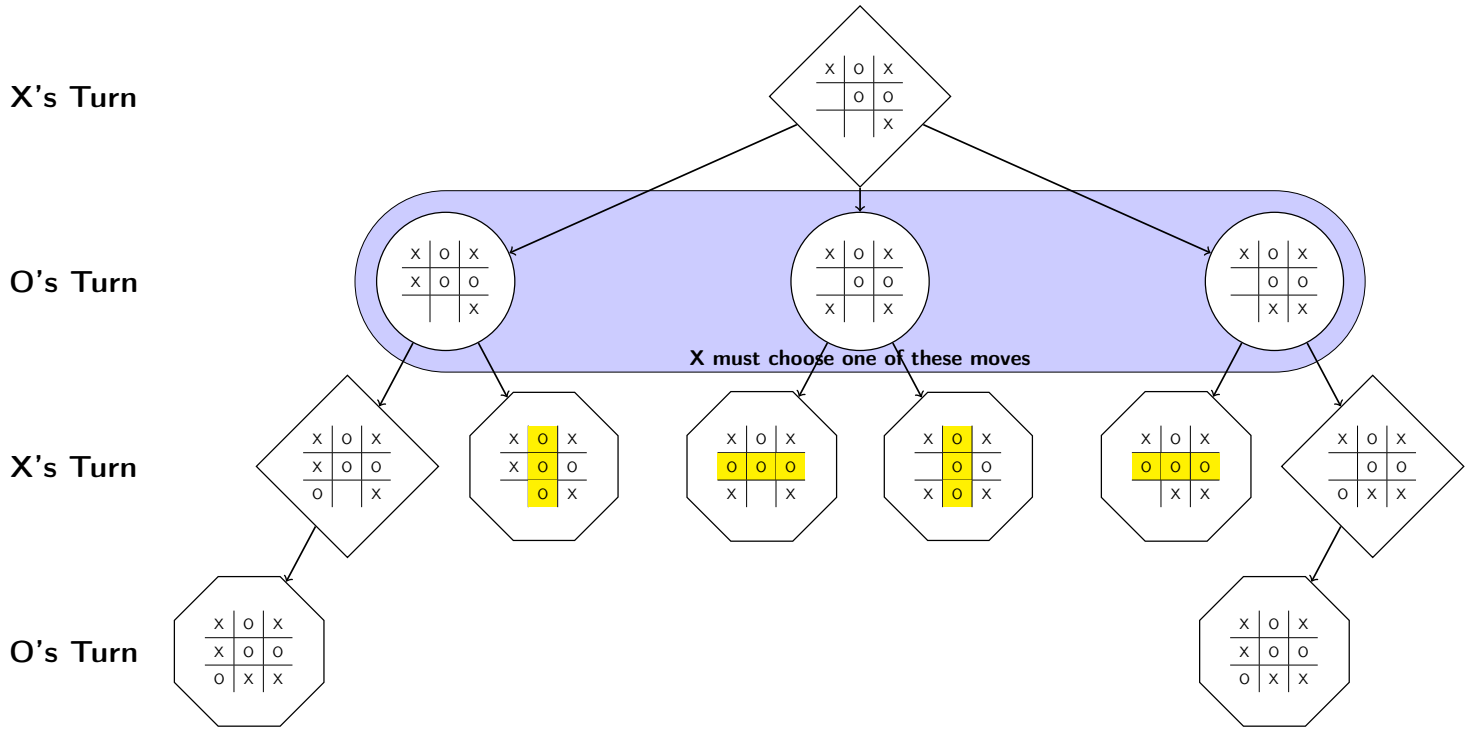
CSE 332: Data Structures and Parallelism

Games, Minimax, and Alpha-Beta Pruning

This handout describes the most essential algorithms for game-playing computers. NOTE: These are only partial algorithms: you will need to work out the details when doing P3.

Playing Games

To play a game of Tic-Tac-Toe, two players (X and O) alternate making moves. The first player to get three of their letter in a row wins. Usually, the board starts empty, but in the interest of a reasonable example, we'll look at a partially played game instead:



We make a few observations about the game above:

- If X and O are both playing optimally, O will win. (Why?)
- The *leaves* of the tree are "terminal positions" of the game.
- Moves alternate between the two players.

