Assignment 7.2 Alpha Beta Pruning & Threading

Before tackling this topic, I would recommend that you make an attempt to implement your checkers game. If you start this section with a completely programmed checkers game there will be some minor changes needed to accommodate alpha-beta pruning, but that's it. Starting this section with a working game would probably cut down on the potential for many difficult to debug errors.

To start, Alpha-beta uses the exact same recursive tree structure so keep the pictures from section 7.1 in your head when reading this section. Hopefully you all remember some of the recursive diagrams from that section and recall how we were able to save some time by skipping recursive branches once we had found an optimal value. For example, if I was at a level that was trying to minimize the heuristic value and one branch reported a value of -infinity, why try any other branches?

However, you may also have noticed that at certain times there appears to be unnecessary duplicated effort. Sometimes these recursive processes can be seen exploring branches when known better values elsewhere in the tree have already been discovered. However, since the recursive function had no memory of the rest of the state of the rest of the tree there was no way to include this information. Alpha-beta pruning is one way to overcome these issues. Alpha-Beta pruning relies on two additional parameters being passed recursively as part of your game tree:

> **α (alpha)** – alpha represents the minimum score that the player looking for high heuristic values is assured. (AKA the smallest value found when looking for large values).
>
> **β (beta)** – beta represents the maximum score that the player looking for low heuristic values is assured. (AKA the largest value found when looking for small values).

In the regular game tree recall that the normal process was to make all your recursive calls, then loop through the heuristic values returned to identify the best value. The only exception to this was if we happened to discover a value of infinity (if at a maximizing level) or -infinity (if at a minimizing level). In alpha-beta pruning we instead rely on a local variable called bv for best value below to help us track the best value at the current level as we are making the recursive calls.

So suppose we're at a maximizing level:

> bv is initialized to INT_MIN.
>
> Our first recursive call returns 10. bv is updated to 10;
>
> Our second recursive call returns 20. bv is updated to 20.
>
> Our third recursive call returns 15. bv stays at 20, etc.
>
> (This process is flipped for minimizing level).

Each time you update the value of bv: There are two things that may need to be done. Assuming we are on a maximizing level:

If bv is greater than alpha update alpha.

Recall that at any given point alpha represents the lowest possible heuristic value that the maximizing player could choose. Since we're on a maximizing level, the lowest possible value the maximizing player could choose is the highest between the alpha value that was passed in and the highest value found at this level.

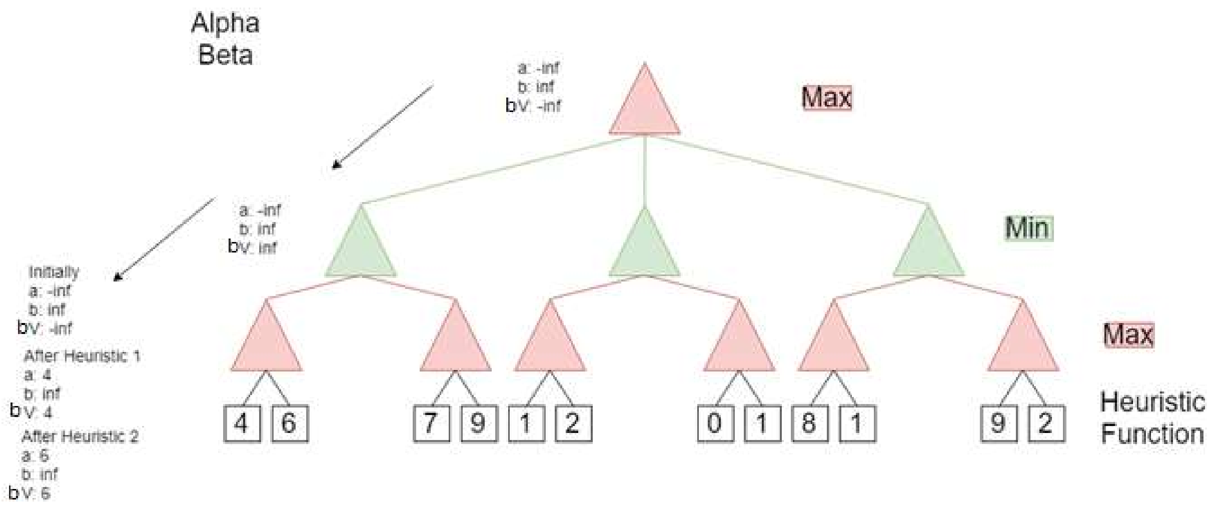
If bv is also higher than (or equal to) beta.

