Game Trees

CSM 61B

February 2018

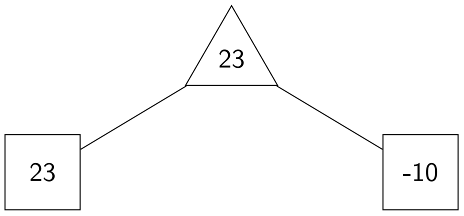
Created by: Vibha Seshadri, Yi Zhao, Sarah Zhou, Anson Tsai, Jose Chavez

1 Introduction

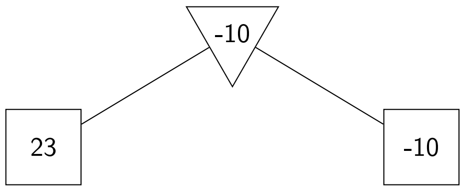
The basis of various "CPU" opponents in video games use game trees to predict and decide moves against you. As the name suggests, it is a recursive tree structure that can store the resulting score of almost every possible move a player can make, and most importantly, the possible moves that can be made against it! In this worksheet, we dive into the mechanics, code, and run-time of two game tree algorithms that find the optimal move for a certain player: Minimax and Alpha-beta pruning.

2 Minimax

So how do we find the optimal move to make, given a current state of the game? We can use a naive algorithm, named Minimax, to solve this problem. Given the current state of the game represented as a node, we first enumerate the states of all possible *future* states of the game, creating a game tree. With the full tree materialized and all the end states scored, we can simulate the moves chosen at each level, representing moves, based on the role of the current player. Recall that the role of the maximizer is to get the highest score possible, while the minimizer is to get the lowest. In terms of a game tree, a maximizing node will take the value of the highest valued child, and a minimizing node will take the value of the lowest valued child. Let's first take a look at Figure 1(a) to solidify our understanding. We see that the root is a maximizer, it will take the left edge, or move, for the larger score. In Figure 1(b), we can see the reverse, as the root is now a minimizer and will take the right move for the lower score.