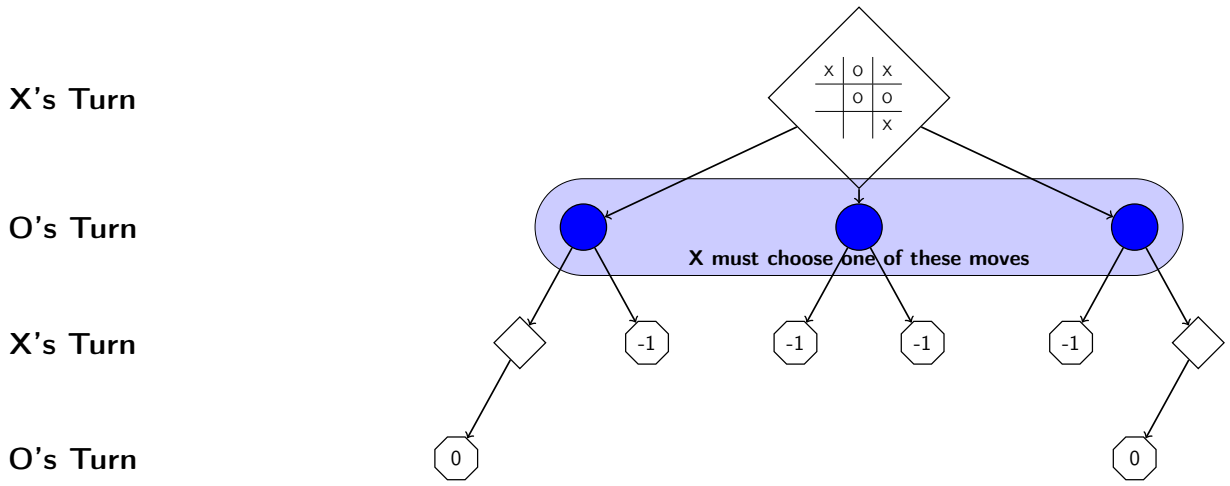
This diagram is called a *game tree* and it's generated by starting at a move and recursively generating all the possible moves that could be made until the game ends. Putting this idea into pseudocode, we have:

```
1  void printTerminalPositions(Position p) {
2     if (p is a leaf) {
3        print p
4     }
5     else {
6        for (move in p.getMoves()) {
7           p.applyMove(move);
8           printTerminalPositions(move);
9           p.undoMove();
10       }
11    }
12 }
```

Notice that this is a *recursive backtracking* algorithm.

The definitions of `getMoves`, `applyMove`, and `undoMove` depend on the game that we're playing. For example, in Tic-Tac-Toe, `getMoves` returns a list of all the valid X moves (or O moves, depending on the player's turn).

In a two player game (like Tic-Tac-Toe), there are three possible outcomes: (I win, I lose, We draw) Because *every* leaf must be one of these options, we can give them numerical values to evaluate how good they are. Since there are only these three, we use $+1$ for win, 0 for draw, and -1 for lose. Importantly, these values are *the only thing about the position* that we actually care about! If we know a move is a $+1$, it doesn't matter what exactly the series of moves we made is. So, taking this into account, we re-draw our game tree (from X's perspective):