Give up! This one is perhaps less obvious, but remember beta is the highest possible value that the minimizing player could choose. This level is **not** a minimizing level, so if you end up with a value that's worse for the minimizing player then they would never choose this branch at the previous level of the tree. Since we now know this branch will never be chosen there is no point in doing any additional branches. Give up and return whatever value you have for bv.

This process is flipped for a minimizing level. I.E. If bv is less than beta update beta. If bv is also less or equal to alpha give up.

Note: Keep in mind that these values are passed in! As you return from recursion the values of alpha and beta will revert to what they were at the previous level to represent the state of the board at that position.

The next page will illustrate what this looks like with a recursive diagram.



- Note that on the way down the first series of recursive calls that alpha and beta are initialized to -infinity and infinity respectively.

- Since we don't have any heuristic values to analyze at this point, the values for alpha and beta are passed all the way to the bottom of the tree. When the ply is zero, the heuristic function is called and 4 is returned.

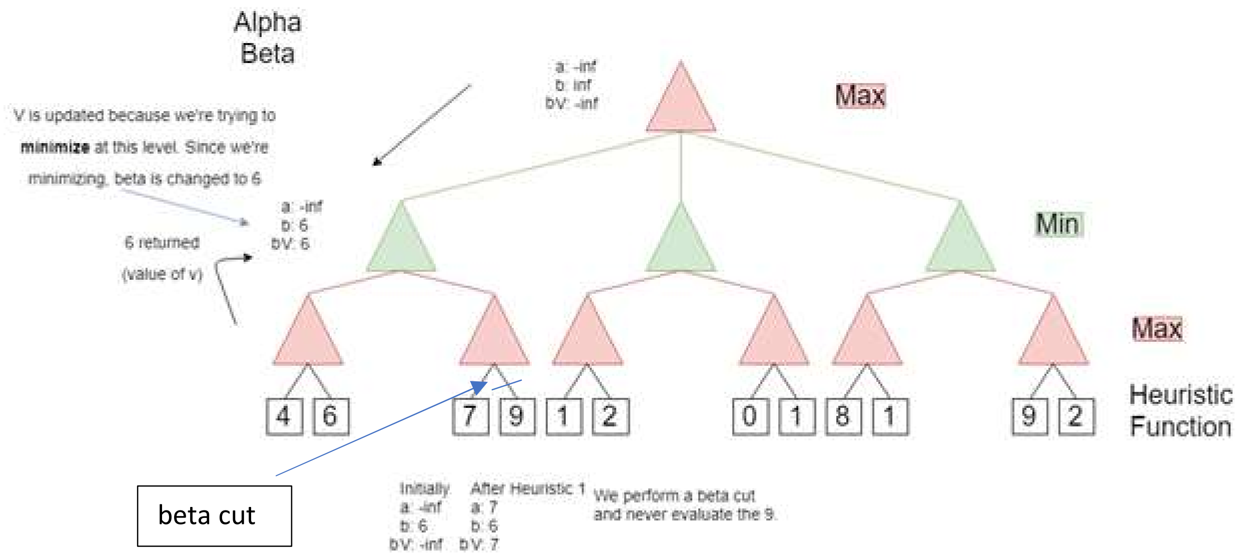
- The bottom most level in this example is a maximizing level, so bv was initialized at -infinity. Obviously 4 is an improvement here, so we update bv. Next, we check if 4 is greater than alpha (passed in as -infinity). It is, so we update alpha. Lastly, we check and 4 is not greater than or equal to beta (passed in as infinity), so we don't quit.

