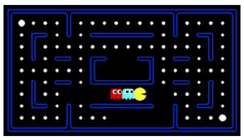
Games: alpha-beta pruning



Pruning principle

Choose A or B with maximum value:

A: [3, **5**]   B: [**5**, 100]

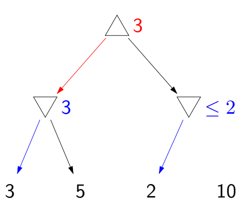
Key idea: branch and bound

Maintain lower and upper bounds on values.

If intervals don't overlap non-trivially, then can choose optimally without further work.

- We continue on our quest to make minimax run faster based on **pruning**. Unlike evaluation functions, these are general purpose and have theoretical guarantees.
- The core idea of pruning is based on the branch and bound principle. As we are searching (branching), we keep lower and upper bounds on each value we're trying to compute. If we ever get into a situation where we are choosing between two options A and B whose intervals don't overlap or just meet at a single point (in other words, they do not **overlap non-trivially**), then we can choose the interval containing larger values (B in the example). The significance of this observation is that we don't have to do extra work to figure out the precise value of A.

Pruning game trees



Once see 2, we know that value of right node must be ≤ 2

Root computes $\max(3, \leq 2) = 3$

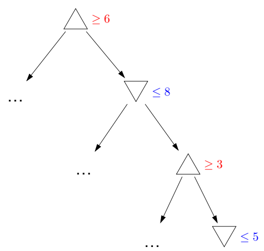
Since branch doesn't affect root value, can safely prune

- In the context of minimax search, we note that the root node is a max over its children.
- Once we see the left child, we know that the root value must be at least 3.
- Once we get the 2 on the right, we know the right child has to be at most 2.
- Since those two intervals are non-overlapping, we can prune the rest of the right subtree and not explore it.

Alpha-beta pruning

Key idea: optimal path

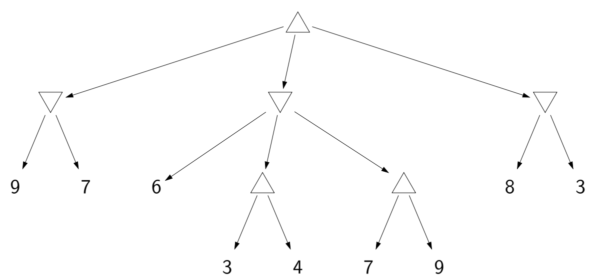
The optimal path is path that minimax policies take. Values of all nodes on path are the same.



- a_s: lower bound on value of max node s
- b_s: upper bound on value of min node s
- Prune a node if its interval doesn't have non-trivial overlap with every ancestor (store $\alpha_s = \max_{s' \preceq s} a_{s'}$ and $\beta_s = \min_{s' \preceq s} b_{s'}$)

- In general, let's think about the minimax values in the game tree. The value of a node is equal to the utility of at least one of its leaf nodes (because all the values are just propagated from the leaves with min and max applied to them). Call the first path (ordering by children left-to-right) that leads to the first such leaf node the **optimal path**. An important observation is that the values of all nodes on the optimal path are the same (equal to the minimax value of the root).
- Since we are interested in computing the value of the root node, if we can certify that a node is not on the optimal path, then we can prune it and its subtree.
- To do this, during the depth-first exhaustive search of the game tree, we think about maintaining a lower bound ($\geq a_s$) for all the max nodes s and an upper bound ($\leq b_s$) for all the min nodes s.
- If the interval of the current node does not non-trivially overlap the interval of every one of its ancestors, then we can prune the current node. In the example, we've determined the root's node must be ≥ 6. Once we get to the node on at ply 4 and determine that node is ≤ 5, we can prune the rest of its children since it is impossible that this node will be on the optimal path (≤ 5 and ≥ 6 are incompatible). Remember that all the nodes on the optimal path have the same value.
- Implementation note: for each max node s, rather than keeping a_s, we keep α_s, which is the maximum value of $a_{s'}$ over s and all its max node ancestors. Similarly, for each min node s, rather than keeping b_s, we keep β_s, which is the minimum value of $b_{s'}$ over s and all its min node ancestors. That way, at any given node, we can check interval overlap in constant time regardless of how deep we are in the tree.

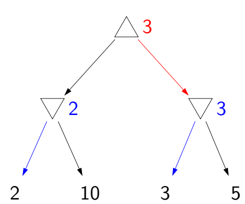
Alpha-beta pruning example



Move ordering

Pruning depends on order of actions.

Can't prune the 5 node:



- We have so far shown that alpha-beta pruning correctly computes the minimax value at the root, and seems to save some work by pruning subtrees. But how much of a savings do we get?
- The answer is that it depends on the order in which we explore the children. This simple example shows that with one ordering, we can prune the final leaf, but in the second, we can't.

Move ordering

Which ordering to choose?

- Worst ordering: $O(b^{2\cdot d})$ time
- Best ordering: $O(b^{2\cdot 0.5d})$ time
- Random ordering: $O(b^{2\cdot 0.75d})$ time when $b = 2$
- Random ordering: $O((\frac{b-1+\sqrt{b^2+14b+1}}{4})^{2\cdot d})$ for general b

In practice, can use evaluation function Eval(s):

- Max nodes: order successors by decreasing Eval(s)
- Min nodes: order successors by increasing Eval(s)

- In the worst case, we don't get any savings.
- If we use the best possible ordering, then we save half the exponent, which is *significant*. This means that if could search to depth 10 before, we can now search to depth 20, which is truly remarkable given that the time increases exponentially with the depth.
- In practice, of course we don't know the best ordering. But interestingly, if we just use a random ordering, that allows us to search 33 percent deeper.
- We could also use a heuristic ordering based on a simple evaluation function. Intuitively, we want to search children that are going to give us the largest lower bound for max nodes and the smallest upper bound for min nodes.