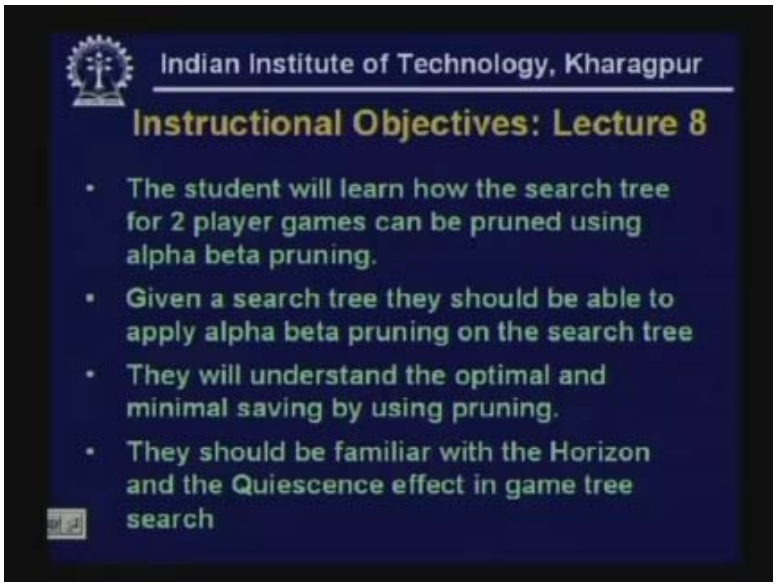
Artificial Intelligence
Prof. Sudeshna Sarkar
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture - 8
Two Players Games - II

Today is our second lecture on game tree search. In the last class we talked about two player games and how we can formulate it as a search problem. Today we will have the second part of this lecture.

(Refer Slide Time: 01:24)



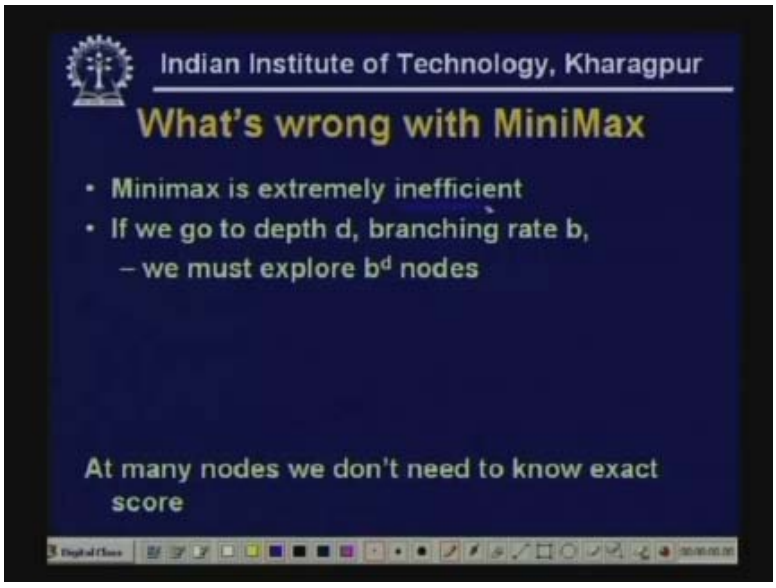
In this lecture the student will learn how the search tree for two player games can be pruned using alpha beta pruning. In the previous class we looked at MINIMAX search for computing the best move for a player. But MINIMAX is quite inefficient and we will see how MINIMAX search can be pruned using alpha beta cutoff values. Given a search tree the student is expected to able to apply alpha beta pruning on the search tree and show which are the nodes which will be expanded and the nodes which will be pruned.

The student will understand what is the optimal saving by using alpha beta pruning for a game tree and what minimum amount of pruning can be obtained?
They will also be familiar with the certain strategies used in game tree search like the quiescent effect, the horizon effect, transposition table and end game database etc. So, to recollect we have considered 2 players perfect information 0 sum games where we have 2 players min and max. And the full minmax tree can be drawn so that the terminal positions are either a win draw or a loss and we can assign a value a pay off value to the terminal position.

(Refer Slide Time: 02:45)

When generating the entire tree is not possible we saw that we can decide to cutoff search at a particular depth at a particular level and use a static evaluation function at the terminal nodes. Thus in the game tree we generate the leaves are given definite score. In the last class we saw how the MINIMAX algorithm defines how the scores are backed up from the leaf towards the root. That is what we studied in the previous class. To briefly recollect the game tree is like a search tree. The nodes are the search states. The edges between the nodes correspond to moves. Leaf nodes correspond to determined positions and at each node it is one or the other player's turn to move. Now the problem with MINIMAX is that MINIMAX is extremely inefficient.

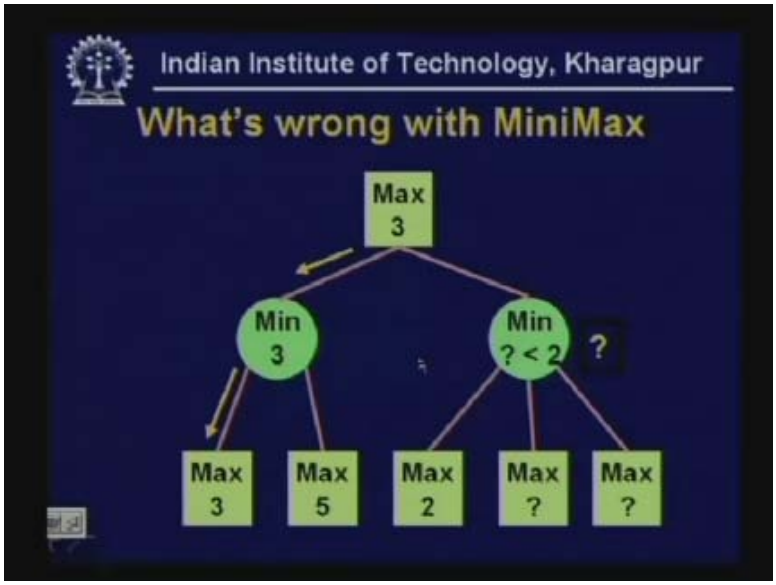
(Refer Slide Time: 04:10)



That is, MINIMAX expands all the nodes in the given search tree. That is, if we start from the root and search p-ply or d-ply up to a depth of d and assuming that this game tree has a uniform branching factor of b then the number of nodes expanded or explored by MINIMAX is b power d. So MINIMAX will expand b power d nodes in order to compute the MINIMAX value at the root if it decides to cutoff search at depth d. However, note that while we are doing this MINIMAX search we are only concerned with the best move for max at the root level.

We really do not need to have the correct MINIMAX value at all the intermediate nodes. We need not have the exact MINIMAX value for all the nodes. We really just need to know the MINIMAX value at the root node or rather we want to know which is the best move at the root node. So, if we consider this we can we show that we can device an algorithm which expands less nodes than MINIMAX does. Before we go to the formal properties of the algorithm we will take up an example and show why such pruning is necessary or such pruning can be done.

(Refer Slide Time: 06:00)



So let us consider this game tree. The square nodes correspond to the max player and the round nodes correspond to the min player. Here it is the max player's turn to move. At this level it is the min player's turn to move and at the lowest level here again we have the evaluation for the max player. Therefore the MINIMAX values can be computed by depth first search. Suppose this leaf gets evaluated first and it has a value of 3 now when this leaf is evaluated the value of the min node must be 3 or less than 3 because min will compute the min of all the children values. So the value of the min node can be 3 or less than 3.

