AI Module 22
Alpha Beta cutoffs

Introduction

We have already seen the MiniMax algorithm as well as need for improvement using alpha beta pruning in previous module. We will see how we can write a modified algorithm incorporating alpha beta pruning in this module. We will also see how we can improve the performance of the algorithm using some other measures.

The algorithm that we propose here is again a recursive algorithm with two more parameters. One is used for checking the threshold value against the nodes heuristic values and another is to set threshold for the next level; as we have different threshold values of Min and Max levels. The process of the cutoff is applied after each node is examined and children are returned with their MiniMax values.

If the ordering of the moves is perfect, they can save exponential amount of nodes to be explored. We will discuss an output of one such study at the end.

MiniMax with Alpha Beta Pruning

Now we will see how the MiniMax algorithm that we have presented in the previous module can be modified to include alpha beta pruning. We have already seen that the cutoff we apply at maximizing level is called beta and minimizing level is called alpha. While we are exploring a minimizing level, we can avoid a move early if we find that the values are greater than the current threshold as we know that opponent is unlikely to choose that move. Ruling out a move by us (a maximizing player), happens while searching at minimizing layer. Alpha cutoffs are applied *by the maximizing player*, cuts off *moves at minimizing level*. Similarly *Beta cutoffs are applied by minimizing player* (the opponent) cuts of *moves at maximizing* ply.

It is also important to note that alpha and beta cutoffs are not applied together but at alternative plies. Our recursive algorithm needs to pass both values at every level but at one level only one of them will be used. For example at maximizing level only beta is used, but when we call a minimizing level (next level) from it we need the alpha value as a threshold. Similarly, at minimizing level we only need alpha, but we also need to call next level as maximizing level from it and we need to pass beta to it. Thus both values are to be passed to each level.

We have designed MiniMax algorithm in a way that we do not need to differentiate between maximizing and minimizing level. We have just negated best values at alternate levels to create that effect. We would like to do the same while updating the algorithm for alpha beta pruning. To do that, we need to pass two threshold values, one which is current and another is next. The CurrentThreshold value is to be applied to current level to allow or cutoff moves while NextThreshold is to be applied

when the next level is to be explored. Thus we will pass NextThreshold value to next level as CurrentThreshold. The current threshold value for current level becomes next threshold value for the next level as that threshold is to be applied to next to next level. Thus we will not be specifying which one is alpha and which one is beta. If the current level is maximizing level, the CurentThreshold is beta and NextThreshold is alpha and vice versa for minimizing level. Exactly like the heuristic values, we will negate these values too to remain in sync. As we are negating at each level exactly with the heuristic values, our testing of best value being greater or less than these threshold values makes sense.

You probably have noticed that the use of alpha or beta threshold value requires it to set in the previous level. For example let us take the case depicted in figure 22.1 which is same as 20.4.

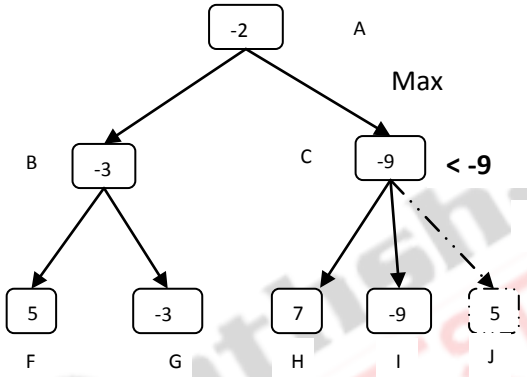

Figure 22.1 alpha cutoff

While we are exploring the second level and when we encounter -9, we know that we already have the other option of playing B to get -3 so the maximizing user will not choose this move (C). This value -3 was set during exploration of level 1. Thus this value -3 is the best so far node at level 3. The idea of alpha cutoff is to make sure that when a minimizing node guarantees to provide lesser value than -3, there is no point in exploring other siblings of that node. The value that we use for cutoff (-3) is decided at B, level 1, which is used as the threshold value at level 2. Thus the threshold value applied at level n is the best value one gets at level n-1 for alpha cutoff. Similarly look at figure 22.2 which is essentially same as 22.1 to see if it is also true for the beta cutoff.

Our beta cutoff, -9 has come from a sibling H. when we get a better value than -9, we know this being minimizing ply, and the opponent is not going to play any move better than -9 as he already have the option of playing -9. Thus the decision made at level 3, is decided on the basis of threshold value calculated at the previous level. Thus the value for beta cutoff is applied at the minimzing ply but the threshold decided at the previous minimizing ply. While we are exploring children of I, the value found at previous level (-9) must be passed to it. Thus the same thing is true for minimizing ply as well!

Thus we can state a simple rule of thumb for this process, the best successor of the previous level is to be considered as the threshold value for current level.

