Computer Game-Playing

3.1 Game Players and Static Analyzers

Unfortunately, for many interesting games, the tree is far too large to construct, let alone to search. The solution is to cut the tree off at a certain level. Rather than exploring further, one tries to estimate the goodness or badness of that position. For some games, we can see just by glancing that one player is doing better than the other.

This heuristic measure of goodness or badness is known as a ***static analyzer***. The same minimax procedure is used, but now at a certain level, called the ***ply***, the minimax procedure calls the static analyzer rather than itself.

For example, in chess a simple static analyzer is the traditional point count: Queen = 9, Rook = 5, Bishop = Knight = 3, Pawn = 1. Of course, if you're about to be mated this might give a false impression. (What about a static analyzer for tictactoe?)

No matter how deep one goes, however, there's always a problem with cutting off the search just after taking your opponent's queen. The static evaluator thinks highly of this, even though your queen is about to be captured in return. This is called the ***horizon effect***.

Indeed, actual game-playing is a lot more complicated. Many of the fundamental ideals were developed by Samuels who in the 50s produced a checkers player that crushed a human. There are two obvious areas for improvement: a better static evaluator or a better search strategy.

3.2 Training a Static Evaluator

One of Samuels' great ideas was a method of setting the static evaluator. He constructed a number of possible heuristic measures x_i. Then the static evaluator was a linear combination of these:

$$\sum a_i x_i$$

But what were suitable values of a_i? The coefficients a_i were then tuned by learning!

In particular a hill-climbing approach was used. After playing a game, a loss was blamed on large weights, which were thus reduced. On the other hand, a win was credited to the large weights, which were increased. The program introduced a bit of randomness (to try to avoid foothill/ridges). It trained against a human or itself.

This method has an interesting side-effect. By deliberately losing to such a program, one can train it to play a bad game!

3.3 Better searching

Another idea is to try to avoid the problems of limited search. Three approaches are:

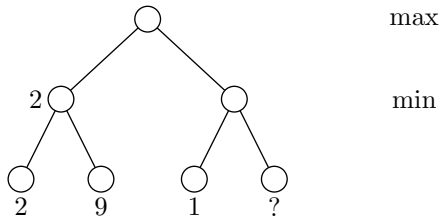
- *progressive deepening.* find the best move at 2 ply, then at 4 ply, and so on. This has the advantage that one always has a candidate move. And the extra computation is minimal.
- *selective deepening.* Either (a) always search the tree until quiet situation reached; or (b) always search deeper when one move is deemed a lot better (or worse?) than the others. Why useful?
- *tapering.* give more attention to plausible moves: reduce search breadth as the depth increases and as plausibility decreases.

3.4 Alpha–Beta Pruning

The branch-and-bound improvement in normal search has a counterpart here: Alpha–Beta pruning. There is a certain mystique to this method.

The idea is that if an opponent has one response that shows a potential move to be bad, we do not need to explore the other responses. Alpha–Beta stops work on guaranteed losers.

How does this work? Consider the following example.



Suppose we know that the value of the left-hand child is 2. And we find the first grandchild on the right to have value 1. Then we do not need to calculate the value of the other grandchild.

Why? Well, the value of the right child is at most 1. The maximum of two values, one of which is 2 and the other at most 1, is 2. So the value of the game is 2. The first grandchild show the right-hand option is a bad idea—we do not have to calculate exactly how bad it is.

The paperwork is translated into the mysterious Alpha–Beta pruning procedure. We maintain at each node two values: Alpha and Beta. They keep track of bounds on the actual and worthwhile values of the node. Note that $\alpha \leq \beta$ always.

> *Suppose the node is a minimizer. Then Beta is an upped bound on the actual value of the node: the value can be obtained by the minimizer choosing some*

child already examined. And Alpha is a lower bound on the worthwhile value of the node: the value can be obtained by the maximizer avoiding the node.

There is a corresponding meaning at Maximizer nodes. Then we traverse the tree recursively just as in MiniMax. The process starts be calling the root with $\alpha = -\infty$ and $\beta = +\infty$.

FLOAT ALPHA-BETA(α,β)

- if ply of search reached, then compute static evaluator and return result
- if node is Minimizer then
  - while (not all children examined) and ($\alpha < \beta$)
    $\beta :=$ min of β and Alpha-Beta() on next child
  - return β
- if node is Maximizer then
  - while (not all children examined) and ($\alpha < \beta$)
    $\alpha :=$ max of α and Alpha-Beta() on next child
  - return α

This method does not necessarily prune much off the tree. Even in the best case it only slows the exponential growth and limits the branching factor. But that might be enough to squeeze out an extra ply of search.

3.5 Improvements

Chess programs have continued to evolve. A lot of thought went into the control of search—more sophisticated pruning etc. Recent success of Deep Blue have suggested that naive search (with selective deepening) might not be so bad after all—if you have a truck-load of ***chips*** each designed as an incredibly fast static analyzer.

One obvious idea is that the same position can appear in two different ways (if for example we interchange the first two moves of one of the players). So we could store the result and not re-evaluate the tree at this point. The savings would be enormous.

However, the amount of storage needed is large. But more important is that the computation needed to keep track of this seems to be not worth the savings. For each new position, we have to search through the collection of previously analyzed positions, which takes time. But there are places where so-called ***transposition tables*** are used. In particular, in trying to analyze positions with very few pieces on the board.

Another area of improvement is to try to encapsulate human knowledge (for example, opening moves in chess or backgammon). Or to do a tremendous amount of offline computation (and so play a perfect game with very few pieces on the board). This knowledge is store in a ***database***.

3.6 Imperfect Information

Some games are still basically two-person games, but there is incomplete information. Many card games fall into this category. Earlier attempts at bridge tried an Expert system approach, with little success.

Some recent Bridge programs have tried a rather unintelligent approach. Randomly place the unknown cards subject to the known constraints and then play the hand, which now has perfect information, using Minimax. Repeat many many times. Then pick the move with the best record.

Exercises

3.1. Propose a static analyzer for tictactoe.

3.2. For the tree of Exercise 2.3, what nodes would not be examined if α–β pruning were used? Explain.

3.3. Suppose we search a full tree for a game where there are only **two** possible outcomes, win or lose. Explain how α–β pruning would behave.

3.4. Consider a game tree where each player has 2 choices at each level and which lasts three moves.

(a) Suppose the values of the 8 leaves are in order 1, 4, 5, 2, 7, 8, 3, 6. What's the value of the game if (i) first player is maximizer? (ii) first player is minimizer?

(b) Show by a suitable labeling of the leaves with 1 up to 8 that α–β-pruning sometimes provides no savings.

3.5. **Project.** Propose a static analyzer for Othello/Reversi. Code it up along with alpha–beta pruning.

3.6. **Term Paper.** Deep Blue and other Computer Champions