(a) Maximizing

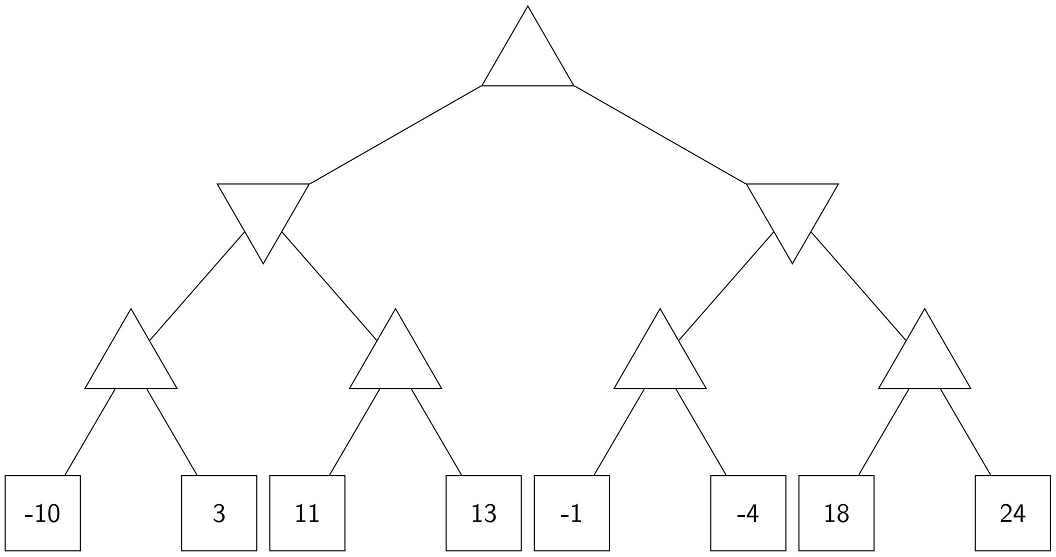


(b) Minimizing

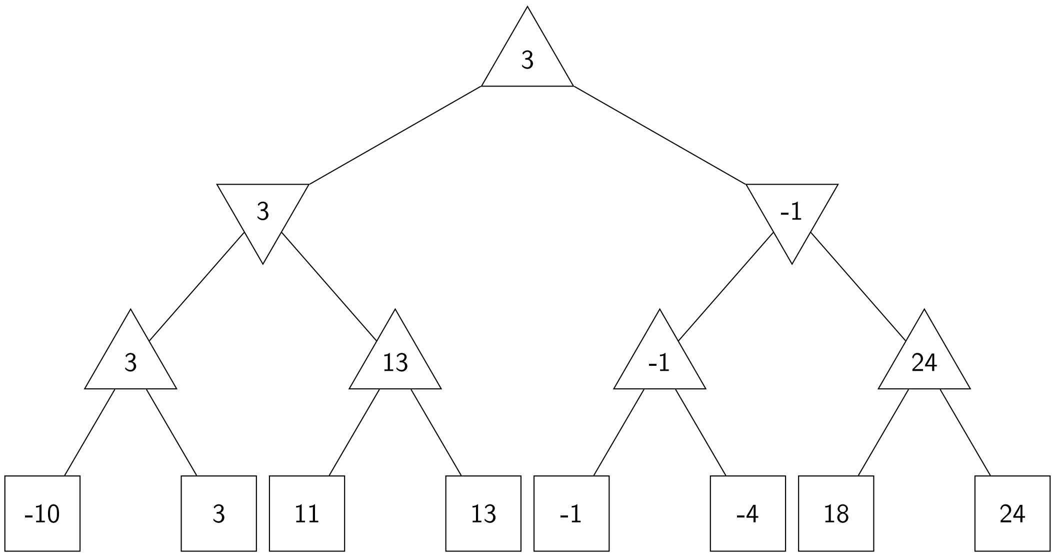
Figure 1: Simple Max and Min

2.1 Example

Now lets tackle a bigger problem. In Figure 2(a), we have fully materialized all possible future game states, including the end states with their respective scores. Note the different types of nodes, with the upwards triangle as the maximizer and the downwards triangle as the minimizer. With this current game tree, which move should our root take? Using the same methods as in Figure 1, we can propagate the scores upwards to get filled-in game tree shown in Figure 2(b). Since the root is a maximizer, it would choose the bigger child, which is the left node with a value of 3.



(a) Before Minimax



(b) After Minimax

Figure 2: Harder Example

If the 4th end state from the left changes from 13 to 100, would the root change its move? No — while its immediate parent will choose the new value (it's a maximizer), the next parent will still choose the left move (it's a minimizer) and our modification will not be propagated upwards.

While this method reveals the optimal choice for us to take, Minimax cannot be used for all situations. This is because game trees tend to be either infinite or impossibly large, which can make Minimax infeasible. For this reason, we will have to optimize this process, which will bring us to Alpha-Beta pruning in the next section.

2.2 Pseudocode

```
public static int minimax(Tree gameTree, boolean isMaxPlayer) {
    if(gameTree.isLeaf()) {
        return gameTree.val(); //Computed heuristic value for our game tree
    }

    if(isMaxPlayer) { //Maximizing player wants to MAX their utility
        int bestVal = Integer.MIN_VALUE;
        for(Node node : gameTree.branches()) {
            bestVal = Math.max(bestVal, minimax(node, false));
        }
        return bestVal;
    } else { //Minimizing player wants to MIN their loss
        int bestVal = Integer.MAX_VALUE;
        for(Node node : gameTree.branches()) {
            bestVal = Math.min(bestVal, minimax(node, true));
        }
        return bestVal;
    }
}
```