    - **Note**: Whether or not the bottom most level is a max or min level depends on how many layers deep your algorithm is searching. Don't assume that it will always be a maximizing level!

- The second recursive heuristic call returns a 6. 6 is an improvement over bv (currently 4), so it's updated. Since bv was updated we check to see if 6 is an improvement over alpha. It is, so we update it. 6 is also not greater than infinity so we can't update beta and quit.

- That's all the recursive branches for this level! You might be asking yourself, if beta is infinity how will we ever quit early!?! Remember that beta only changes on min levels and we have yet to evaluate any min values, so there's never going to be a chance of quitting early on this first branch.

    - So what happens now? Just like in the initial game tree if you don't quit early the best heuristic value is returned. The only difference is that we've been calculating that as bv as we make each recursive calls, so there is no additional code at this point. We just return bv (6 in this case).



You can see a lot has changed in this second picture. Remember that the last image left off with us returning bv on the bottom-left red triangle. At that point, bv had a value of 6, alpha was 6 and beta was infinity.

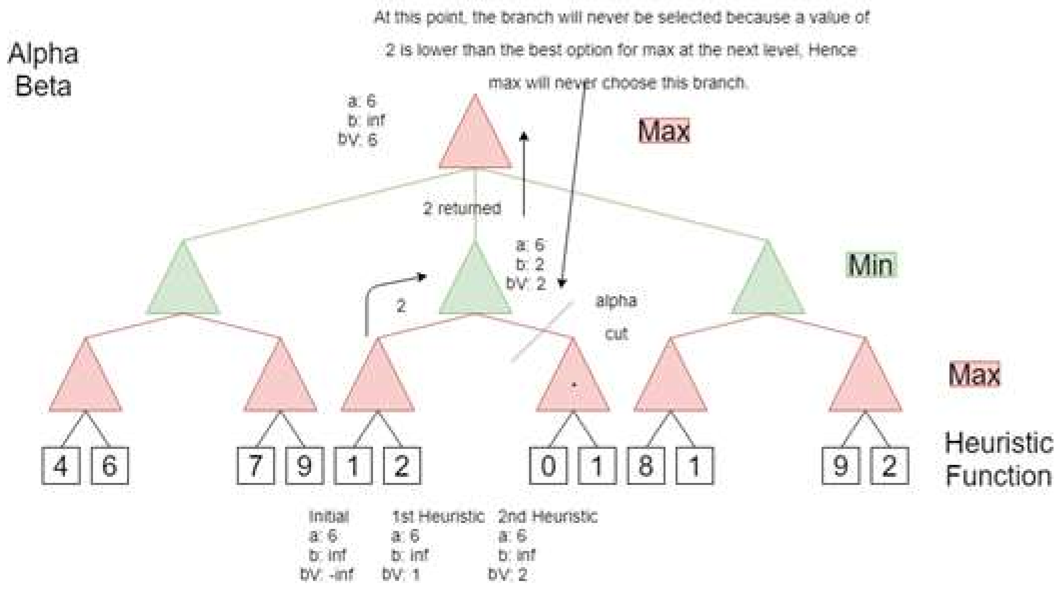
Returning back in recursion to the leftmost green triangle note that we are now on a min level. alpha now has a value of -infinity, beta is infinity, and bv is infinity (since this is a min level).

- Remember these were the values for alpha and beta passed down to us from the top. The value of alpha **was** changed at the red node below us, but these are **local variables** so when we leave a function they reset to whatever they were before the recursive call.
- Since this is a min level we are trying to update bv with **smaller** values. The 6 returned from the first recursive call is definitely smaller than infinity, so bv is updated to 6.
    - Since we updated bv, we first check if beta needs to be updated. Beta is still infinity and is updated to 6. A check to see if 6 is an improvement over alpha is performed, but alpha is still -infinity, so we cannot cut the tree at this point.
- We now make the second recursive call from the green min node and descend down passing alpha in as -infinity and beta as 6.
- **Notice how this situation is different than the first call!**
    - Beta is telling us that the highest min value found so far is 6. That means if **any** of the recursive calls at this level return something higher than 6 the whole tree is useless as it will never be selected at our parent level.