That means that the NextThreshold is to be updated with the best value one gets during current ply while the CurrentThreshold will be used to check the further exploration at the same level. You can see

that in figure 22.3, in case of level 1, the CurrentThreshold value is 3 which is passed to next level while we are exploring.

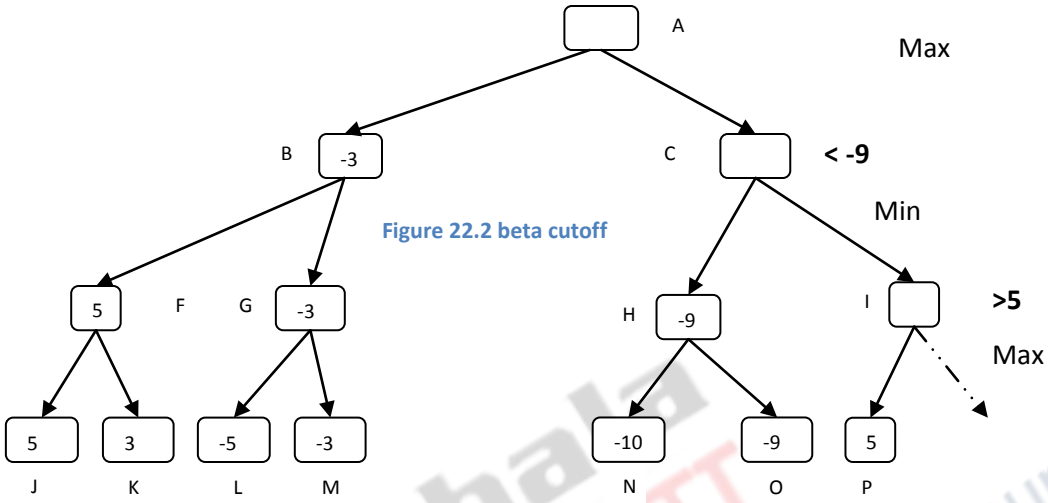

Figure 22.2 beta cutoff

Another point is also important to note. There are two different values to be considered. First the maximum value at parent level so far and maximum value overall.

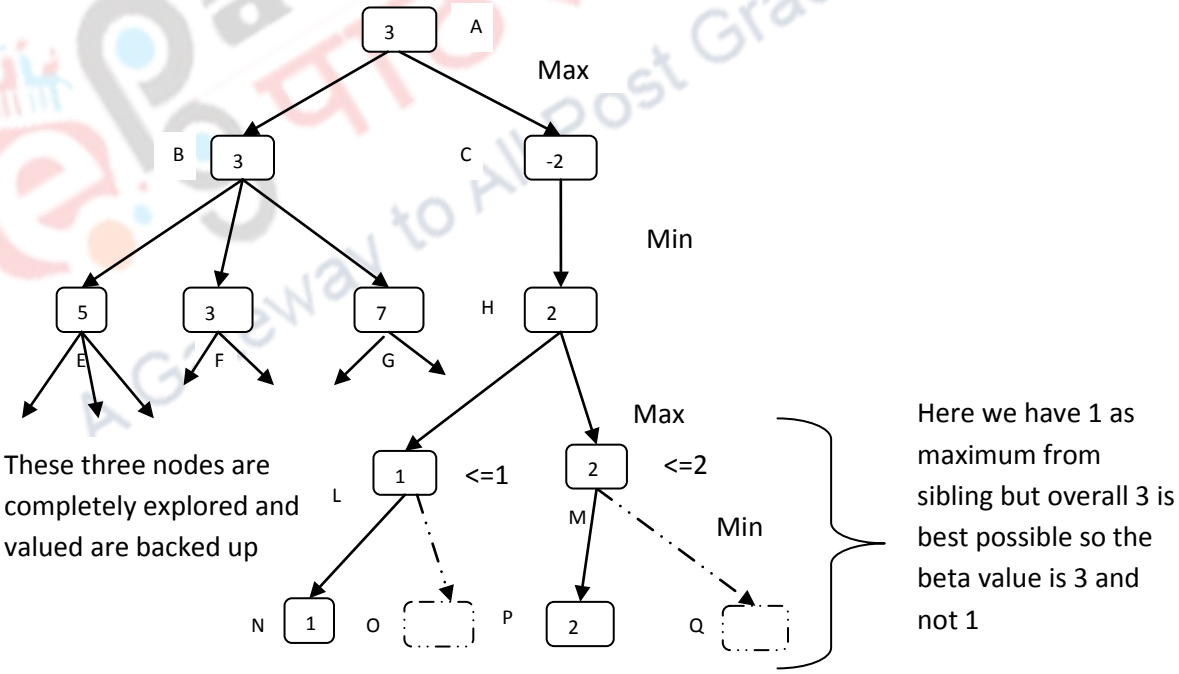

Figure 22.3 The cutoff is maximum possible value

Look at figure 22.3. This is a game tree with 5 levels till the leaves. We can call it 5 ply tree. The leaves are evaluated and their values are backed up one after another. The branch where the root node is B, is

already explored with best value being 3. We are taking a case where C is being explored. We begin with the exploration of N, passing the value up to L, checking with threshold and omitting exploration of O, pass the value up to H and start exploring M and its child P, we apply SEF to P and yield 2, which we pass up. This is where we have taken the snap shot of the tree.