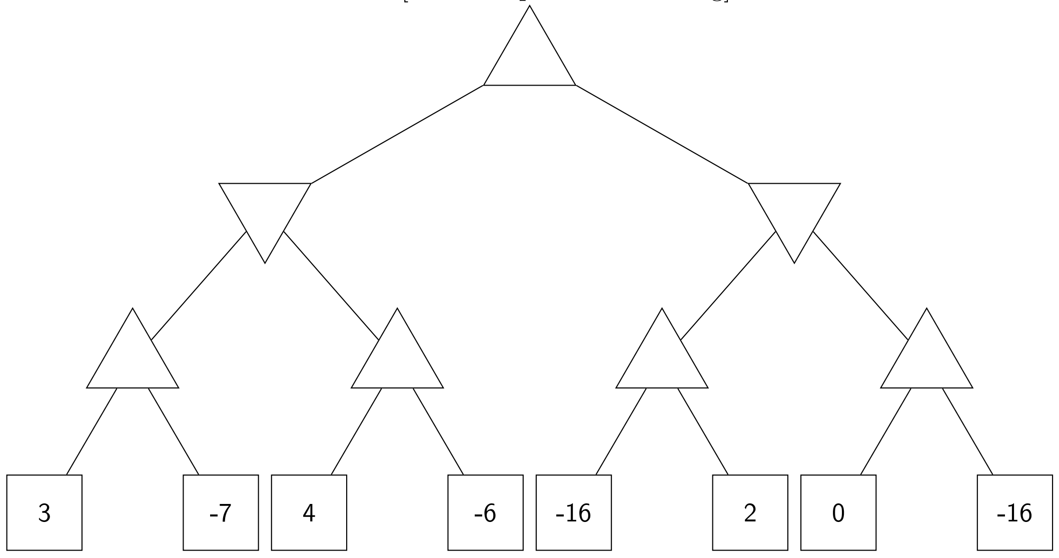
Basic idea is that we want to simulate two optimal players playing. At the leaf nodes, we already have pre-computed ”end game states”. These states indicate the gain of the maximizing player or the loss of the minimizing player. Each step of the algorithm represents a turn taken by a player. The maximizing player goes first, and wishes to play the move (look in all the child branches) for the game state that maximizes his gain. The minimizing player wants to do the opposite, at each move, they look for the game state that minimizes their loss in all their child branches.

3 Alpha-beta Pruning

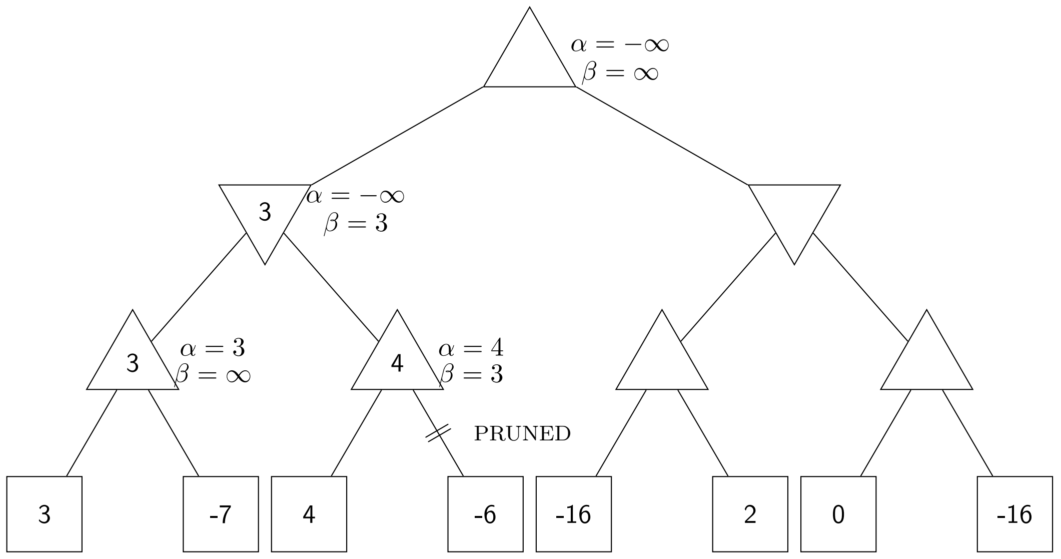
Alpha-beta Pruning is very similar to Minimax but has the possibility of not searching through every possible move at every depth. As the name suggests, there will be two values being kept track of at every move node: α and β. These values are switch depending whether a player is maximizing or minimizing, but the exist to help us determine a cut off to stop searching through a section of a game tree, namely, when $\alpha > \beta$.