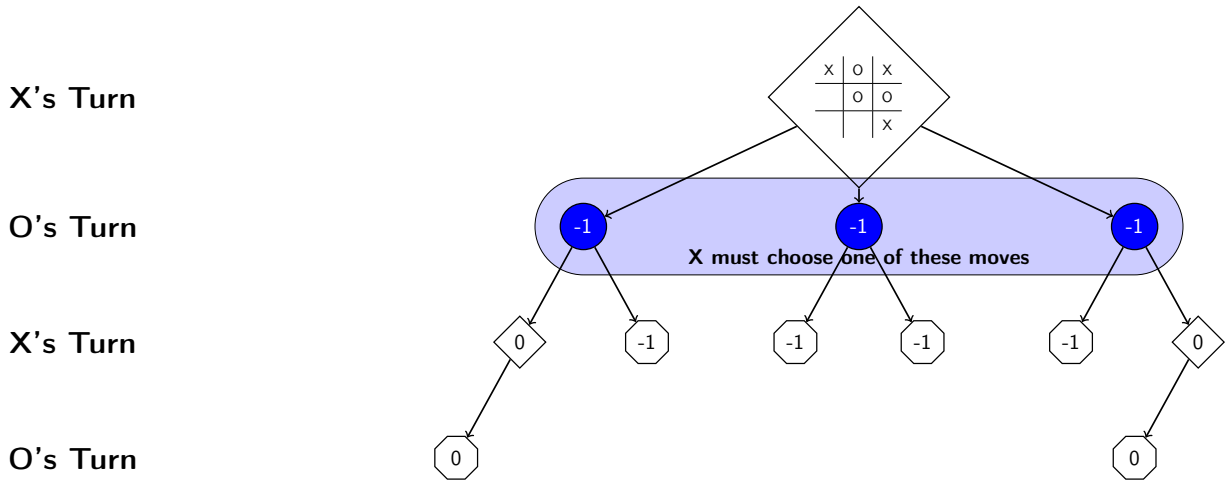
Now, to figure out which move to make, all we have to do is *figure out the values of the blue moves*. To do this, we make a major assumption:

Our opponent will make the best possible move they can.

Intuitively, if we give our opponent the benefit of the doubt, then we can't be surprised by any move they make. The best possible move for our opponent is the *worst* possible move for us. To figure out the values of the blue moves, we *recursively* figure out the values of the moves below them in the game tree. There are two cases:

- If it's our turn, then we'll take the *best possible move we can*. In other words, we take the maximum value of the children's values.
- If it's our opponent's turn, then they will give us the *worst possible move they can*. In other words, we take the minimum value of the children's values.

So, on the lines labeled "X's Turn", we take the maximum of the moves below, and on the lines labeled "O's Turn", we take the minimum of the moves below. The filled in game tree looks like this (from X's perspective):



Unfortunately for us, since all of the choices we have are "-1", it means no matter what we do, a perfect opponent can always force us to lose this game of Tic-Tac-Toe. If we follow the -1's down the game tree, we can see the moves in every case that make us lose.

Minimax

The idea we just used to fill in the Tic-Tac-Toe board is a general one called *Minimax*. First, we describe the general algorithm, and then we get into some important changes that must be implemented in practice.

The Algorithm

```
1  int minimax(Position p) {
2     if (p is a leaf) {
3        // evaluate tells us the
4        // value of the current
5        // position
6        return p.evaluate();
7     }
8
9     int bestValue = −∞;
10    for (move in p.getMoves()) {
11       p.applyMove(move);
12       int value = −minimax(p);
13       p.undoMove();
14       if (value > bestValue) {
15          bestValue = value;
16       }
17    }
18 }
```

This really is the same algorithm that we describe on the previous page. Notice the $-$ in front of the recursive call. This is because the move after us is our opponent who is attempting to do the *opposite* thing from us. Mathematically, this works because

$$\max(a, b) = -\min(-a, -b)$$

When writing a bot to play a game, we'd also need to keep track of the *actual move* corresponding to the best score. This involves a small addition to the if statement where we update the best score. Notice that we're only interested in the *very next move*. We're using the future move to help us *understand* the next move better.

The version of the algorithm we've described here is usually called *negamax*, because it uses this property to reduce code redundancy.

Using Minimax in a Real Game

Since our goal is to ultimately implement a chess bot, let's do some back-of-the-hand calculations on a chess game. The *branching factor* of a tree is the number of children a node has. Since some positions in chess have more moves than others, we work with the *average branching factor* instead. It turns out in chess, the average branching factor is approximately **35**. The average chess game lasts approximately 40 moves. Putting these numbers together, we would need to evaluate at least $35^{40} \approx 5.8 \times 10^{61}$ leaves to use this method in a real chess game. If we were able to evaluate **1 trillion** leaves per second, it would take at least 10^{48} seconds (which is more than 10^{30} times the number of seconds the universe has existed). This is clearly infeasible.