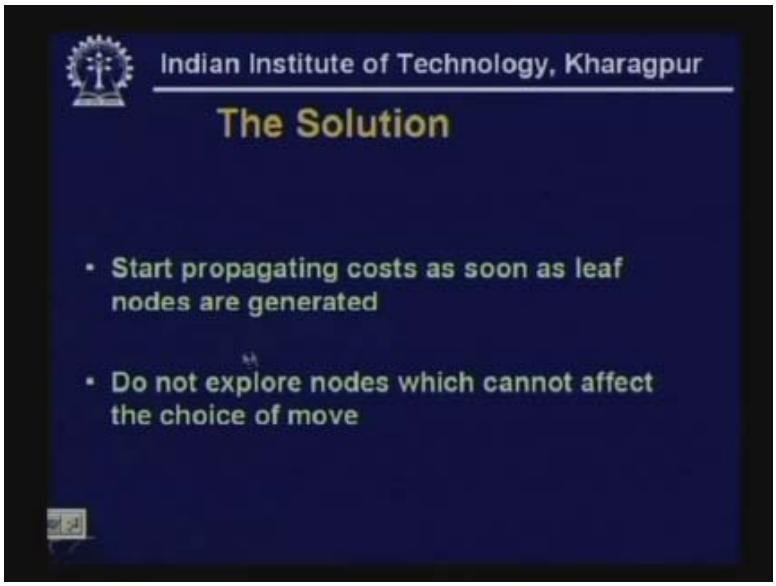
When the second node is evaluated we know the exact MINIMAX value of this min node which is equal to 3. Now, when the MINIMAX value of this node is equal to 3, since this is a maximizing node the value of this node must be greater than 3. The min of this node

is exactly equal to 3. So the value of this max node must be greater than 3 because max is a maximizing node. It has already found a path through which the MINIMAX value is 3. It can try another path. It should go to another path if its MINIMAX value is greater than 3. So when we come here we know that the max value must be greater than 3.

Now we explore this node. Now at this node max has a value of 2. If max has a value of 2 the value of min at this node must be less than 2. But if the value of min is less than 2 then max has already a path where the value of MINIMAX is 3. Along this path the value of MINIMAX is less than 2. So whatever be the actual pay off values at these nodes max will not be interested in this path.

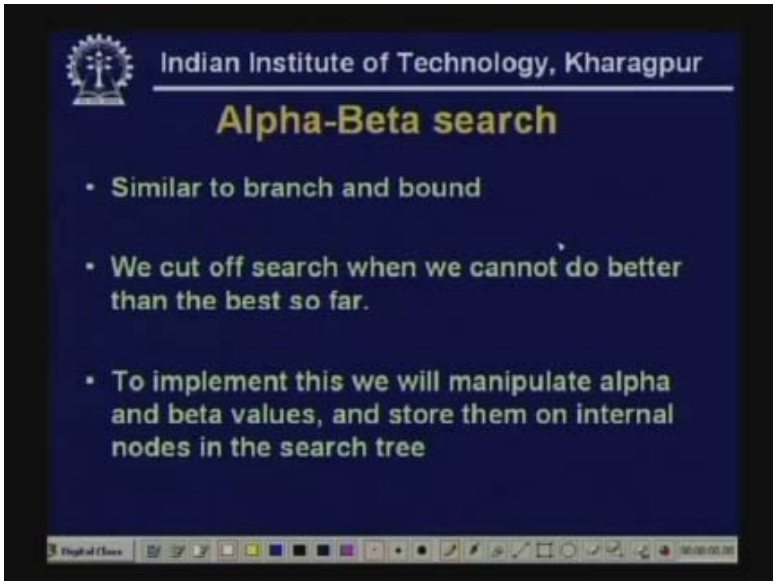
For example, even if the value at this node is equal to let us 100 even then max will not be interested in this path because in that case min value will be still less than 2. If the value here is minus 1 the min value will be equal to minus 1. So the min value at this node can be less than 2 and cannot be greater than 2. Therefore the values of these nodes are not important for max because max knows that this is a path with a better guarantee. So max need not compute these nodes. So search can be cutoff at these two paths. This is the idea of pruning that we will talk about today. So what we do is we start propagating the costs as soon as the leaf nodes are generated.

(Refer Slide Time: 09:45)



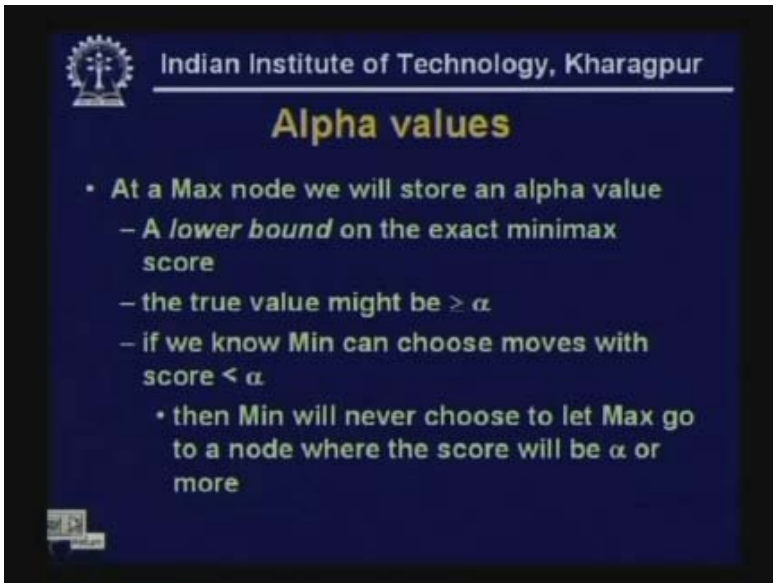
We do not explore nodes which are guaranteed not to affect the choice of the move using MINIMAX search. So we want to prune the search tree so that we still get the same strategy as we would get by full MINIMAX search but we avoid expanding those nodes which do not have any effect on deciding the correct move for max. This strategy of our pruning is called alpha beta pruning. So alpha beta pruning is some type of branch and bound pruning. And the idea is that, we cutoff search when we cannot do better than the best so far.

(Refer Slide Time: 10:34)



To implement alpha beta pruning we associate with the nodes, the alpha and beta values are associated with the nodes. Usually alpha is meaningful for max nodes and beta for min nodes. So we can keep our beta values at all the nodes and alpha value of a min node could be the alpha value of its max ancestor and beta value of the max node is the beta value of its min ancestor. So at every node we store this alpha and beta values.

(Refer Slide Time: 11:18)

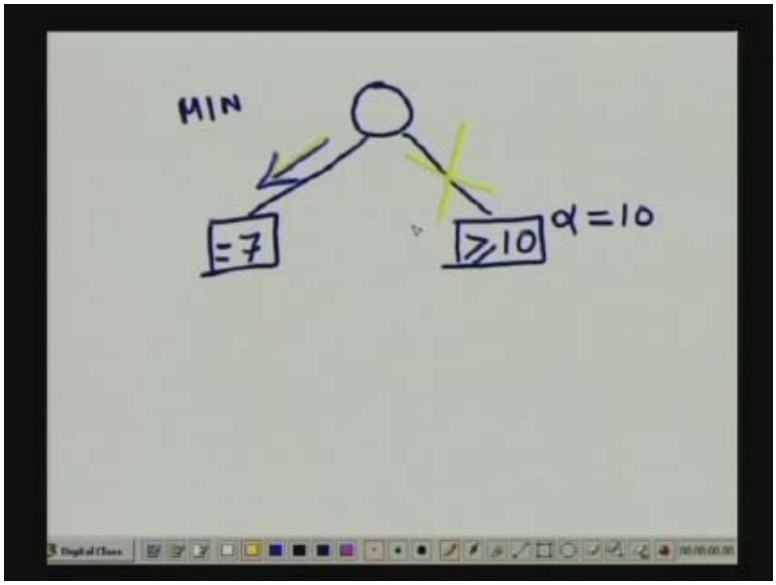


At a max node we will store an alpha value. The alpha value at the max node is a lower bound on the exact MINIMAX score of that node. So initially we can initialize the alpha values of all max nodes to be minus infinity. At any intermediate point the alpha value of

the node is a lower bound on its actual MINIMAX value. So alpha value of a node can only increase. So, if a max node has an alpha value equal to alpha its true value might be greater than equal to alpha. Now, if we know that min can choose moves with scores less than alpha then min will never choose to let max go to a node where the score will be alpha or more.

Let us just look at a diagram to explain what this is about. Suppose this is a min node. Suppose min has a path whose MINIMAX value is known to be equal to 7. Suppose min has another path to a max node whose alpha value is equal to 10 which means that its MINIMAX value is greater than or equal to 10. Now, since min is a minimizing node min knows that this path is better than this path because in this path the actual value is going to be greater than equal to 10. So search can be pruned at this path.

