Real-world use of alpha-beta

- (Regular) minimax is normally run as a preprocessing step to find the optimal move from every possible situation.
- Minimax with alpha-beta can be run as a preprocessing step, but might have to re-run during play if a non-optimal move is chosen.
- Save states somewhere so if we re-encounter them, we don't have to recalculate everything.

Real-world use of alpha-beta

- States get repeated in the game tree because of *transpositions*.
- When you discover a best move in minimax or alpha-beta, save it in a lookup table (probably a hash table).
  - Called a *transposition table*.

Real-world use of alpha-beta

- In the real-world, alpha-beta does not "pre-generate" the game tree.
  - The whole point of alpha-beta is to not have to generate all the nodes.
- The DFS part of minimax/alpha-beta is what generates the tree.

Improving on alpha-beta

- Alpha-beta still has to search down to terminal nodes sometimes.
  - (and minimax has to search to terminal nodes all the time!)
- Improvement idea: can we get away with only looking a few moves ahead?

Heuristic minimax algorithm

h-minimax(s, d) =

heuristic-eval(s)	if cutoff(s, d)
$\max_{a \text{ in actions(s)}}$ h-minimax(result(s, *a*), d+1)	if player(s)=MAX
$\min_{a \text{ in actions(s)}}$ h-minimax(result(s, *a*), d+1)	if player(s)=MIN

result(s, a) means the new state generated
by taking action *a* in state *s*.

cutoff(s, d) is a boolean test that determines whether
we should stop the search and evaluate our position.

How to create a good evaluation function?

- Trying to judge the probability of winning from a given state.
- Typically use features: simple characteristics of the game that correlate well with the probability of winning.

One last point