CS312 Recitation 21
Minimax search and Alpha-Beta Pruning

A game can be thought of as a tree of possible future game states. For example, in Gomoku the game state is the arrangement of the board, plus information about whose move it is. The current state of the game is the root of the tree (drawn at the top). In general this node has several children, representing all of the possible moves that we could make. Each of those nodes has children representing the game state after each of the opponent's moves. These nodes have children corresponding to the possible second moves of the current player, and so on. The leaves of the tree are final states of the game: states where no further move can be made because one player has won, or perhaps the game is a tie. Actually, in general the tree is a graph, because there may be more than one way to get to a particular state. In some games (e.g., checkers) it is even possible to revisit a prior game state.

Minimax search

Suppose that we assign a value of positive infinity to a leaf state in which we win, negative infinity to states in which the opponent wins, and zero to tie states. We define a function **evaluate** that can be applied to a leaf state to determine which of these values is correct.

If we can traverse the entire game tree, we can figure out whether the game is a win for the current player assuming perfect play: we assign a value to the current game state by we recursively walking the tree. At leaf nodes we return the appropriate values. At nodes where we get to move, we take the max of the child values because we want to pick the best move; at nodes where the opponent moves we take the min of child values. This gives us the following pseudo-code procedure for **minimax** evaluation of a game tree.

```
fun minimax(n: node): int =
   if leaf(n) then return evaluate(n)
   if n is a max node
      v := L
      for each child of n
         v' := minimax (child)
         if v' > v, v:= v'
      return v
   if n is a min node
      v := W
      for each child of n
         v' := minimax (child)
         if v' < v, v:= v'
      return v
```

Consider the following game tree, where the leaves are annotated with W or L to indicate a winning or losing position for the current player (L < W), and interior nodes are labeled with + or - to indicate whether they are "max" nodes where we move o "min" nodes where the opponent moves. In this game tree, the position at the root of the tree is a losing position because the opponent can force the game to proceed to an "L" node:

```
       +
      / \
     /   \
    -     -
   / \   /|\
  +   L L W W
 / \
W   L
```

We can see this by doing a minimax evaluation of all the nodes in the tree. Each node is labeled with its minimax value in red:

```
       L
      / \
     /   \
    L     L
   / \   /|\
  W   L L W W
 / \
W   L
```

Static evaluation

Usually expanding the entire game tree is infeasible because there are so many possible states. The solution is to only search the tree to a specified depth. The `evaluate` function (the **static evaluator**) is extended so it returns a value between L and W for game positions that are not final positions. For game positions that look better for the current player it returns larger numbers. When the depth limit of the search is exceeded, the static evaluator is applied to the node as if it were a leaf

```
(* the minimax value of n, searched to depth d *)
 fun minimax(n: node, d: int): int =
   if leaf(n) or depth=0 return evaluate(n)
   if n is a max node
      v := L
      for each child of n
         v' := minimax (child,d-1)
         if v' > v, v:= v'
      return v
   if n is a min node
      v := W
      for each child of n
         v' := minimax (child,d-1)
         if v' < v, v:= v'
      return v
```

For example, consider the following game tree searched to depth 3, where the static evaluator is applied to a number of nodes that are not leaves in the game tree:

```
        6
       / \
      /   \
    6     L
   / \    |\
  6   8   L W
 /|\  |\    |\
1 5 6 8 3   1 W
```

The value of the root of the tree is 6 because the current player can force the game to go to a "leaf" node (as defined by the depth cutoff) whose value is at least 6. Notice that by finding out the value of the current position, the player also learns what is the best move to make: the move that transitions the game to the immediate child with maximum value.

Designing the static evaluator is an art: a good static evaluator should be very fast, because it is the limiting factor in how quickly the search algorithm runs. But it also needs to capture a reasonable approximation of how good the current board position is, because it captures what the player is trying to achieve during play. In practice, game AI designers have found that it doesn't pay to build intelligence into the static evaluator when the same information can be obtained by searching a level or two deeper in the game tree.

How deeply should the tree be searched? This depends on how much time is available to do the search. Each increase in depth multiplies the total search time by about the number of moves available at each level.

Alpha-Beta Pruning

The full minimax search explores some parts of the tree it doesn't have to. For example, consider the tree above and suppos that our search is proceeding in left-to-right order

```
        6
       / \
      /
    6 (a)
   / \
  6   8 (b)
 /|\  |\
1 5 6 8 ...
```

Once we have seen the node whose static evaluation is 8, we know that there is no point to exploring any of the rest of the children of the max node above it. Those children could only increase the value of the max node (b) above, but the min node above that (a) is going to have value at most 6 anyway. No matter what happens in the part of the tree under the `...`, it can't affect the minimax value of the min node labeled 6. Avoiding searching a part of a tree is called **pruning**; this is an example of **alpha-beta pruning**.