3.1 Example

[Before Alpha-beta Pruning]

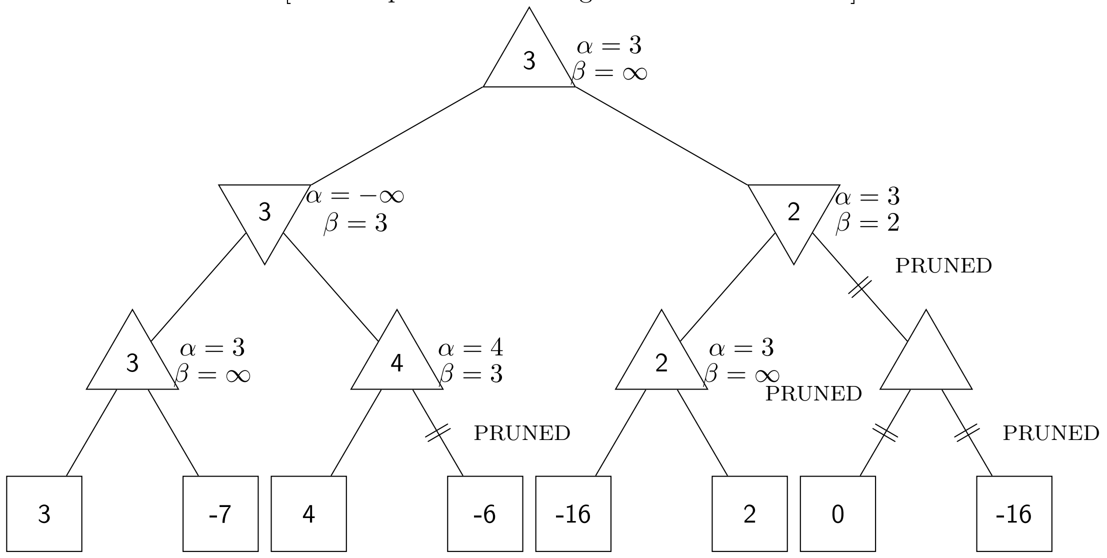


[After alpha-beta pruning finishes on left sub-tree]



Meta: Notice the **left** maximizer's alpha value (the left child of the minimizer) is the minizimer's beta value. On the other hand, the **right** maximizer's beta value (the right child of the minizer) is the minimizer's beta value. When going from child to parent, the alpha or beta value (depending on if the node is maximizer or minimizer) will flip-flop. This is because a minimizer's beta value is the maximizer's alpha value. Going from parent to child, the alpha/beta values are carried down as is.

[After Alpha-beta Pruning finishes on entire tree]



3.2 Pseudocode

```
/*
Alpha beta pruning
*/

public static int alphabeta(Tree gameTree, int alpha, int beta, boolean isMaxPlayer) {

     //Computed heuristic value for our game tree
    if(gameTree.isLeaf()) {
        return gameTree.val();
    }

    //Maximizing player
    if(isMaxPlayer) {

        int bestVal = Integer.MIN_VALUE;
        for(Node node : gameTree.branches()) {
            bestVal = Math.max(bestVal, alphabeta(node, alpha, beta, false));

            //If my value is > beta, the minimizing player above will never choose me!
            if(bestVal > beta) {
                return bestVal;
            }

            //Represents the best current value for the maximizing player
            alpha = Math.max(alpha, bestVal));
        }
        return bestVal;

     //Minimizing player
    } else {

        int bestVal = Integer.MAX_VALUE;
        for(Node node : gameTree.branches()) {
            bestVal = Math.min(bestVal, alphabeta(node, alpha, beta, true));

            // If my value is < alpha, the maximizing player above will never choose me!
            if(bestVal < alpha) {
                return bestVal;
            }

            //Represents the best current value for the minimizing player
            beta = Math.min(beta, bestVal);

        }
        return bestVal;
    }
}
```