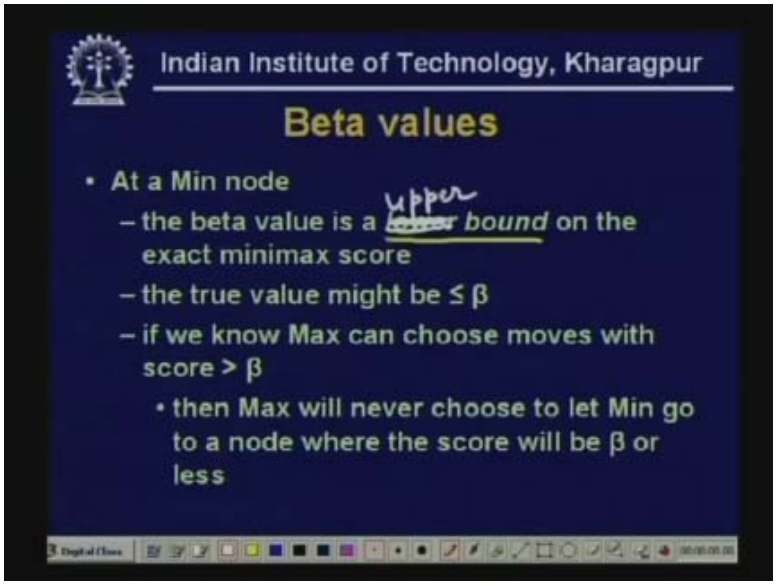
(Refer Slide Time: 13:37)



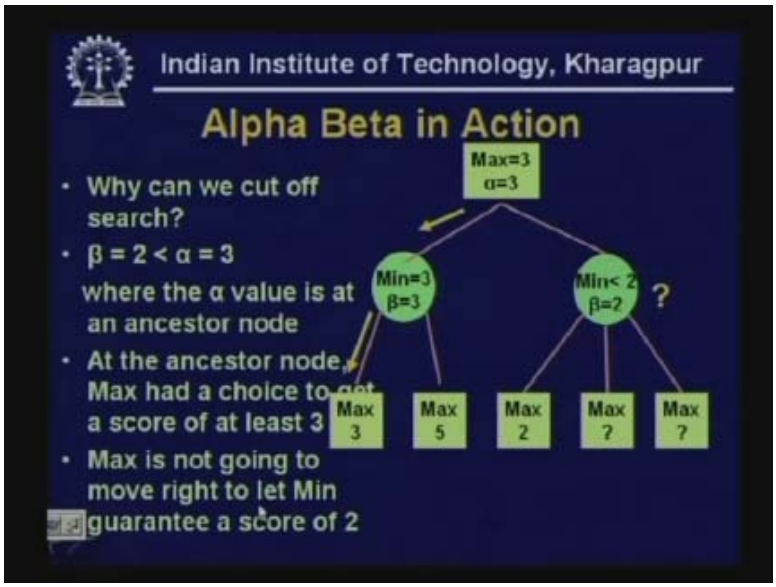
Similarly, the beta values can also be used for another type of cutoff. At a min node the beat value is actually an upper bound to the exact MINIMAX score. At a max node the alpha value is a lower bound and at a min node the beta value is upper bound. So the true value might be less than equal to beta because beta value is an upper bound. Now, if we know that max can choose moves with score greater than beta then max will never choose to let min go to a node for the score will be beta or less. Let us again draw a diagram.

Suppose this is a max node. Suppose at this min node max has a path whose MINIMAX value is known to be equal to 10 and there is another path so there is a min successor whose beta value is equal to 7. Since the beta value is an upper bound the actual value of this node is less than equal to 7 because max is a maximizing node and max will always prefer this path to his path so this path need not be explored any further. This is the sense of beta cutoff.

(Refer Slide Time: 14:09)



(Refer Slide Time: 15:34)

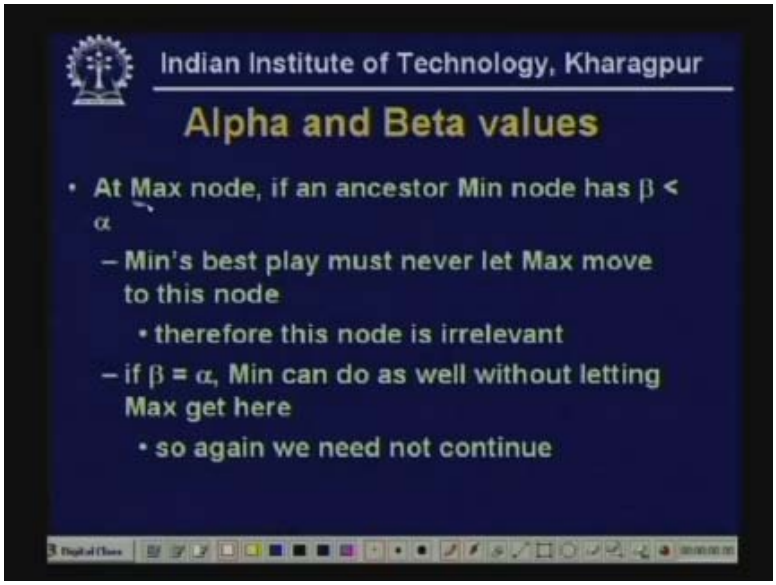


Now why can we cutoff search?
Let us consider this example:
In this example at this node the terminal value is equal to 3, at this node the value is equal to 5. So when this 3 is discovered min gets a beta value of 3. That is, the value of min will be greater than equal to 3. When this node is found then beta value remains 3 and the value of min is now exactly equal to 3, because the value of min is exactly equal to 3 we can set the alpha value of this node equal to 3. Therefore max value will be less than equal to 3.

Now, search comes to this node. This node has a value equal to 2. So because this node has a value equal to 2 the beta value of this min node is set to be equal to 2 and the actual value of min is less than equal to 2 Since beta value is 2 here and the actual value is 3 here max will never consider going along this path. So the rest of this path can be pruned off which means that these nodes need not be explored. And the optimum path for max is along this path and for min it is along this path.

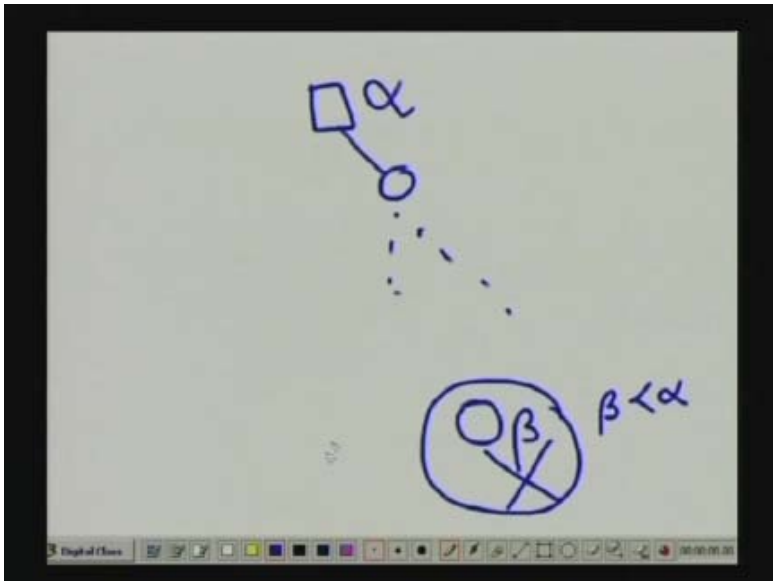
Now, if the alpha value at a max node is equal to alpha means that max can guarantee a score of at least alpha if he plays judiciously at that point. If the beta value of a min node is equal to beta it means min will guarantee a score no more than beta if it plays judiciously. So, this is the essence of alpha beta cutoff. More generally speaking, suppose we have a max node if there is an ancestor min node whose beta is less than alpha then this node is irrelevant so the ancestor need not be the exact predecessor it could be some other ancestor.