The first call to evaluate our heuristic is made and 7 is returned. On a max level this will update bv (initialized to -infinity) and alpha (passed in as -infinity) both to 7.

Next, we check to see if bv is greater than b. It is! This means this tree will never be selected. We've already encountered 7, which is a worse choice at our parent level. If the remaining recursive branches return smaller numbers they'll never be selected as the max value at this level. If they return greater numbers, they'll never be selected at the parent level. So we just stop here and return 7.

- In the picture it's show the heuristic returned for the next move would have been 9, but 7 was returned. 9 is greater, but this is irrelevant since we know the branch won't be selected.



This picture again jumps forward a bit, but hopefully the process is starting to make sense. A quick examination of this and you can start to see how this adjustment in your code can be powerful as it not only cuts off small branches at the bottom, but large ones closer to the top of the tree.

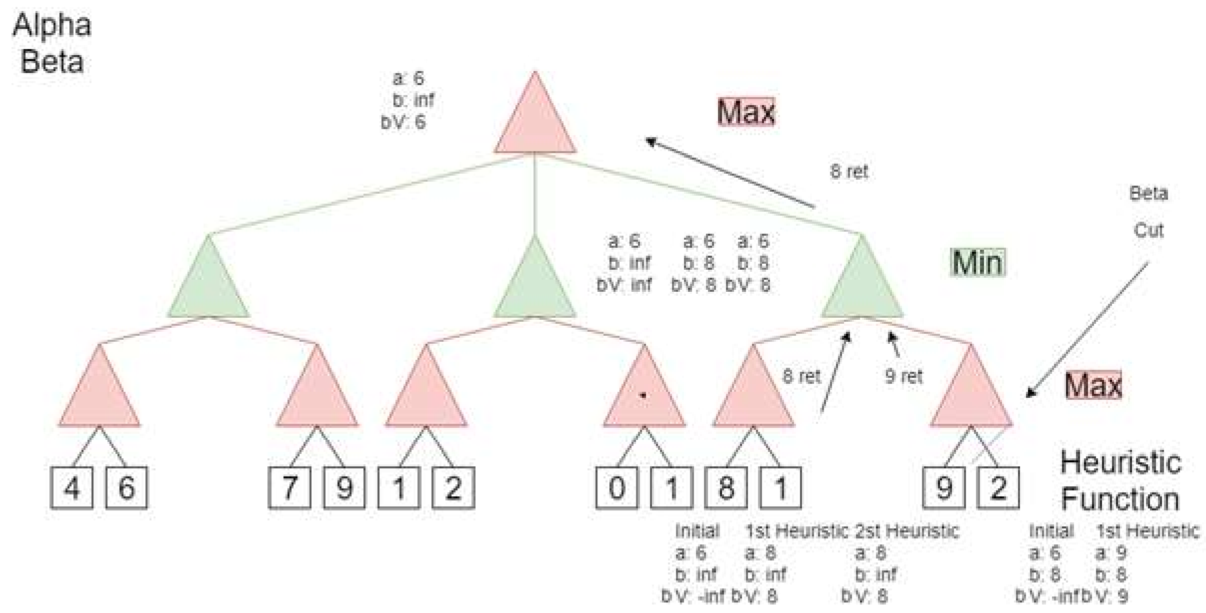
From the previous picture we know that 6 was returned to the top most level from the left branch of the tree. Since this is a max level bV and then alpha were updated (both were -infinity prior to this). The call to the center branch was then made with alpha: 6 & beta: infinity.

alpha & beta are again passed all the way to the bottom and the first heuristic value was returned (1).