Alpha beta pruning is very similar to the above minimax algorithm with a couple of modifications. The key insight is that the naive minimax algorithm might be doing additional work by searching down redundant branches. Imagine I am the maximizing player. I traverse down a branch and discover I can gain a value of, lets just say 5 from that branch. Then, the next branch I traverse down, I realize early on that the best value I can gain is 3. Well, I can stop exploring, because I know I am not going to choose that branch anyways! This is the main motivation of alpha beta pruning. We keep track of the optimal values for both the maximizing player (the alphas), and the minimizing player (the betas) at each node and prune accordingly.

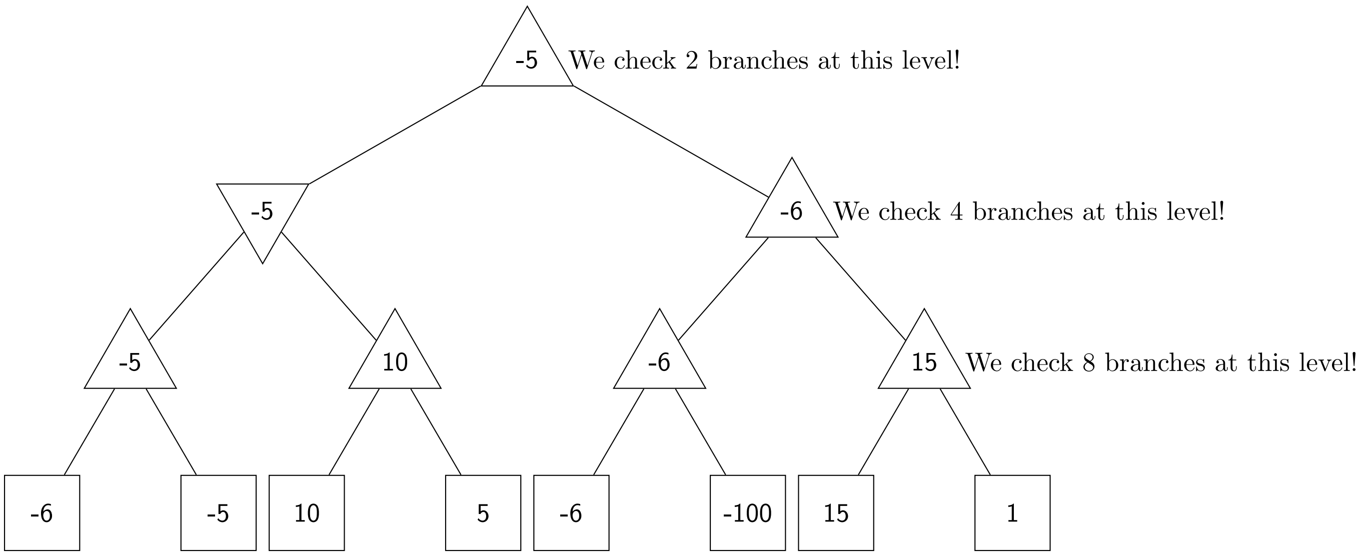
4 Run time Analysis

So far we have learned about the mini-max algorithm as well as the role alpha-beta pruning plays in making our search for the best move more efficient. But exactly how helpful is alpha-beta pruning? Well now that we know how to do Asymptotic Analysis, we can actually compare a simple mini max algorithm and a mini max algorithm with alpha-beta pruning, and quantitatively see how much alpha-beta pruning can help us.

4.1 Without Alpha-Beta Pruning

Although Game Trees aren't actual trees, we picture them to have a tree structure. These tree structures can have a depth of d, which corresponds to how long our game goes on, and at each depth, our nodes can have a branching factor of size b which corresponds to the number of possible moves we can make at each depth.

Without alpha-beta pruning we end up actually checking every possible move up to a certain depth, and choose the best move with the best score. However, when our games can run for a long time, and if at each step we have a multitude of moves we can make, our algorithm will have exponential time!