(Refer Slide Time: 18:15)



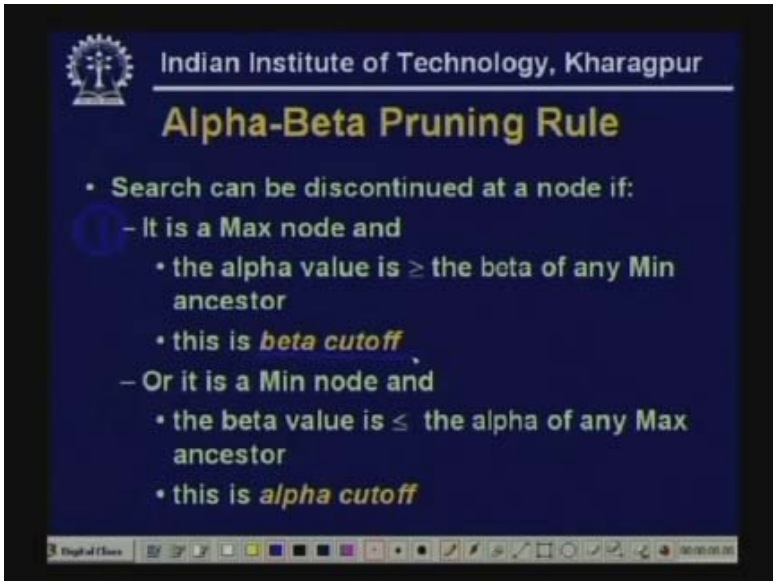
So just to draw a diagram, if this is a max ancestor and there is some parts of the tree and there is a min predecessor. If this max ancestor has an alpha value of equal to alpha and this node has a beta value of equal to beta and if beta happens to be less than alpha then max will never consider a move through this point. So below this node the rest of the node expansions can be cutoff. So, if a min node has a beta value which is smaller than alpha value of one of its max node ancestors then search can be cutoff at this node because a path through this node would be irrelevant for max because he has already found a path with a guarantee of at least alpha.

(Refer Slide Time: 19:56)



Two types of pruning can be done; one is from the point of view of max and another from the point of view of min. So search can be discontinued at a node under circumstances. Number one, if the node is a max node and the alpha value of this node is greater than equal to the beta value of any min node ancestor. This is called beta cutoff.

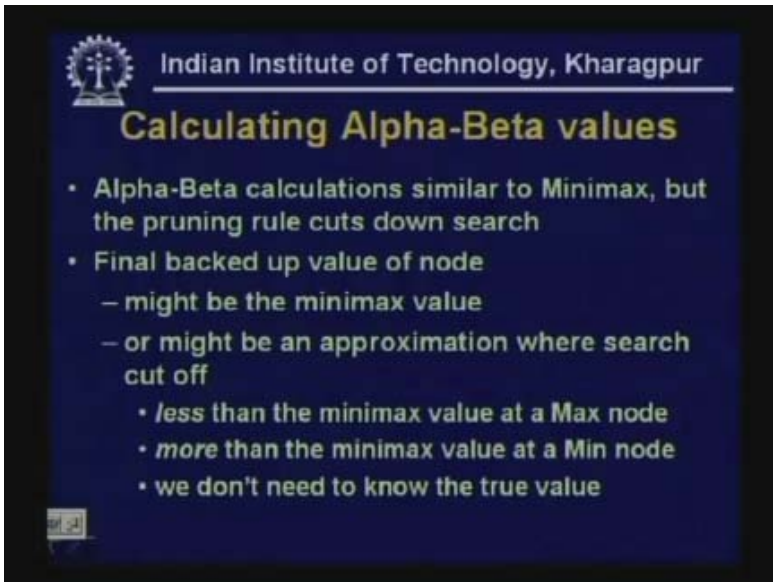
(Refer Slide Time: 20:35)



The other case which we just discussed is, if we have a min node and its beta value is smaller than the alpha value of any of its max ancestor then also search can be cutoff. This is called alpha cutoff.

Now, how to compute these alpha beta values?
We will outline an algorithm for computing the alpha beta values. The computations of these alpha beta values will be very similar to MINIMAX computations and they can be done in a depth first manner. However, while computing the alpha beta values we need not explore all the nodes of the search tree. And as a result of this computation we get a final backed up value of the nodes. For some nodes that we do not explore we do not care about the value. For those nodes where we do find a backed up value that backed up value may be the exact MINIMAX value or it may not be the exact MINIMAX value.

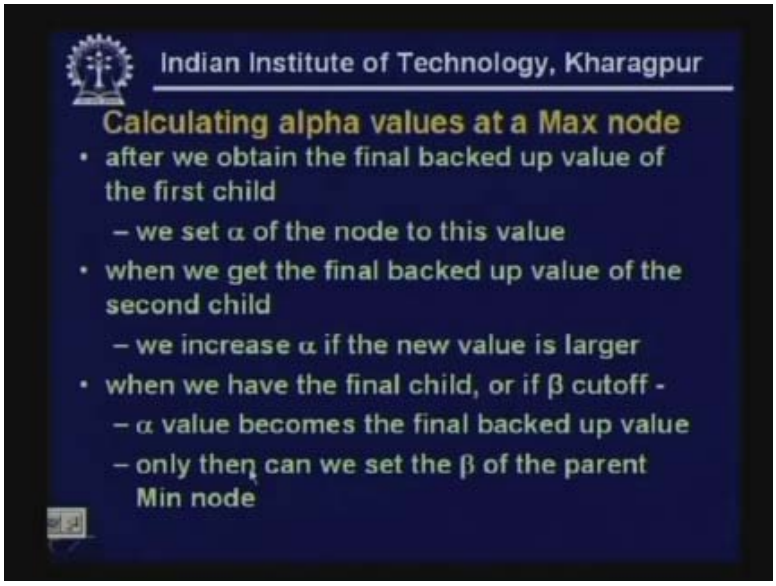
(Refer Slide Time: 21:52)



So, the backed up value is either the exact MINIMAX value of the node or it is an approximate value. And the alpha value of the max node is a lower bound on the actual MINIMAX value. The beta value of a min node is an upper bound on the actual MINIMAX value. And we do not really need to know the exact MINIMAX values to compute the best strategy for the max. Now let us see how we can calculate the alpha values at a max node.

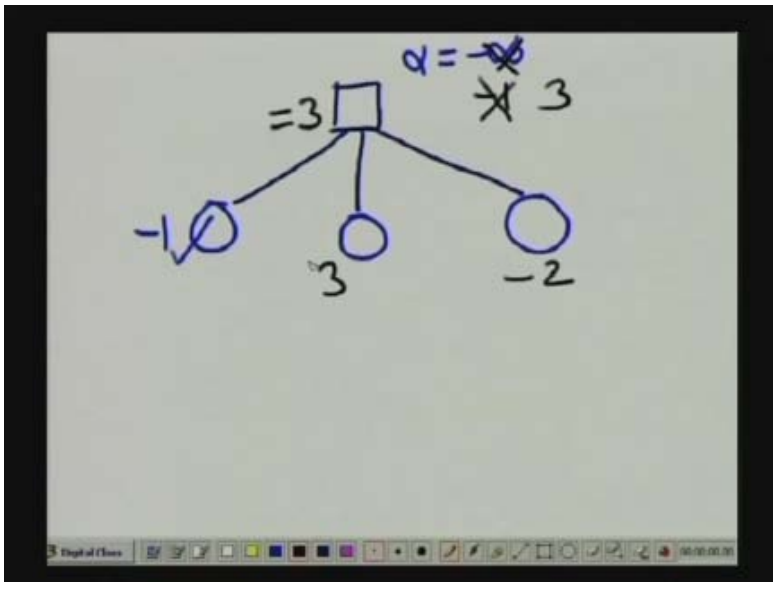
What we do is, when we are at a max node we first evaluate the value of one of its successors. Suppose this is a max node and suppose this max node has these three min children. So, when the exact value of the first child is computed. Suppose the value of this first child is minus 1 so initially the alpha value of this max node is minus infinity.