So, in the real world, instead of evaluating all the way down to the leaves, we *estimate* the leaves by going several moves ahead. Although this is less accurate, it's the best we can do. The only change this makes to the code is to add a second parameter `depth` and change our base case to `depth == 0` in addition to checking for a leaf.

Unfortunately, this also makes our evaluation function more complicated, because we must estimate how good a position is without knowing if it actually leads to a win.
A natural question to ask is "how many levels ahead can we look?" (we call these *ply*). You will determine this yourself experimentally on the homework, but the best chess bots in the world can look about 20 ply ahead; you should expect your bot to be able to do a few less than half of that.

To review, in the real world...

- We only look a few moves ahead instead of going to the end of the game
- The evaluation function takes on a much larger range of numbers than just -1, 0, and 1, because we're less sure of the value of the position.

In p3, you will be provided with a reasonable evaluation function. You may edit it if you like, but it's not required. Your bot will be given three minutes for each game (and it will gain two seconds every time it makes a move). This is much less time than it sounds like it is.

Parallel Minimax

The Algorithm

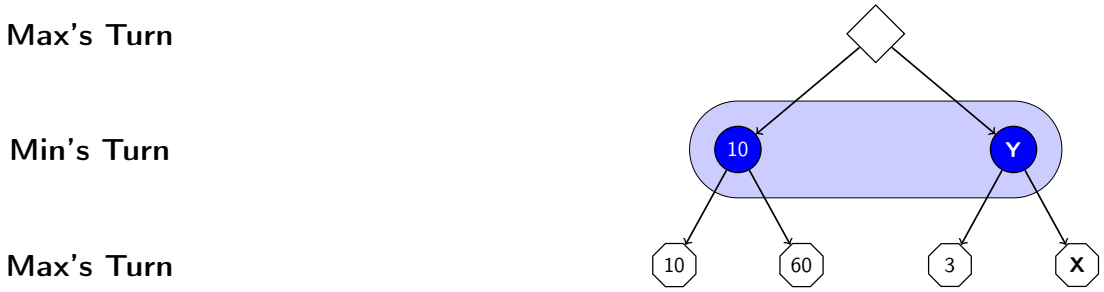
```
1  int minimax(Position p) {
2     if (p is a leaf) {
3        return p.evaluate();
4     }
5
6     int bestValue = −∞;
7     parallel (move in p.getMoves()) {
8        p = p.copy();
9        int value = −minimax(p);
10       if (value > bestValue) {
11          bestValue = value;
12       }
13    }
14 }
```

Minimax is a *naturally parallelizable* algorithm. Each "node" of the game tree can be run on independent threads. Even though the algorithm is very similar there are a couple of gotchas:

- Since different threads will be working at the same time, they can't share one position. This means you'll need to copy the position for each thread.
- As always, you'll want to have a cutoff. Your cutoff should be in terms of the depth remaining of the tree.
- Make sure you use divide and conquer to get the threads running as quickly as possible.

Alpha-Beta Pruning

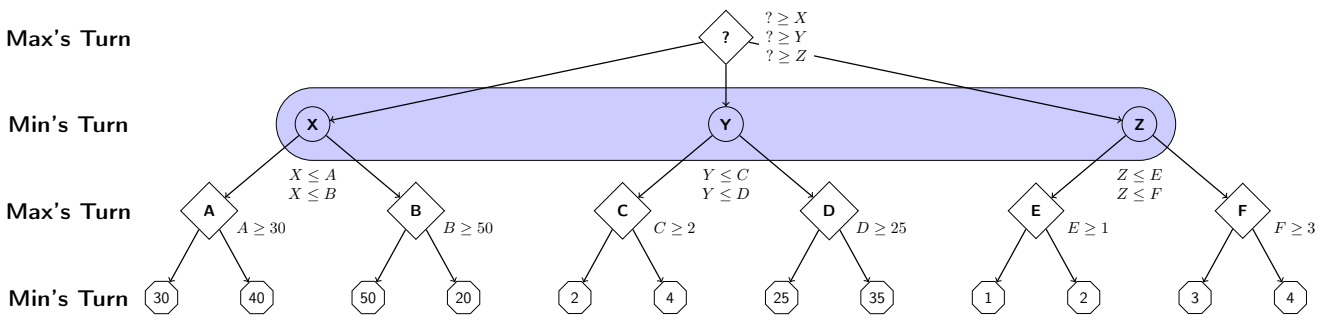
Alpha-beta Pruning is a more efficient version of Minimax that avoids considering branches of the game tree that are irrelevant. Before getting too deep into the algorithm, it is very important to note that a correct **Alpha-beta Search will return the same answer as Minimax**. In other words, it is *not* an approximation algorithm, it *only* ignores moves that cannot change the answer. What might such a move look like? Consider the following:



Suppose that we've gone through most of the game tree and evaluated the first three leaves. The question that remains is "is it possible that Y is a better move than the 10?"" If it is, then we have to evaluate **X**; otherwise, we don't have to waste the time. It turns out to not matter, here's why:

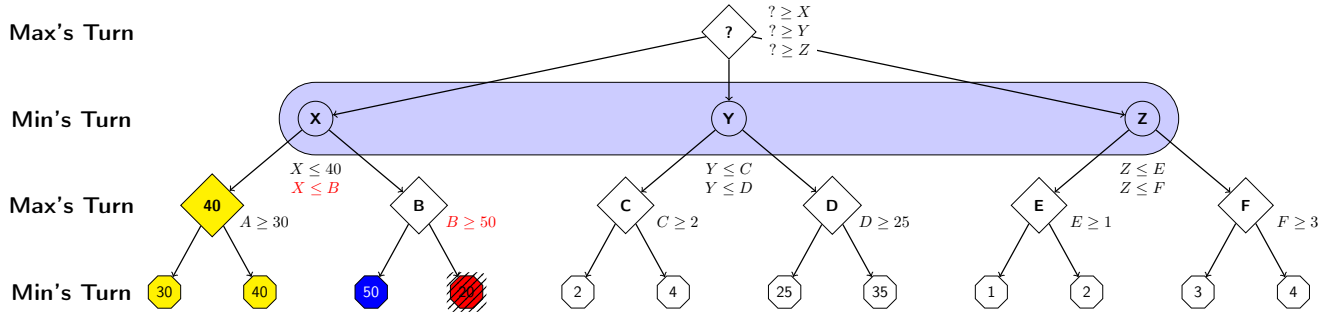
- If $\mathbf{X} \geq 3$, then Min would choose the 3; so, $\mathbf{Y} = 3$. But, this is less than the 10 we can already get.
- If $\mathbf{X} < 3$, then Min would choose **X**; so, $\mathbf{Y} < 3$. But, this is less than the 10 we can already get.

More succinctly, because $Y = \min(3, X)$, we know that $Y \leq 3$ which is less than another move we already found. It follows that we can ignore this last value. This sort of "bounding" argument can be very powerful. Let's consider another game tree where we write down all of the bounds as we go:

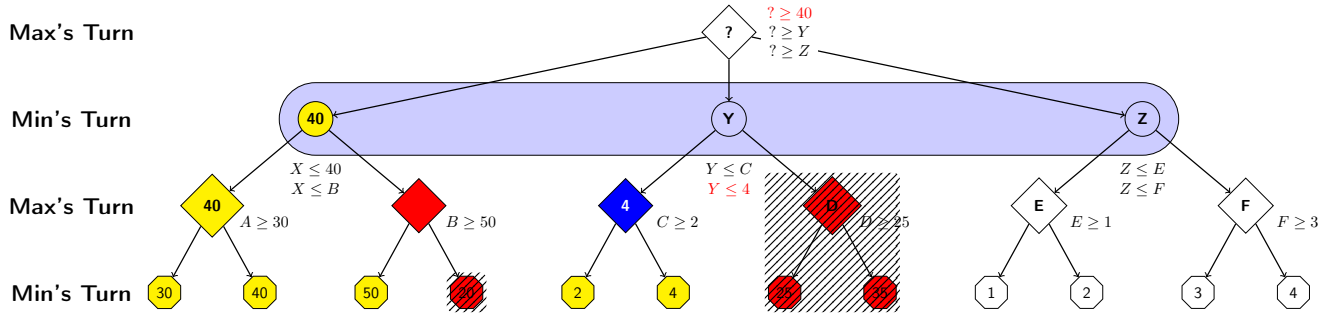


The idea is that as we fill in these values, if we find one that contradicts a bound, we can stop looking in that subtree. **Before looking at the next page, try to figure out which leaves we don't need to evaluate.**