When L explored, we get 1 from N and thus it is less than 3 which is guaranteed from B so we do not explore O. now when we are exploring M and we get 2 from P, we have a problem. If we compare this node with the sibling L, we have a better value so we should explore as we are guaranteed to get 2 from this. Though the sibling provides an upper bound of 1, an inherited upper bound from A will be 3. That means that A already has a possibility to play B with a guarantee of 3. If we explore the sibling of P, called Q, we are guaranteed of maximum 2. If O has higher value, the opponent will choose P. Though this move is better than the sibling L, it is still worse than the best so far, B, and thus we will not explore other children of M. In figure 21.2 we can see that we can continue working on the same problem using 3 as the best score so far and eventually coming to the conclusion that we must pick up B as our next step. That means even when sibling is better, if it is not better than global best so far, we only will look at global best so far value. That means, we will also be passing this best so far down the tree while exploring with other options.

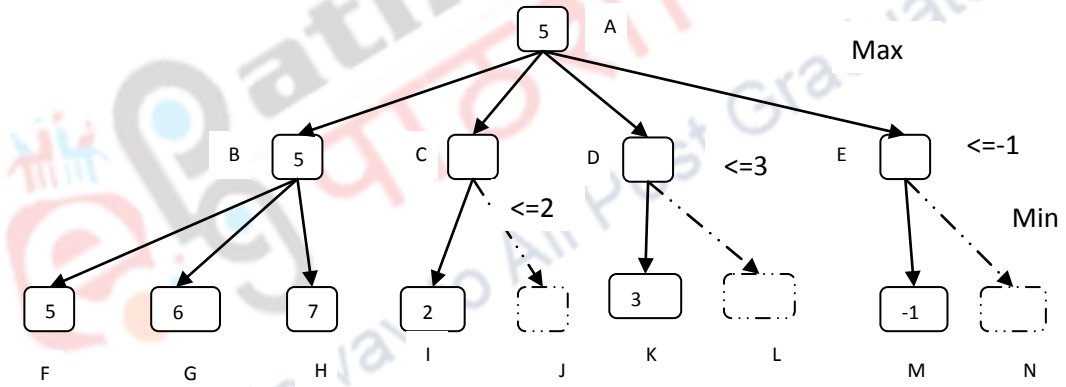

Figure 21.4 The further exploration of the game tree depicted in 21.1

One more important observation is about ordering the exploration. As we have explored the best node B, before C, D and E, it becomes easier for us to have alpha cutoffs at all of them. We are not only removing one child as it might seem from the figure, but more than one child if there are, and all children with complete sub tree beneath. The amount of save is almost 50%. What if we have explored E, D, C before B? We would not have that advantage. If we have explored E first, we will have to examine all other children of E. Assuming all other children are better than -1, we will have -1 from E. That will not help us alpha cutting K and other children of D. We might have some cutoff at C but again B will have to be completely explored to find that it is better than the rest. Thus the order of exploration is an important issue. If we can somehow know which nodes are more promising, we can certainly take better advantage of alpha beta cutoff.

Algorithm

Here is the algorithm.

Minimax (CurrentState, CurrentDepth, CurentPlayer, CurrentThreshold, NextThreshold)

1. If Over (CurrentState, CurrentPlayer)
   {
   NodeInfo NF;
   NF.Path= NULL
   NF.Value = SEF(CurrentState,CurrentPlayer)
   return NF,
   }
2. Else
   ChildrenStatesList = PMG(CurrentState, CurrentPlayer)
3. If ChildrenStateList == []
   {
   NodeInfo NF;
   NF.Path = NULL;
   NF.Value = SEF(CurrentState,CurrentPlayer);
   return NF;
   }
4. Else
   for each Child in ChildrenStateList do
   {NodeInfoChildValues
= MiniMax(Child, CurrentDepth + 1, 1 – CurrentPlayer, -NextThreshold ,-CurrentTHreshold);
   ChildSEF = (-1) * ChildValues.Value;
   If ChildSEF>NextThreshold
   {
   NextThreshold= ChildSEF;
   Path = Child + ChildValues.Path;// add Child in the front of the path
   }

   If CurrentThreshold>ChildSEF// Cutoff
   [
   NodeInfo NF;
   NF.Path =Path;
   NF.Value = NextThreshold;
   Return NF;
   }
   }

5. NodeInfo NF

```
NF.Path  =  Path;
NF.Value =NextThreshold;
return NF;
```

The process

The process is exactly like conventional MiniMax described in the previous module additionally considering alpha beta cutoff. Let us try to understand.