(Refer Slide Time: 22:36)



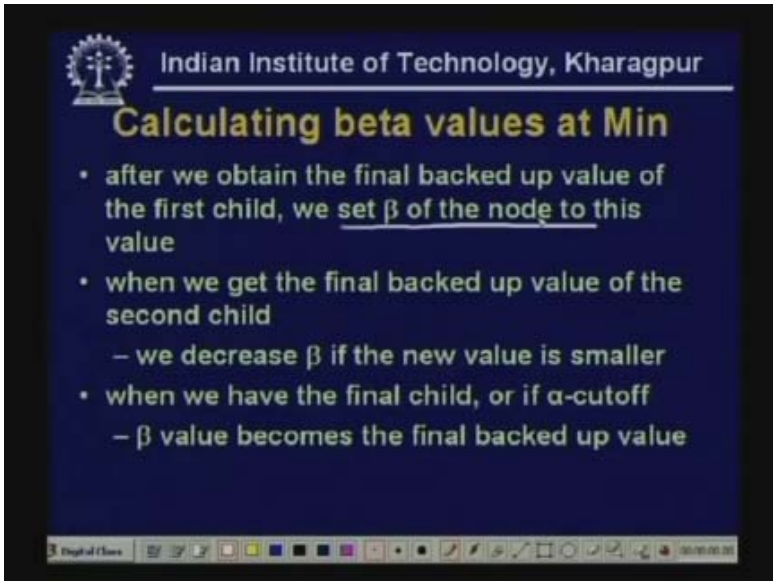
When this node is fully evaluated and its exact value is equal to minus 1 we set the alpha value to be minus 1. Then we evaluate the value the beta value of this min node. Or rather when we find the value of this min node if its value is 3 then the alpha value is updated to 3. If the value of this node is minus 2 because minus 2 is less than 3 the alpha value stays as it is. When all the children of the max node have been evaluated its alpha value is the exact value. But before that we can say that the alpha value is a lower bound on the exact MINIMAX value.

(Refer Slide Time: 24:42)



This happens when all the nodes have been fully evaluated. Even if one of its successors have not been fully evaluated but a beta cutoff has occurred we can terminate search at that point and find out the final backed up value at the max node as its current alpha value when we finish dealing with all its children or when a beta cutoff occurs at this node. Similarly and symmetrically we can compute the beta value at min node. After we obtain the final backed up value of the first child of a min node we set beta of the node to the value of this first child.

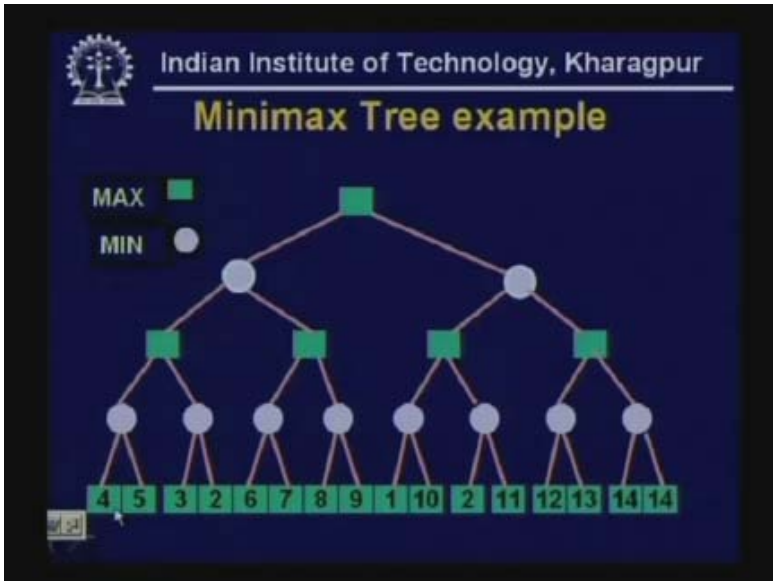
(Refer Slide Time: 25:38)



When we get the final backed up value of the second child we decrease beta update beta if the new value is smaller. When we have the value of the final child or if alpha cutoff occurs at this node then the beta value becomes the final backed up value of this node.

Now the question we have to ask is; how effective is alpha beta pruning? How many nodes does it prune?
Before we answer this question let us trace alpha beta pruning over a slightly larger tree and see how it works and then we will discuss how many nodes are pruned under different circumstances. This happens to be the schematic diagram of a game tree where the leaf nodes have been assigned certain values.

(Refer Slide Time: 26:57)



When depth first reversal occurs first this left most leaf will be evaluated. Its value is equal to 4. Therefore the beta value of this min node is equal to 4. That is, the value of this min node is less than 4. Now, after 5 this node is evaluated, the value of this min node will be equal to 4. So initially the value at this min node is less than 4, when this is evaluated the value at this mind node is equal to 4. Now, if the value at this node is equal to 4 the alpha value at this max node will be greater than 4. So we will put here greater than 4.

Now this child will be evaluated and to evaluate this, this child will be evaluated. When this value is evaluated the value of this min node will be less than equal to 3 and when this node is evaluated this value will become equal to 2 the final value will be equal to 2. And if the final value is equal to 2 the final value here will be equal to 4. If this value is equal to 4 the beta value of the min node will be equal to 4. That is, the value of this min node is less than 4. Now this child is going to be evaluated. To evaluate this this child to evaluate this this child so this node gets a beta value of 6.

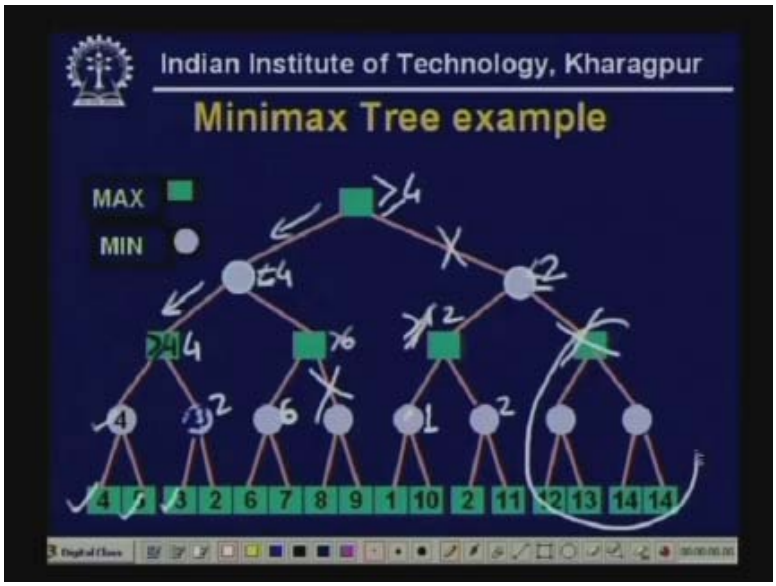
After this node is evaluated the final value of this node is equal to 6. Now, the alpha value of this max node will be equal to 6. That is, the value of this max node is greater than 6. Now let us look at this min ancestor. The value of this node is equal to 4. The value of this node is greater than 6. So this node does not need to look at this path. This path is clearly better than this path. So search can be pruned at this point and the backed up value here will be equal to 4.

Now since the backed up value of this is equal to 4 the alpha value of max will be equal to 4 that is the value of max is greater than equal to 4. Now this child, therefore this child, therefore this child, therefore this node will be evaluated. The beta value here will be 1 and after this is evaluated the actual value will be equal to 1. And then the alpha value of this node will be equal to 1.

Now let us come to this child. For this child we evaluate so the beta value of this node is equal to 2 so this node can improve further, so this is better than this so we evaluate this node and the actual value is equal to 2. Therefore the max will actually get a value of 2. Now let us just look at this. This min node now will have a beta value of equal to 2 so the actual value is less than equal to 2.

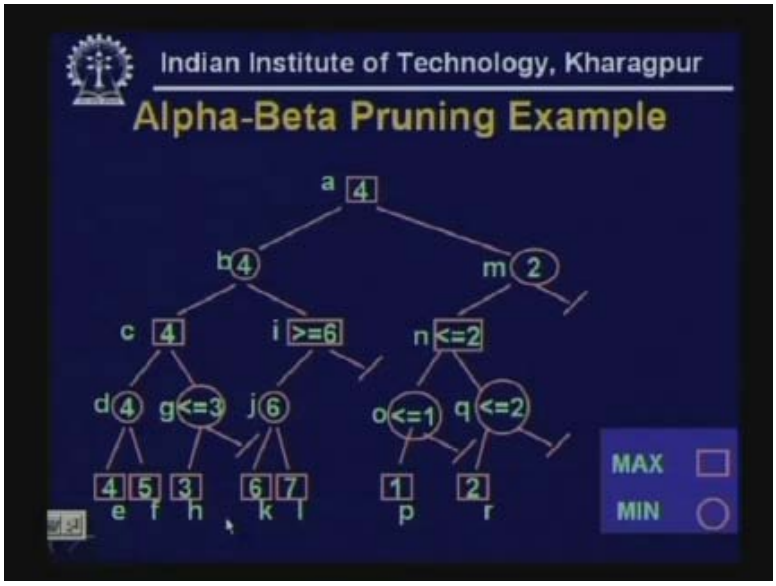
Now look at this max node. It has one child whose value is equal to 4 and this child has a beta value of 2 that is its actual value is less than equal to 2 so max need not consider this path and this will the best strategy for max at this point. Therefore search can be cutoff at these points. So these nodes need not be evaluated.

