_2, \dots, m_k)$ ;
  else return  $\min(m_1, m_2, \dots, m_k)$ 
```

As an example, Figure 2 gives the minimax values for the tree in Figure 1. These values can be used to decide what move to make at any node of the tree: Max should always move to the child having has the largest minimax value, and Min should always move to the child that has the smallest minimax value. Theoretically, this rule is only correct against infallible opponents (Pearl, 1984), but in practice it works well against any good opponent.

## Alpha-beta pruning

Suppose that we are computing the minimax value for the node  $n_1$  of Figure 1, and that we have already computed  $\text{minimax}(n_4) = 10$  and  $\text{minimax}(n_{10}) = 11$  as shown in Figure 3. Then

$$\text{minimax}(n_2) = \min(5, \text{minimax}(n_5)) \leq 5;$$

$$\text{minimax}(n_5) = \max(8, \text{minimax}(n_{11})) \geq 8.$$

It follows that we do not need to know what  $\text{minimax}(n_{11})$  is, because there is no way for  $\text{minimax}(n_{11})$  to affect  $\text{minimax}(n_2)$  and  $\text{minimax}(n_1)$ . Generalizing this analysis gives us the following algorithm, which is called the *alpha-beta pruning algorithm*:

```
procedure alphabeta ( $n, \alpha, \beta$ )
  if  $n$  is a leaf node then return the payoff for Max at  $n$ 
  else if it is Max's move at  $n$  then
    let  $n_1, \dots, n_k$  be the children of  $n$ 
    for  $i = 1, \dots, k$  do
       $\alpha = \max(\alpha, \text{alphabeta}(n_i, \alpha, \beta))$ 
      if  $\alpha \geq \beta$  then return  $\alpha$ 
    end
    return  $\alpha$ 
  else /* it must be Min's move */
```

---

<sup>1</sup> The algorithm's name comes from the famous *minimax theorem* of von Neuman and Morgenstern. However, the principle embodied in the algorithm—that a player should maximize the minimum payoff for every alternative—is what decision theorists call the *maximin decision criterion*.

```

    let  $n_1, \dots, n_k$  be the children of  $n$ 
    for  $i = 1, \dots, k$  do
         $\beta = \min(\beta, \text{alphabeta}(n_i, \alpha, \beta))$ 
        if  $\beta \leq \alpha$  then return  $\beta$ 
    end
    return  $\beta$ 
end if
end alphabeta

```

Alpha-beta pruning is guaranteed to compute a node's minimax value, and in general it will visit far fewer nodes than the minimax algorithm. There also are several other algorithms having this same property (Pearl, 1984), but because of its simplicity and low overhead, alpha-beta pruning is the one that is most widely used. In the worst case, alpha-beta pruning will visit every node that the minimax algorithm visits, but in the best case, it can search a tree of roughly twice the depth as the minimax algorithm in roughly the same amount of time (Knuth and Moore, 1975).

## Limiting the depth of the search

Most game trees contain far too many nodes to allow the entire tree to be searched quickly, even with a tree-pruning algorithm such as alpha-beta. Another way to visit fewer nodes is to stop searching at some arbitrary search depth  $d$ , and instead to estimate the minimax values of the nodes at this depth using a *static evaluation function*. This gives the following modification to the alpha-beta algorithm:

```

procedure alphabeta ( $n, d, \alpha, \beta$ )
    if  $n$  is a leaf node or  $d = 0$  then return  $e(n)$ 
    else if it is Max's move at  $n$  then
        let  $n_1, \dots, n_k$  be the children of  $n$ 
        for  $i = 1, \dots, k$  do
             $\alpha = \max(\alpha, \text{alphabeta}(n_i, d-1, \alpha, \beta))$ 
            if  $\alpha \geq \beta$  then return  $\alpha$ :
        end
        return  $\alpha$ :
    else /* it must be Min's move */
        let  $n_1, \dots, n_k$  be the children of  $n$ 
        for  $i = 1, \dots, k$  do
             $\beta = \min(\beta, \text{alphabeta}(n_i, d-1, \alpha, \beta))$ 
            if  $\beta \leq \alpha$  then return  $\beta$ 
        end
        return  $\beta$ 
    end if
end alphabeta

```

A search to depth  $d$  is likely to yield inaccurate results if something happens (such as an exchange of material in the game of chess) that makes a large change in the minimax values just beyond the cutoff depth  $d$ . In order to avoid this *horizon effect*, it is best to modify the above

algorithm so that instead of stopping when  $d = 0$ , it keeps searching until it reaches a node that is *quiescent* (e.g., in chess, a node where there are no pending exchanges of material). However, this means that at different branches of the search tree, the evaluation function may be applied to nodes of different depths. Since a move usually strengthens the position of the player who makes the move, nodes at which Max has just moved are likely to get higher evaluations than nodes where Min has just moved. This can cause the computer program erroneously to prefer branches that end at an odd depth. To overcome this, practical implementations of game-tree search will usually include a *biasing factor* that is added to node evaluations at even depths and subtracted from node evaluations at odd depths.

## Speeding up the search

Although there exist games and game trees in which searching deeper will produce worse decision-making (Nau, 1983), a deeper search will produce significantly better decisions in nearly all practical situations (Kaindl, 1988). However, in order to do a deeper search, a search algorithm must visit exponentially many more nodes, which makes it impossible to search very deeply unless the algorithm is extremely fast. Here are several techniques for speeding up the search.

*Databases of standard openings.* Sometimes it is possible to develop databases of various standard ways to start a game. As long as the computer's opponent uses one of these standard openings, the computer can make moves very quickly by retrieving them from the database.

*Thinking on the opponent's time.* In order to decide what move to make, a search algorithm will need to predict what node the opponent is likely to move to. After making its move, a computer program can immediately begin a game-tree search starting at this node, thereby saving some time if the opponent actually makes the predicted response.

*Iterative deepening.* One problem with the procedure described in the previous section is deciding what value to use for the cutoff depth  $d$ . If we make  $d$  too large, then the search will take too long—but we make  $d$  too small, then it will reduce the accuracy of the decision-making. One way to solve this problem is to choose a value for  $d$  that we know is not too large. If we finish our search more quickly than expected, then we can increment  $d$  and search again. Since the number of nodes visited usually grows exponentially with the search depth, this usually will not take significantly more time than if we had used the correct value of  $d$  the first time around.

*Transposition tables.* The term “game tree” is a misnomer: most games actually correspond to graphs such as the one in Figure 4. We can take advantage of this by using a *transposition table*, a cache of recently visited positions and their minimax values. If we visit a node that we visited a short time ago, we can retrieve its value from the transposition table rather than computing it again.

*Move ordering.* For alpha-beta pruning, the difference between the best case and the worst case depends on the order in which the moves are considered. At a node where a cutoff is to occur, the best case is if the cutoff occurs at the first child that is visited—which occurs if the children appear in order of decreasing minimax values when it is Max's move, or in order of increasing minimax values when it is Min's move. A way to make this occurrence more likely is to sort a

node's children in decreasing order of their evaluation-function values when it is Max's move, and in increasing order of their evaluation-function values when it is Min's move.

*Specialized hardware.* An expensive but effective technique, used in the Deep Blue program, is to design special-purpose computer hardware to speed up its game-tree search.

## Other Kinds of Games

Not all games are amenable to the “brute-force” game-tree search techniques described above. Here are several examples:

- *Backgammon* incorporates a chance element, namely the rolls of the dice. The game-tree model can be generalized to model the dice rolls, but it produces too large a game tree to be searched in a reasonable amount of time. However, through the use of machine-learning techniques, static evaluation functions can be created that are good enough to work well with only a shallow game-tree search. Thus, the best computer backgammon programs play as well as the best human beings (Tesauro, 1995).
- *Bridge* incorporates both chance and imperfect information: the cards are dealt randomly, and no player knows all of the other players' cards. As in backgammon, the game-tree model can be extended to model this, but it produces too large a game tree to search in a reasonable time. Research in bridge has produced a variety of techniques for reducing the size of this game tree, ranging from Monte Carlo simulation to AI planning. Bridge programs are not yet as good as the best humans, but may get there in the next few years.
- *Poker* includes not only chance and imperfect information, but also presents the difficulty of recognizing what kind of betting strategy the opponent is using. Current programs play much worse than good human players.
- *Go*, unlike the above games, fits the game-tree model perfectly. However, its game tree is huge because there are hundreds of possible moves at each node. This makes it impossible to search the game tree to any significant depth. Currently, the best available computer programs are ranked below the level of an average amateur.
- Most current *video games* are action-oriented games that emphasize hand-to-eye coordination. However, a few of them (warcraft, command and conquer, civilization, etc.) involve longer-term strategic reasoning. These games incorporate several features not discussed above. For example, one may need to plan actions that have various time durations, which durations may overlap those of the opponents' moves. Computer-science game-playing research is only just beginning to address such problems, and thus the strategic reasoning capabilities of such computer programs still remain very poor.

## Conclusion

The biggest successes for adversarial search techniques have been in board games such as chess, checkers, othello, and backgammon. The best computer programs for these games are now as good or better than the best human players. However, the techniques used in most of these programs work rather differently from human thought processes: the programs work by examining as many board positions as possible within the time constraints, often examining many thousands of positions in order to decide which move to make—and in contrast, the best

humans examine at most a few dozen positions in order to decide upon their moves. How humans are able to do this is still ill-understood.

## Glossary

*Game tree*: a tree whose nodes and edges correspond to game positions and to moves, respectively.

*Minimax algorithm*: algorithm that computes a node's payoff by taking the maximum when it is one player's move, and the minimum when it is the other player's move.

*Payoff*: a numerical quantity representing the amount won by a player in a game.

*Search*: the process of solving a problem by looking at many different alternatives.

*Static evaluation function*: a function that returns an approximation of a node's payoff without doing any search.

*Tree*: a mathematical structure consisting of *nodes* and of *edges* connecting the nodes, such that there is at most one path (i.e., sequence of edges) between any two nodes of the tree.

*Zero-sum game*: a game in which the amount one player wins equals the amount the other player loses.

## Further Reading

S. Alexander. "Scrabble FAQ." <<http://www.teleport.com/~stevena/scrabble/faqtext.html>>, 2001.

M. Buro. "LOGISTELLO's Homepage." <<http://www.neci.nj.nec.com/homepages/mic/log.html>>, 2000.

L. Geoffroy. "Hannibal Home Page." <<http://www.cam.org/~bigjeff/Hannibal.html>>, 2000.

IBM Corporation. "Kasparov Versus Deep Blue: The Rematch." <<http://www.research.ibm.com/deepblue/home/html/b.html>>, 1997.

H. Kaindl. "Minimaxing: Theory and Practice." *AI Magazine*, Fall 1988, p. 69–76.

D. Knuth and R. Moore (1975). "An Analysis of Alpha-Beta Pruning." *Artificial Intelligence* **6**:293–326.

D. Nau. "Pathology on Game Trees Revisited, and an Alternative to Minimaxing." *Artificial Intelligence* **21**(1,2):221–244, 1983.

J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley, 1984.

J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer Verlag, 1997. Further information available at <<http://www.cs.ualberta.ca/~chinook/>>.

C. Shannon. "Programming a Computer for Playing Chess." *Philosophical Magazine* (Series 7), vol. 41, pp. 256–275, 1950.

G. Tesauro. "Temporal Difference Learning and TD-Gammon," *Communications of the ACM* **38**(3), March 1995. Also available at <<http://www.research.ibm.com/massive/tld.html>>.