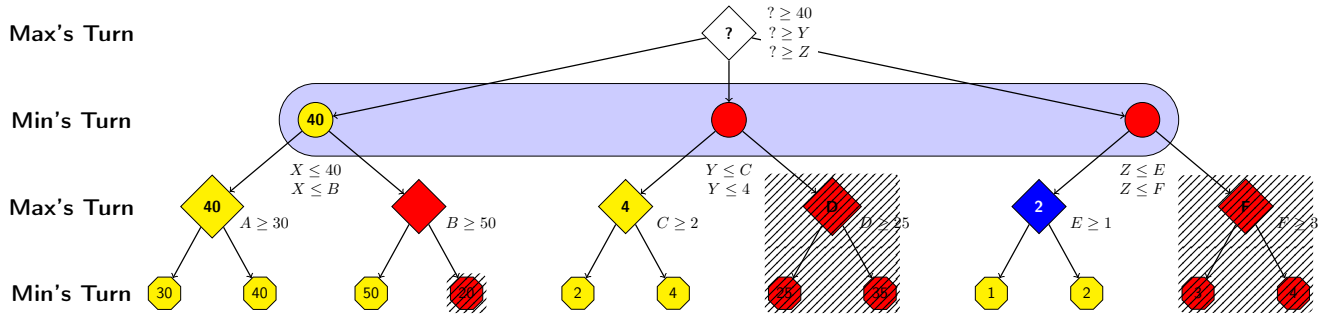
We evaluate 30, 40, and 50; then, we notice that $X \leq 40$, but we hit a 50 which violates the condition. So, we cut off the rest of that subtree.



We evaluate 2 then 4; then, we notice that min can force a 4 if we choose **Y**. So, we can cut off the rest of that subtree.

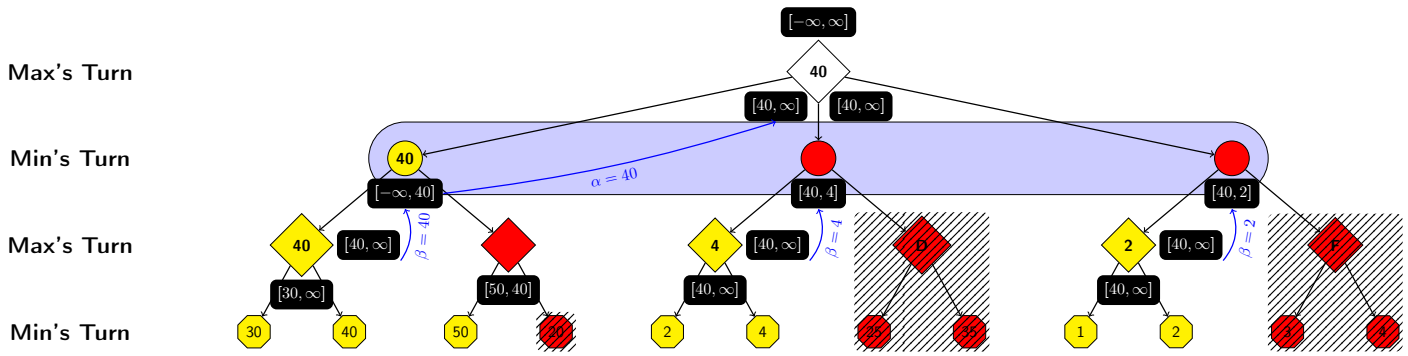


Finally, we evaluate 1 and 2 and notice that that gives us a cutoff for the remaining subtree.



Notice, again, that, if we were to evaluate the whole tree (via minimax), we would get the same answer. Furthermore, we are able to make these cutoffs both as the min player *and* the max player.