(Refer Slide Time: 31:34)



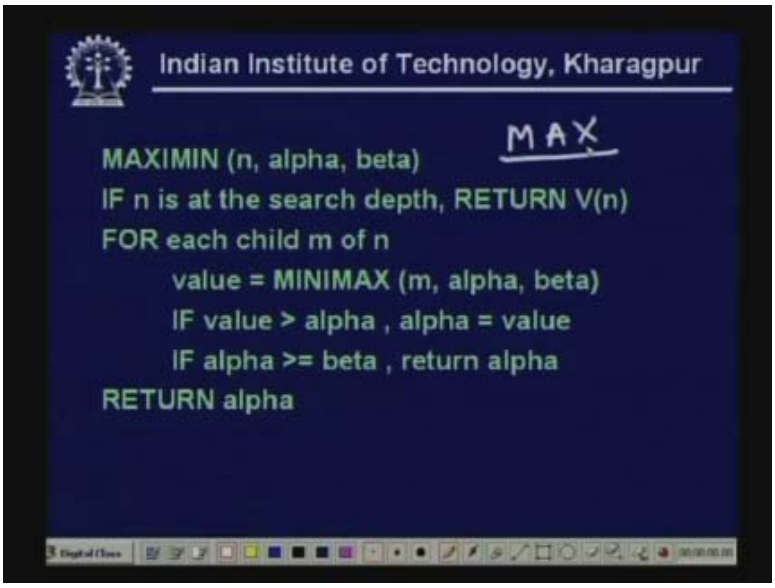
This is the actual the MINIMAX values of the nodes. But if we do alpha beta pruning this is how the nodes are evaluated. And as we can see these nodes were cutoff by alpha beta pruning and were not evaluated. And also this portion of the tree need not be evaluated when we use alpha beta pruning.

(Refer Slide Time: 32:22)



This shows the same diagram. So here we have some deep pruning at the right half of the tree. Now let us try to outline the entire MINIMAX search. So we assume MAXIMIN is done at a max node and MINIMAX is done at a min node and let us assume that vn is the static evaluation of node n. Now we outline the algorithm for MAXIMIN and MINIMAX. MAXIMIN is done at a max node.

(Refer Slide Time: 33:13)

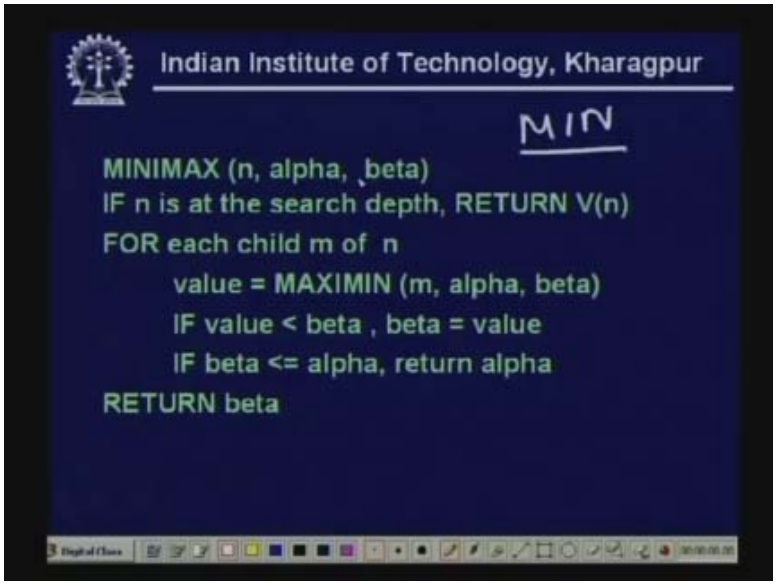


MAXIMIN takes as argument n the node and the current alpha and beta values. If n is at the search depth we have reached a leaf then we return the v(n). Otherwise for each child m of n each child of max will be a min node. Its value will be computed by recursively

computing MINIMAX m alpha beta. If this value returned is greater than alpha then we set alpha equal to value. So we look at one child, it returns a value and if the value is greater than the current alpha value of this max node we update the value of alpha. If the current alpha value happens to be greater than equal to beta where beta is the value of its min ancestor which has been propagated to this node in that case we will return alpha because we need not explore this max node any further. So this is the computation done at the max node.

We start at a max node, we find all its min successors, we expand the min successors one by one, we take the first successor find its MINIMAX value and if that value is greater than the current value of alpha then alpha is updated. And as a result if alpha becomes greater than equal to beta then alpha value is returned. Otherwise we continue until all the children have been solved and the alpha value has been obtained. And this alpha value is returned by the procedure MAXIMIN. At a min node we do MINIMAX which is just a symmetrical function. MINIMAX of a min node again takes alpha and beta.

(Refer Slide Time: 35:24)

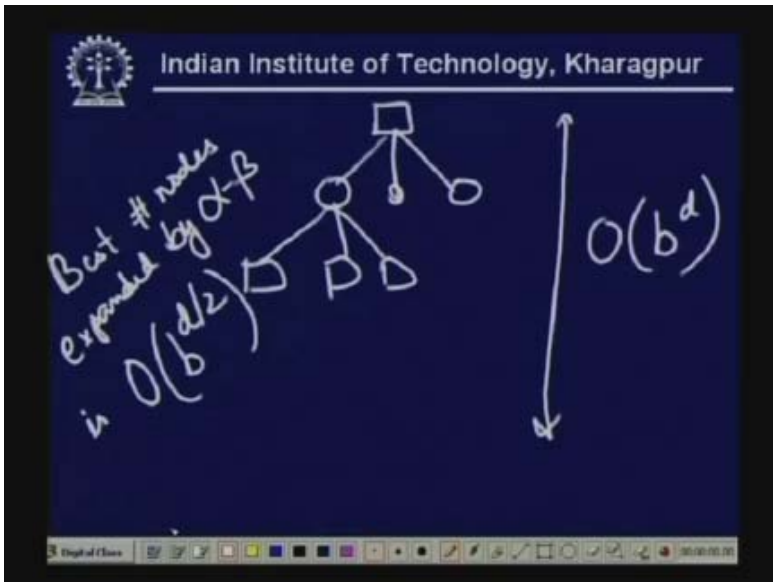


So beta is the beta value of this node and alpha is the alpha value of its max ancestor or max parent. So if n is a leaf node we return v(n). Then otherwise for each max child of n we find value of that child by calling recursively MAXIMIN. If the value returned is less than beta we update the beta value of this node. If this beta value happens to be less than equal to the alpha value of its max ancestor then we return the current beta value and then we cutoff search at this point. Otherwise at the end of this loop we return the beta value. This is MINIMAX.

Now let us look at the performance of alpha beta. When we have a tree whose branching factor is b and its depth is d we have seen that the number of nodes in this tree is of order b power d. It has been shown by Noth that in the best case the minimum number of nodes expanded or explored by alpha beta pruning is order b power d by 2 which is square root

of order b power d. And the best case occurs when the best child is always the first one which is explored for at every node. In this case when the best child is the left most child then that is explored first.