-5
We check 2 branches at this level!
-5
-6
We check 4 branches at this level!
-5
10
-6
15
We check 8 branches at this level!
-6
-5
10
5
-6
-100
15
1

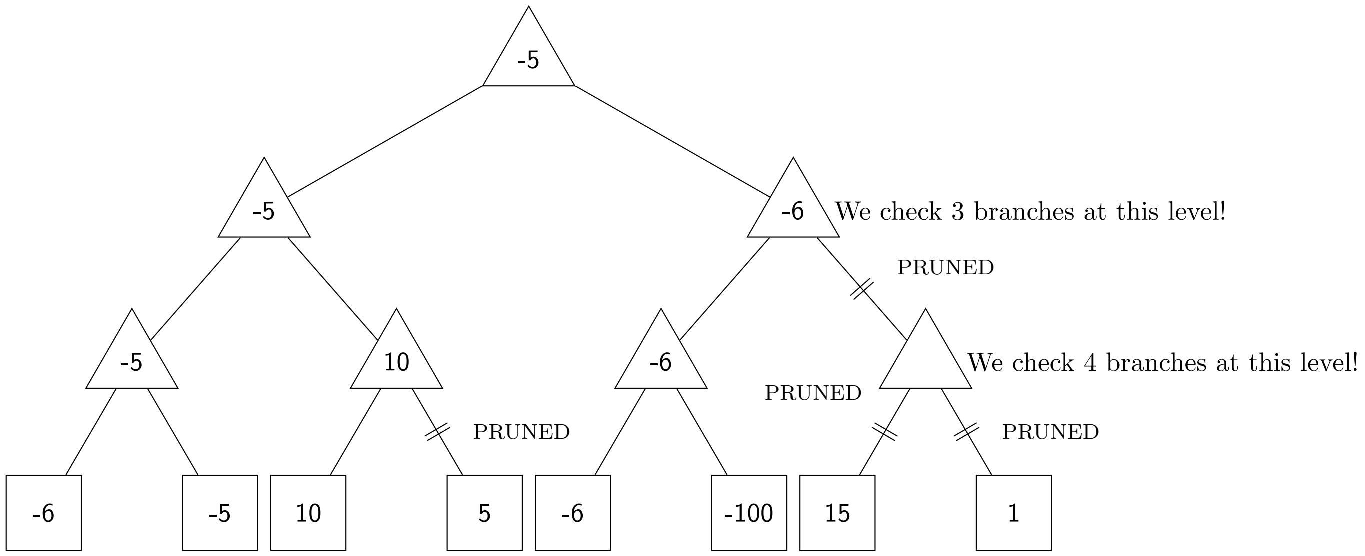
More formally, if our game tree has a depth d, then we have to check b branches at every single level of the tree.

At level d, we would check up to b branches
At level d-1 we would check up to b branches
. . .
At level 1 we would check up to b branches

Since our mini-max algorithm is recursive, this is equivalent to multiplying b to itself, d times. So the run time is $\mathcal{O}\left(b^d\right)$

4.2 With Alpha-Beta Pruning

If we include Alpha-Beta Pruning in our algorithm, we no longer have to check each of the b branches in our Game Tree of d levels. In the case where we prune optimally (i.e. we always check the best moves first so we can prune the rest of the possible moves), we don't need to generate moves for branches we prune, so the amount of work we have to do decreases.


-5
-5
-6
We check 3 branches at this level!
PRUNED
-5
10
-6
We check 4 branches at this level!
PRUNED
PRUNED
PRUNED
-6
-5
10
5
-6
-100
15
1

More formally, at any level, we check the leaves at the left most branch and find the best move. For the branches to the right of this branch, we can potentially prune off many of the moves. This would be an optimal situation. If we can do this for many of the branches, then the run time goes from $\mathcal{O}\left(b^d\right)$ to $\mathcal{O}(\sqrt{b^d})$ because we can go twice as deep down our tree and still do the same amount of work, since we won't have to check all the branches at each level.