Since this is a max level, bv starts at -infinity and is updated to 1. However, 1 is not an improvement over 6 and neither is 2 so alpha is not updated at this level.

bv (2) is returned to the min level and this, in turn, updates bV at the min level (previously was infinity). As this is a min level, beta is updated as bv (2) is less than beta (infinity).

At this point, we compare bv (2) to alpha (6) and notice that we can trim skipping the entire right site of the tree. This works because if the right tree returns a value larger than 2 it will never be selected as this is a min level. However, 2 or any smaller value will never be selected at the parent level because we try to maximize at that level and we’ve already produced a branch of the tree with a value of 6.

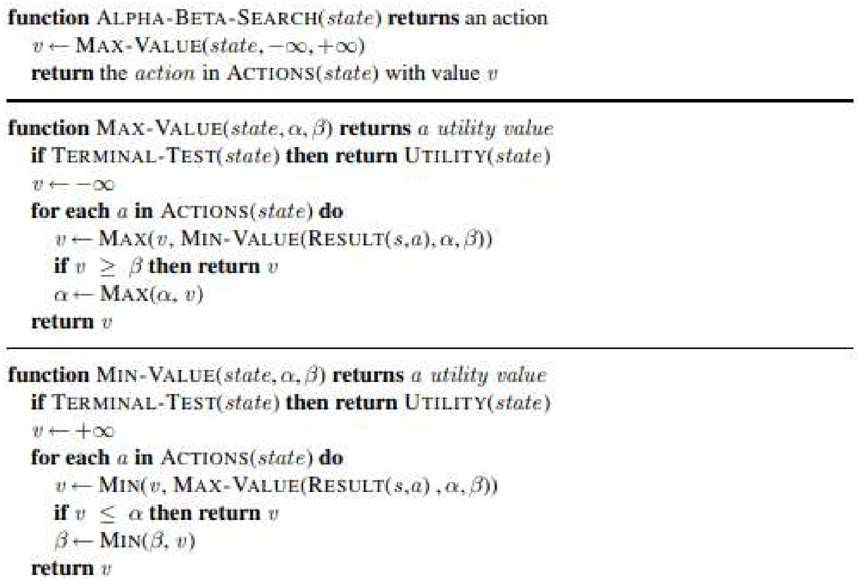


Alpha Beta
a: 6
b: inf
bV: 6
Max
8 ret
Beta Cut
a: 6 b: inf bV: inf
a: 6 b: 8 bV: 8
a: 6 b: 8 bV: 8
Min
8 ret
9 ret
Max
4
6
7
9
1
2
0
1
8
1
9
2
Heuristic Function
Initial a: 6 b: inf bV: -inf
1st Heuristic a: 8 b: inf bV: 8
2st Heuristic a: 8 b: inf bV: 8
Initial a: 6 b: 8 bV: -inf
1st Heuristic a: 9 b: 8 bV: 9

Shown here are the remaining alpha beta calculations. Recall from above that when the call to the third call to the min level is made alpha is 6 from the first tree and beta is still infinity as beta is only updated at min levels and there are no min levels above this point in the code.

Try and follow through the rest of the process. All the values of the function calls are listed. Note that when the process is finished 8 gets returned. This is an improvement over the current bv (6), so both bv and alpha are updated. Assuming this is the topmost level the right movement should be chosen by the computer player.

Pseudo Code:



```
function ALPHA-BETA-SEARCH(state) returns an action
  $v \leftarrow$ MAX-VALUE(state, $-\infty, +\infty$)
  return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT(s,a), $\alpha$, $\beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE(state, $\alpha$, $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT(s,a) , $\alpha$, $\beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Swiped from: Artificial Intelligence: A Modern Approach by Russell & Norvig 3rd Ed.

This pseudo code may have a slightly different format from your code, but keep in mind this can be written as a single function with an if/else statement to determine turn.

The bv variable described above is just called v here.

ACTIONS(state) is the function that establishes the array/vector of possible checker moves for the specified player. a then represents the current move.

Threading

Threading this project is pretty straight forward. Each branch of the recursive tree can be calculated by a different core of your CPU to further increase performance. However, to accomplish this, you have to create a sort of hybrid of Alpha Beta pruning and the original game tree.

If you've never created a threaded program before this is about as easy as it gets. The threads do not need to interact in any way. Each thread will run completely independently and return a calculated value. However, to do this you have to give the thread a name of a function to run and (optionally) a data set to operate on. We'll get to these pieces in a minute

The top level function (I called mine Layer1)

The topmost level will create an array of moves which will each get executed on a separate thread executing a slightly different function that performs alpha beta pruning. To do this, you'll need to create and fill an array of moves just as described with the initial algorithm. In my code this looks something like this:

```
InitialMoves = new CheckerMove[48];
for (int i = 0; i < 48; i++)
       InitialMoves[i] = new CheckerMove();

int moveCount = 0;

GetMoves(InitialMoves, ref moveCount, CheckerType.Red, black, red, kings);
```

InitialMoves is created as a class variable as access is needed both in this function as well as the threaded function. This is because each thread will be responsible for calculating a heuristic value for its checkerboard, but when all threads have finished we'll need to return the optimal value discovered from all threads back to our Layer1 function.

We now create an array of threads to process each of our moves:

```
Thread[] ThreadArr = new Thread[moveCount];
```

Note just like the first line above with InitialMoves this only dimensions the array. We now need to create each individual thread:

```
for (int curMove = 0; curMove < moveCount; curMove++)
{
       ThreadArr[curMove] = new Thread(new ParameterizedThreadStart(ThreadMove));
       ThreadArr[curMove].Start(InitialMoves[curMove]);
}
```

You can see in the loop where each thread is created and is passed a ParameterizedThreadStart object. This object is used to specify the name of the function the thread will operate on, which is ThreadMove here.

ThreadMove looks like this:

```
private void ThreadMove(object obj)
{

…

}
```

and is responsible for starting the alpha beta pruning version of our code. You may have noticed that the Start() function call as well. This call is responsible for actually starting the thread running and, as such, passes in the object that will be sent into the thread. Here we are passing in a member of the InitialMoves array, which is of the type CheckerMove discussed in the previous document.

However, you can see the ThreadMove function receives this piece of data as an object. To make use of this object inside of ThreadMove, we must first type cast it back. In C# this looks like:

`CheckerMove curMove = obj as CheckerMove;` (This would most likely be the first line of ThreadMove)

Note: This is still just a one way process. To get the heuristic back out of ThreadMove you may want to augment a CheckerMove to store the index position of this board inside InitialMoves. This index could then be used to update the class-level array. There are many ways to do this, but this worked for me.

At this point, ThreadMove makes a call to my normal alpha-beta game tree and waits for a heuristic value to be returned.

If you've threaded an application before, you know that threads run asynchronously from the invoking function. This means once you've launched all the threads our Layer1 code will just continue on executing unless we tell it to wait for the threads to finish. Sometimes this is needed sometimes it isn't, but in this case we need to be able to choose a move from all those evaluated by our threads, so we must wait for them to finish. In C# this is done with a Join() call on the thread in our Layer1 function. This needs to be done on each thread to ensure that all threads have returned before we chose our best value. In my code, this looked something like this:

```
int bestID=0;
for (int curThread = 0; curThread < moveCount; curThread++)
{
    ThreadArr[curThread].Join();
    if (InitialMoves[curThread].HVal > InitialMoves[bestID].HVal)
        bestID = curThread;
    ThreadArr[curThread] = null;
}
```

```
This is pretty much it! At this point, the Layer1 function then returned the element of
InitialMoves that had the bestID.
```

If you haven't programmed with C# very much you may be asking why there's no delete statement after we set the TreadArr position to null. This is because C# is garbage collected. Each piece of dynamic memory that is declared keeps a count of how many pointers are pointing to it. By setting the ThreadArr position to null it reduces the pointer count to zero on the thread and the system automatically recognizes it can release that memory. Since ThreadArr is a local variable this line really isn't even needed as when the declaration goes out of scope the pointer count also would go to zero.