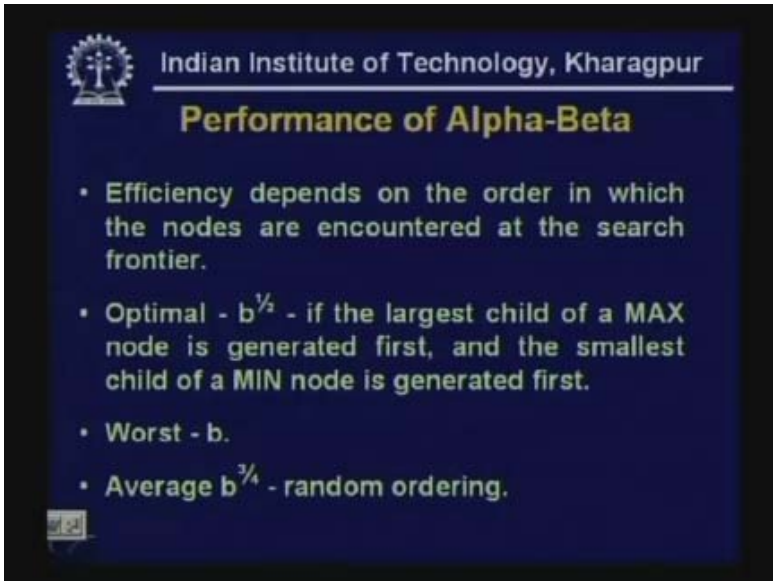
(Refer Slide Time: 37:45)



We get the correct value in the beginning and in that case the number of nodes expanded is smallest and it happens to be the square root of what would be expanded by MINIMAX search. The worst case not surprisingly occurs when the worst child is always the first one to be expanded and the best child is the last one to be expanded at every node. In that case alpha beat pruning would not have any extra savings over MINIMAX and all the nodes expanded by MINIMAX would be explored. So the number of nodes expanded is between order b power d by 2 and order b power d if we use alpha beta pruning. In practice for random ordering of the nodes you do, the average number of nodes pruned is quite significant to make alpha beta pruning very attractive.

However, we can improve our chance of achieving better pruning by doing move ordering. That is, when we start with the search we do not know which child is better. But if we use the static evaluation function of the children to decide the order in which the children are expanded. For a good evaluation function we can expect that a MINIMAX or with alpha beta pruning it will expand less nodes and prune more nodes. So the efficiency of alpha beta depends on the order in which the nodes are encountered at the search frontier.

(Refer Slide Time: 40:15)

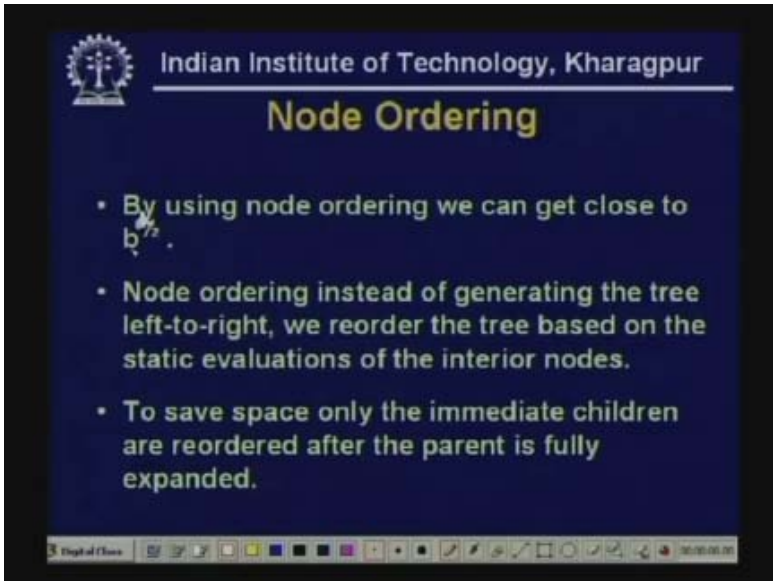


The optimal number of nodes expanded is order b power d by 2. This happens if the largest child of a max node is generated first and the smallest child of a min node is generated first. In the worst case the complexity is order b power d. And if you assume a random ordering then the number of nodes expanded is order b power ¾ d which actually means that due to our resource limitations if we use MINIMAX search we can go down to a depth of d. That is, if we can do a deep ply search with MINIMAX we would be able to do a 4 d by 3 ply search using MINIMAX assuming random node ordering. And if assuming very good node ordering we can search twice the depth and that would often lead us to have a much better strategy. So MINIMAX does improve the depth to which search can proceed due to pruning the number of nodes that need to be expanded.

(Refer Slide Time: 41:58)

Additionally a number of improvements have been developed to improve performance with limited computation. We will briefly discuss some of the other issues people have come up with for game trees. By using node ordering, using the static evaluation function or the current value etc we can get close to order b power by 2.

(Refer Slide Time: 42:32)



Or the effective branching factor would become root b. So node ordering instead of generating the tree left to right we reorder the tree based on the static evaluations at the interior nodes. So at every node we look at the children. We apply static evaluation on the

children and according to the value of the static evaluation we order the nodes. When we have limited resources available another idea is to do iterative deepening search.

Initially we do a search up to 1 ply and we get a possible value a suggested best move and then if we have time we do a 2 ply search and update our best move and then we do a 3 ply search then 4 ply search then 5 ply search then 6 ply search and so on. When the time is over, when there is no more time left we use the best move as given by the previous iteration. So we use iterative deepening and we use time, when time runs out the move recommended by the last completed iteration is made.

Iterative deepening can be combined with node ordering to improve pruning efficiency. Instead of using the heuristic value we can also use the backed up value from the previous iteration at the nodes. That is another idea one could do which can improve the pruning in go tree. There are two other points. In a game tree with fixed cutoff depth there are often some problems which people have noticed. One of these problems deals with the concept of quiescence. Quiescence happens when the play has settled down. And if you go down the search tree further it is unlikely that the evaluation function would change much.

(Refer Slide Time: 44:49)

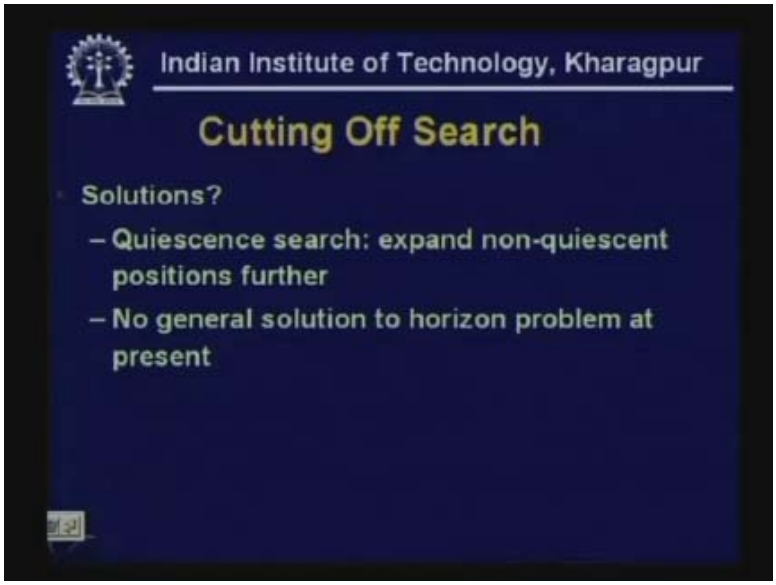
If we can recognize in the game tree that at a node it has reached a quiescent position we need not waste our effort in expanding that search frontier further. And we can concentrate our efforts in other parts of the search tree which are more likely to change the evaluation function as we go down. This effect is called the quiescence.

Secondly, often in game tree search we notice the horizon effect. Horizon effect happens when there is a particular set of moves so that there will be a dramatic change when a particular move is taken but you can use stalling operations to stall the inevitable bad or good position. In the game playing situation one can stall for time. They can postpone the

inevitable win or inevitable damaging move by the opponent over the horizon so that if we stop search at a particular point we might get an evaluation function which is dramatically different from what would happen if a move is taken to take care of that stalling.

What is the solution?
People even write game programs.

