Adversarial Search

Michael Rose

In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us

5.1 | Games

We begin with a definition of the optimal move and an algorithm for finding it. **Pruning** allows us to ignore portions of the search tree that make no difference to the final choice. **Heuristic Evaluation Functions** allow us to approximate the true utility of a state without doing a complete search.

A game can be formally defined as a kind of search problem with the following elements:

- S_0: The initial state, which specifies how the game is set up at the start
- Player(s): defines which player has the move in a state
- Actions: returns the set of legal moves in a state
- Result: The transition model, which defines the result of a move
- Terminal-Test: True when the game is over, false ow
- Utility: Defines the final numeric value for a game that ends in a terminal state s for every player p

A **zero-sum game** is defined as one where the total payoff to all players is the same for every instance of the game.

The initial state, actions function, and result function define the **game tree** for the ame - a tree where the nodes are game states and the edges are moves. For chess there are over 10^{40} nodes, so its more of a theoretical construct.

5.2 | Optimal Decisions in Games

Given a game tree, the optimal strategy for an adversarial game can be determined from the minimax value of each node. This is the utility of being in the corresponding state, assuming that both players play optimally from there to the end of the game. The minimax decision is the action which is the optimal choice for leading to the highest minimax value.

5.2.1 | The Minimax Algorithm

The **minimax algorithm** computes the minimax decision from the current state. It is a recursive algorithm which proceeds down the leaves of the tree and then the minimax values are backed up through the tree as the recursion unwinds. The minimax algorithm performs a complete depth-first exploration of the game tree.

5.3 | Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. We can compute the correct minimax decision without looking at the entire tree by applying the idea of pruning.

We want to examine a technique called **Alpha-Beta Pruning**, which, when applied to a standard minimax tree returns the same move as minimax would, but prunes away branches that cannot possible influence the final decision.

The general principle is this:

Consider a node n somewhere in the tree, such that the player has a choice of moving to that node. If player has a better choice m at either the parent node of n or at any choice further up, then n will never be reached in actual play. Therefore, once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Alpha-Beta pruning gets its name from the following parameters that describe bounds on the backed up values that appear anywhere along the path:

α = The value of the best choice we have found so far at any choice point along the path for max β = The value of the best choice we have found so far at any choice point along the path for min

Alpha-Beta search updates the values of α and β as it goes along and prunes the remaining branches at a node as soon as the current node is known to be worse than the current α or β value for max or min respectively.

5.3.1 | Move Ordering

The effectiveness of alpha-beta pruning is high;y dependent on the order in which the states are examined. Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit.

5.4 | Imperfect Real-Time Decisions

Alpha-Beta pruning still needs to search all the way to the terminal states for at least a portion of the search space. This depth is usually not practical.

Claude Shannon in *Programming a Computer for Playing Chess* proposed altering the minimax or alpha-beta pruning in the following 2 ways:

1. Replace the utility function by a heuristic evaluation function which estimates the positions utility (eval)
2. Replace the terminal test by a cutoff test that decides when to apply eval

5.4.1 | Evaluation Functions

An evaluation function returns an estimate of the expected utility of a game from a given position.

The evaluation function should: 1. order the terminal states (e.g. win > draw > loss) 2. The computation should be fast 3. For nonterminal states, the function should be strongly correlated with the actual chances of winning

We can codify a game into a weighted linear function:

$$\text{EVAL}(s) = \sum_{i=1}^{n} w_i f_i(s)$$

where each w_i is a weight and each f_i is a feature of the position.

5.4.2 | Cutting Off Search

We can set a fixed depth limit so that cutoff-test returns true for all depth greater than some fixed d. We could also use iterative deepening. As a bonus, iterative deepening also helps with move ordering.

The evaluation function should be applied only to positions that are **quiescent** - that is, unlikely to exhibit wild swings in valye in the near future.

5.4.3 | Forward Pruning

It is also possible to do **forward pruning**, meaning that some moves at a given node are pruned immediately without further consideration. An example of this would be to do a **beam search**; on each ply, consider only a beam of the n best moves rather than all possible moves. This is dangerous because it may prune the actual best move, but is quicker than considering all n moves.

The **probabilistic cut algorithm** is a forward pruning version of alpha-beta search that uses statistics gained from prior experience to lessen the chance that the best move will be pruned.

5.4.4 | Search vs. Lookup

Most game playing programs use lookup tables rather than search for opening and ending of games.

5.5 | Stochastic Games

Stochastic games must include **chance nodes** in addition to max and min nodes. Since we can't figure out the best possible move due to chance, we must calculate the expected value of a position. This leads to us generalizing the minimax value for deterministic games to an expecti-minimax value for games with chance nodes.

5.5.1 | Evaluation Functions for Games of Chance

In order to avoid scaling issues in the evaluation function, it must be a linear transformation of the probability of winning from a position. Since we consider all possible values, it takes $O(b^m n^m)$ where n is the number of distinct rolls. This is very inefficient.

We can apply something like alpha-beta pruning to game trees with chance nodes. If we bound the possible values of our utility function, we can arrive at a bounds for the average without checking all the nodes.

We can also use **Monte Carlo Simulation**. From the start position (with an alpha-beta or other search algorithm) have the algorithm play thousands of games against itself using random dice rolls. In many cases the resulting win percentages for different positions is a good approximation of the value of the position.