In code, a cleaner way of dealing with the inequalities is as a *valid range*. The Min player makes the upper bound smaller and the Max player makes the lower bound bigger. Alpha-beta pruning gets its name from this idea: we call the lower bound α and the upper bound β and we provide them as arguments. Whenever $\beta \leq \alpha$, we cut off. Notice how nodes on max levels *only propagate up* β and nodes on min levels *only propagate up* α:



Finally, we can describe the actual algorithm.

The Algorithm

```
1  int alphabeta(Position p, int alpha, int beta) {
2     if (p is a leaf) {
3        return p.evaluate();
4     }
5
6     for (move in p.getMoves()) {
7        p.applyMove(move);
8        int value = -alphabeta(p, -beta, -alpha);
9        p.undoMove();
10
11       // If value is between alpha and beta, we've
12       // found a new lower bound
13       if (value > alpha) {
14          alpha = value;
15       }
16
17       // If the value is bigger than beta, we won't
18       // actually be able to get this move
19       if (alpha >= beta) {
20          return alpha;
21       }
22    }
23
24    // Return the best achievable value
25    return alpha;
26 }
```

Again, we're using the special properties of min and max to make the code cleaner. This time, when we switch from min to max, we swap the upper and lower bounds as well. It's also important to notice that the "best move value" *is alpha*; we're not keeping track of another value in addition to alpha. We strongly recommend running through the algorithm on your own in the above tree before attempting to code it up. Alphabeta is deceptively complicated!

Move Ordering

Because `alphabeta` attempts to prune as many nodes as possible based on *which* nodes it evaluates, the *order* that you visit the moves in matters substantially. The assignment does not require that you do any interesting move ordering, but in both `alphabeta` and `jamboree` (see next section), if you apply move ordering, your performance will be substantially better.

Parallel Alpha-Beta Pruning

After you have `alphabeta` working, you will write a *parallel* version. Unfortunately, unlike `minimax`, `alphabeta` is *not* "naturally parallelizable". In particular, if we attempt to parallelize the loop, we will be unable to propogate the new alpha and beta values to each iteration. This would result in us evaluating unnecessary parts of the tree. In practice, however, it turns out that that this is an acceptable loss, because the parallelism still gives us an overall benefit. So, our general strategy (a variant of an algorithm called `Jamboree`) is the following.

- Evaluate x of the moves sequentially to get reasonable `alpha`/`beta` values that will enable us to cut out large parts of the tree.
- Evaluate the remaining moves in parallel. This means we will evaluate some unnecessary moves, but, in practice, it's worth it.

Then, the algorithm looks something like the following:

The Algorithm

```
1  PERCENTAGE_SEQUENTIAL = 0.5;
2  int jamboree(Position p, int alpha, int beta) {
3     if (p is a leaf) {
4        return p.evaluate();
5     }
6
7     moves = p.getMoves();
8
9     for (i = 0; i < PERCENTAGE_SEQUENTIAL * moves.length; i++) {
10       p.applyMove(moves[i]);
11       int value = -jamboree(p, -beta, -alpha);
12       p.undoMove();
13
14       if (value > alpha) {
15          alpha = value;
16       }
17       if (alpha >= beta) {
18          return alpha;
19       }
20    }
21
22    parallel (i = PERCENTAGE_SEQUENTIAL * moves.length; i < moves.length; i++) {
23       p = p.copy();
24       int value = -jamboree(p, -beta, -alpha);
25
26       if (value > alpha) {
27          alpha = value;
28       }
29       if (alpha >= beta) {
30          return alpha;
31       }
32    }
33
34    return alpha;
35 }
```

This algorithm has a *lot* of important constants to tweak which make a big difference:

- `PERCENTAGE_SEQUENTIAL` can make a big difference. You should play with the value until you find a good one.
- There will also be a sequential cutoff like normal which is *not the same as* `PERCENTAGE_SEQUENTIAL`.
- As with all the other algorithms, you will need to choose a depth to go to. This algorithm *should* get further than any of the others.
- Make sure that your sequential cut-off does *not* recursively call the parallel version. If you accidentally do that, performance will degrade substantially.
- As with the other parallel algorithm, it is important to figure out when you should copy the board vs. just undoing the move.