

# 1 Game Playing, Cont'd

- Back to alpha-beta search. Last time I gave the intuition about what it's doing. Basically, it's eliminating nodes that it *knows* will never be reached.
- Now we'll spend a little while looking at the actual algorithm and tracing the necessary bookkeeping. (See overhead.)
- So alpha-beta can drastically reduce the size of the search tree. But it *still* has to search all the way to the leaves. That's problematic, as it can be a *long* way down to a leaf (or branching factor can render it implausible).
- So we suggested truncating search.
- Truncation turns internal nodes into leaf nodes.
- To do that, you need an evaluation function that tells you what the "probable" result of a non-leaf node is (i.e., if you go down sub-tree, what will be the likely outcome)?
- One way to do that: store a scalar var with each node; play bazillions of games from that node and store the average outcome at the node. Clearly, this is intractable.
- Problem is that each node has a different param/set of params. But each indiv node is encountered only very rarely. I.e., except for root and first few ply, you may only encounter indiv states only in one or two games. Can never get enough statistics to know much about indiv nodes.
- Instead, want a small, quick function that can look at board state and give back some reasonable value. Usually hand tuned.
- E.g., in chess, the difference in number of pieces (pretty simplistic), or values of pieces (e.g., 1 for pawn, 3 for knight, 5 for rook/bishop, 9 for queen).
- Or, could score according to position. E.g., assign each square a value and take the sum of squares you occupy minus sum of squares your opponent occupies.
- Usually, rather than writing down single big, hairy function, you write down a bunch of small *feature* functions,  $f_1(s), f_2(s), \dots, f_k(s)$  and then your total board evaluation is:

$$V(s) = \sum_{i=1}^k w_i f_i(s) = W^T F(s)$$

(N.b.: this is the heuristic value, not the true value that we defined last time.)

- Note that  $W$  is independent of  $s$ , so that *does* apply in every game. I.e., every game you play gives info about  $W$ .
- $W$  basically tells you which features are more important than other features.

- Q: how do you pick  $W$ ? I.e., if you win/lose a game, how do you use that info to change  $W$ ? Or, alternately, how do you change  $W$  as you go along?
- Problem is that you want to reinforce  $W$  when it's accurate – when it actually prioritizes important features – but penalize it when it's not.
- Note that all the *real* feedback you have is the final value. You know how the game ended up.
- Problem is that don't necessarily know what moves were good and which bad (could play first half of game brilliantly and then throw it away in 2nd half).
- One approach is to assume that estimates closer to the end of the game are better than earlier ones, and adjust earlier ones as new evidence comes in.
- E.g., if you move from  $s$  to  $s'$ , your board evaluation changes from  $V(s)$  to  $V(s')$ . If  $V(s')$  is more accurate than  $V(s)$ , then we say that  $V(s)$  has an error of  $V(s') - V(s)$ , and we should adjust  $W$  to minimize that error:

$$W_{t+1} = W_t + \alpha(V(s') - V(s))F(s)$$

- This will adjust  $W$  incrementally, in small steps, as the game is played. Ultimately, will ground out to *real* board val (i.e., win or loss). To ensure convergence, you have to slowly decrease  $\alpha$  over time.
- Also possible to back up more than one step at a time. Usually use an exponential decay parameter to control how far the backups go back.
- Q: what does the linear decomposition say about relationship between diff features?
- In general, features are entangled in complex, non-linear ways. E.g., a queen is more important when she has free movement than when she's pinned down.
- Also, different features may matter more at different times of the game (e.g., early, mid, or endgame)
- Real game playing programs these days usually use nonlinear combinations.
- What if you don't even have a good idea what the features should be?
- Well, you can just learn a board evaluation *directly*. I.e., learn some  $f : S \rightarrow \mathbb{R}$ .
- Q: is this *classification* or *regression*?
- E.g., can use something like a neural net as a function evaluator.
- Q: this is more *general* than hand-tuning features, but is it *better*?
- Another thing you have to specify is when to stop raw search and truncate tree – when to apply eval func.

- Simplest: time cutoff.
- Slightly smarter: iterative deepening w/ time cutoff.
- Better: determine when further search is fruitless and when to stop descending.
- Problem: horizon effect.
- Quiescence.