We begin with our old Over function as well as structure NF. There is no change here so we will not describe it further. Step number 1, 2, 3 also are the same. Next part is different so important here.

For each successor of the current node, we first invoke MiniMax algorithm. We will increase the depth by one, change the player, and change both the threshold values position as well as sign. Why? Because the current threshold will become next threshold now and next threshold becomes current now. We have seen why we do that before. Next is assigning the value returning from the Minimax call to value ChildSEF. If this ChildSEF is better than current threshold value, it is a better node than what we have seen so far (this part is also similar to pervious algorithm, if we get a better node than the BEST, we will make it BEST). Also, the best path should travel through this node so we adjust the path accordingly.

The only logic left now is for cutoff. If our current threshold becomes greater than next threshold (the cutoff value), there is no point in exploring this node's children, so we do not explore other children and return immediately. If that does not happen with any one of the children and all children are exhausted, we will return with the best values like before.

Closely observe following statement

```
If ChildSEF>NextThreshold
```

The value of the current node is more than what the threshold is. NextThreshold is best so far and thus we need to have a better best so far now;so the next statement.

```
NextThreshold = ChildSEF;
Path = Child + ChildValues.Path;// add Child in the front of the path
```

Remember our discussion. The threshold which is to be used in the next ply is set now. So we are setting the next threshold's value. We are setting next threshold's value right here.

Also observe another statement

```
If CurrentThreshold>ChildSEF // Cutoff
```

What are we doing now? The current threshold value is set up at the parent level, we are testing it if it is more than the node's child's SEF. If so, there is no point in exploring that node further. So we return immediately.

One last point; how do you think the algorithm will be called initially?

MiniMax(CurrentPosition, 0, CurrentPlayer, 10,-10)

Why? It is because we need to pass maximum threshold value for CurrentThreshold and minimum for next threshold. Thus, we can have any node having any value more than -10 can be used to set threshold for next level. (It is very similar to setting minimum value when we want to get maximum of some numbers). The current threshold will be set in the next level so we have kept it as a maximum value now which will become a minimum value in the next level and any node with little better value can be accepted.

Futility cutoff

Some research is done on how much saving is possible using alpha beta cutoffs on a general purpose game tree. We have already seen in figure 21.4 that ordering determines the amount of saving. Knuth and Moore proved an interesting theorem. It says that if we have n nodes for a depth of length d without using alpha beta cutoff and we examine all leaves and back the values up. Now if we apply alpha beta cutoff and we also somehow manage to have the tree perfectly ordered. The result is that we can examine the leaves of the same tree for double the depth (2d) and back the values up.

This is indeed a huge saving. For example if a binary tree case, each level adds almost the same number of nodes that we have already explored. Thus even when we save a time for exploring just one level in a binary tree, it is double than without alpha beta cutoff. Having d more levels is 2^d; and for an m-ary tree it is m^d. This is really big saving.

An interesting proposal is not only to ignore nodes which are poorer, but also some of them who are just little better. For example, if we get a threshold value as 5 and if we get a node with a value 5.1, there is no point in exploring it as well as anything less than 5. This is known as futility cutoff.

Summary

We begun this module with understanding that alpha cutoffs are applied at minimizing ply but the threshold for which is decided at previous maximizing ply. Similarly beta cutoffs are applied at maximizing ply but the threshold for which is decided at previous minimizing ply. We followed this and decided two values; one value which is to be passed to next level for testing is set in this ply. One which is set in the previous ply is used to test if the heuristic value is going above threshold here. We use CurrentThreshold is one which we test our values against and NextThreshold which we set during exploration of this ply. The recursive procedure is quite similar to previous module except for passing CurrentThreshold and NextThreshold values. Both values are negated at every ply for the same reason we negate the heuristic values. Additionally, the current threshold is to be set in the next level and must act as NextThreshold in the next ply. Similary the NextThreshold value that we are setting here is to be used for testing in the next ply so should act as CurrentThreshold value. This switching over process happens continuously and thus the algorithm changes the position of both these values across calls. The saving achieved using alpha beta cutoff depends on right ordering. If the ordering is perfect, the saving

is exponential. One can not only avoid worse paths but also paths which are just little better than current and thus can improve search time.