In general the minimax value of a node is going to be worth computing only if it lies within a particular range of values. We can capture this by extending the code of the minimax function with a pair of arguments `min` and `max`. The new spec of `minimax` is that it always returns a value in the range [`min`, `max`]. For example, when evaluating the node (b) above, we can set `max` to 6 because there is no reason to find out about values greater than 6. There are corresponding cases where there is no reason to find out about values less than some minimum value. The `min` and `max` bounds are used to prune away subtrees by terminating a call to `search` early. Once a child node has been seen that pushes the node's value outside the range of interest, there is no point in exploring the rest of the children. This idea is captured by adding the tests `if v > max return max` and `if v < min return min` in the following code:

```
(* the minimax value of n, searched to depth d.
 * If the value is less than min, returns min.
 * If greater than max, returns max. *)
 fun minimax(n: node, d: int, min: int, max: int): int =
   if leaf(n) or depth=0 return evaluate(n)
   if n is a max node
      v := min
      for each child of n
         v' := minimax (child,d-1,...,...)
         if v' > v, v:= v'
         if v > max return max
      return v
   if n is a min node
      v := max
      for each child of n
         v' := minimax (child,d-1,...,...)
         if v' < v, v:= v'
         if v < min return min
      return v
```

Because we don't care about values less than `min` or greater than `max`, we also initialize the variable `v` to `min` in the max case and `max` in the min case, rather than to `L` and `W`. Notice that if this procedure is invoked as `minimax(n,d,L,W)`, it will behave just like the `minimax` procedure without `min` and `max` bounds, assuming that the static evaluator only returns values between `L` and `W`. Thus, a top-level search is invoked in this way so that we get the same answer as before pruning.

The only thing missing from our search algorithm now is to compute the right `min` and `max` values to pass down. Clearly we could safely pass down the same `min` and `max` received in the call, but then we wouldn't have achieved anything. Consider the max node case after we have gone around the loop. In general the variable `v` will be greater than `min`. In the recursive invocation of `minimax` there is no point to finding out about values less or equal to `v`; they can't possibly affect the value of `v` that is returned. Therefore, instead of passing `min` down in the recursive call, we pass `v` itself. Conversely, in the min node case, we pass `v` in place of `max`:

```
(* the minimax value of n, searched to depth d.
 * If the value is less than min, returns min.
 * If greater than max, returns max. *)
 fun minimax(n: node, d: int, min: int, max: int): int =
   if leaf(n) or depth=0 return evaluate(n)
   if n is a max node
      v := min
      for each child of n
         v' := minimax (child,d-1, v,max)
         if v' > v, v:= v'
         if v > max return max
      return v
   if n is a min node
      v := max
      for each child of n
         v' := minimax (child,d-1, min,v)
         if v' < v, v:= v'
         if v < min return min
      return v
```

This is pseudo-code for minimax search with alpha-beta pruning, or simply **alpha-beta search**. We can verify that it works as intended by checking what it does on the example tree above. Each node is shown with the `[min,max]` range that minimax is invoked with. Pruned parts of the tree are marked with X.

```
                      6[L,W]
                     /       \
                    /         \
                   /           \
             6[L,W]             L[6,W]
            /       \            |    \
           /         \           |     X
     6[L,W]           6[L,6]     L[6,W]
    /   |   \         |   \
   /    |    \        |    X
  /     |     \       |
1[L,W] 5[1,W] 6[5,W] 8[L,6]
```

In general the [min,max] bounds become tighter and tighter as you proceed down the tree from the root.

Making pruning effective

How effective is alpha-beta pruning? It depends on the order in which children are visited. If children of a node are visited in the worst possible order, it may be that no pruning occurs. For max nodes, we want to visit the best child first so that time is not wasted in the rest of the children exploring worse scenarios. For min nodes, we want to visit the worst child first (from our perspective, not the opponent's.) There are two obvious sources of this information:

1. The static evaluator function can be used to rank the child nodes
2. Previous searches of the game tree (for example, from previous moves) performed minimax evaluations of many game positions. If available, these values may be used to rank the nodes.

When the optimal child is selected at every opportunity, alpha-beta pruning causes all the rest of the children to be pruned away at every other level of the tree; only that one child is explored. This means that on average the tree can searched twice as deeply as before—a huge increase in searching performance.