(Refer Slide Time: 46:42)



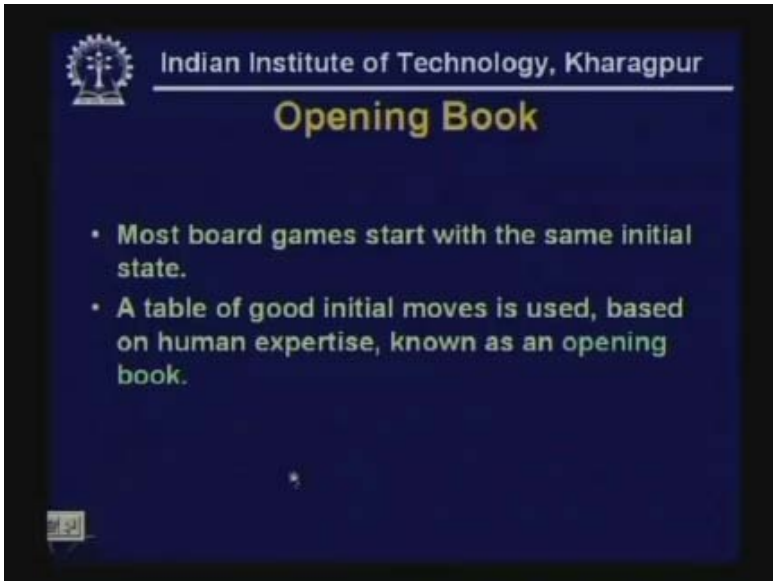
Quiescence search: instead of cutting of the search at a fixed depth the good game playing programs they evaluate those positions which need to be explored further because you expect a change in the backed up values if you explore the search tree. And there are some positions whose value is sort of closed and likely to change where you explore the tree further. So if you can detect the quiescent positions you can terminate search at those points. So you have a variable depth search tree. The horizon problem is not so easy to handle and there is no general solution so some heuristics could be used for some specific games. There are certain other strategies that people have employed. For example, sometimes transposition tables are used.

(Refer Slide Time: 47:45)

Some game trees or game graphs are more graphical in nature. There are many repeated positions. For efficiency, it is important to detect when a state has already been searched. In order to detect a search state previously generated game states with their minmax values are stored. This sort of transposition tables will be important if you look at game of Nim.

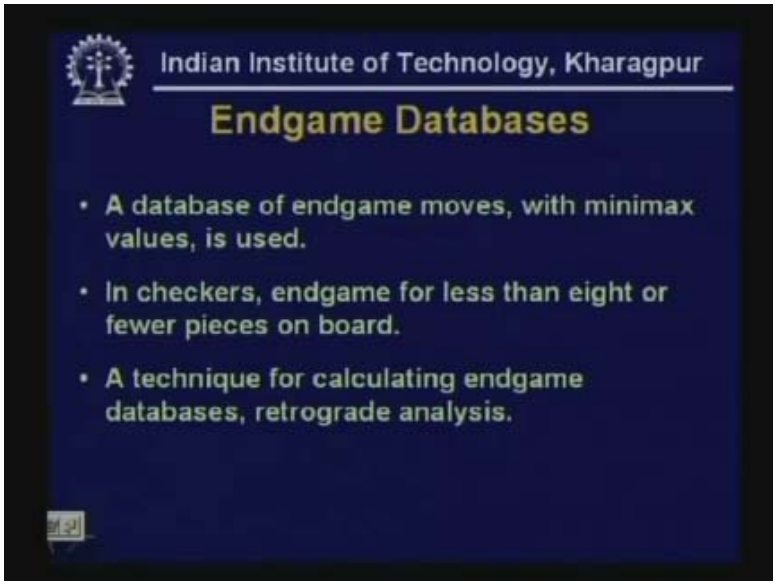
In Nim there are many repeated states so if a node has been solved it makes sense to store the state along with the backed up value obtained there to avoid repeated exploring of the same node. Then in many games the opening and the closing situation need very careful attention. Whether it is a game of chess or a game of backgammon it is of course a game of chance. Most node games start at the same initial position and there are certain best moves that you can take from the initial position. So, instead of using MINIMAX or alpha beta with MINIMAX to find the best move it makes sense to generate a table of good initial moves in consultation with the game expert human expert and this is known as the opening book, so opening book can be used.

(Refer Slide Time: 49:23)



Then the end game is another interesting place where there is a lot of expertise required. So database of end game moves will be very helpful.

(Refer Slide Time: 49:42)



So a technique for calculating endgame databases is also helpful. Then for playing more sophisticated games one needs special purpose hardware.

(Refer Slide Time: 49:51)

For example, in the chess game you can generate more number of moves per second if you have special purpose hardware. So the best machines today are based on special purpose hardware design and built to play specific games. And then do selective search.

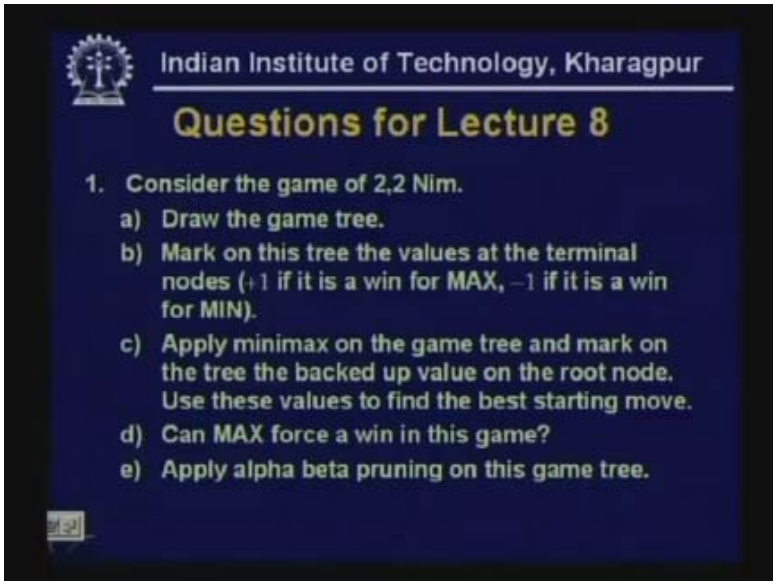
(Refer Slide Time: 50:13)

The motivation is that humans are competitive because when humans play games they do not look at all possibilities because they are not expected to do that. They are very good at selecting those promising possibilities. So selective search will identify the promising possibilities and search only on the interesting portion and we get best first MINIMAX. Let us learn about the game of m, p Nim.

(Refer Slide Time: 51:06)

The m, p Nim is a stones game where we have a matches game, there are p piles of m matches in each pile. To move a player will remove any number of matches from exactly one pile and the winning player is the one who removes the final match. This is the game of m, p Nim. So the first question for this lecture asks you to consider the game of 2 2 Nim. That is, you start with two piles each containing two matches. So you have to draw the game tree.

(Refer Slide Time: 51:41)



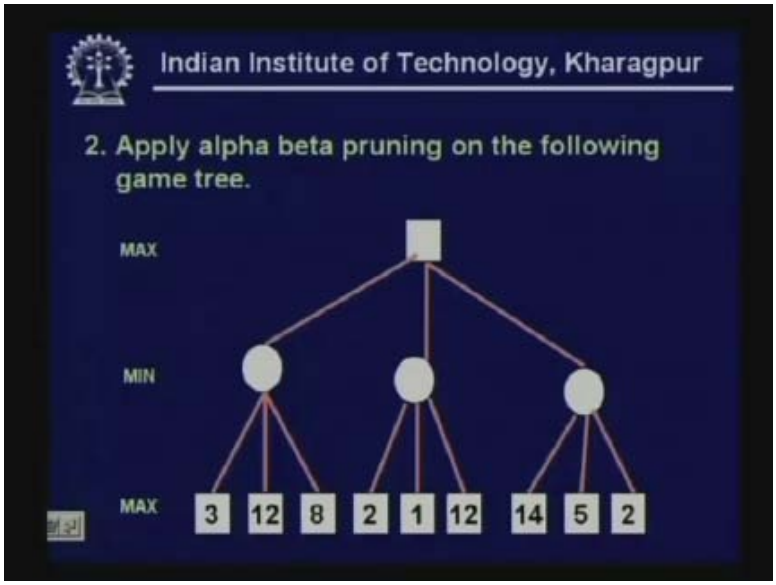
b) You mark on this tree the values at the terminal nodes. The terminal nodes will have a value of plus 1 if it is a win for max and minus 1 if it is a win for min.

c) Apply MINIMAX on this game tree and mark on the tree the backed up value on the root node and use these values to find the best starting move for max.

d) Can max force a win in this game?

e) Find out if alpha beta pruning on this tree gives rise to pruning some parts of this particular search tree.

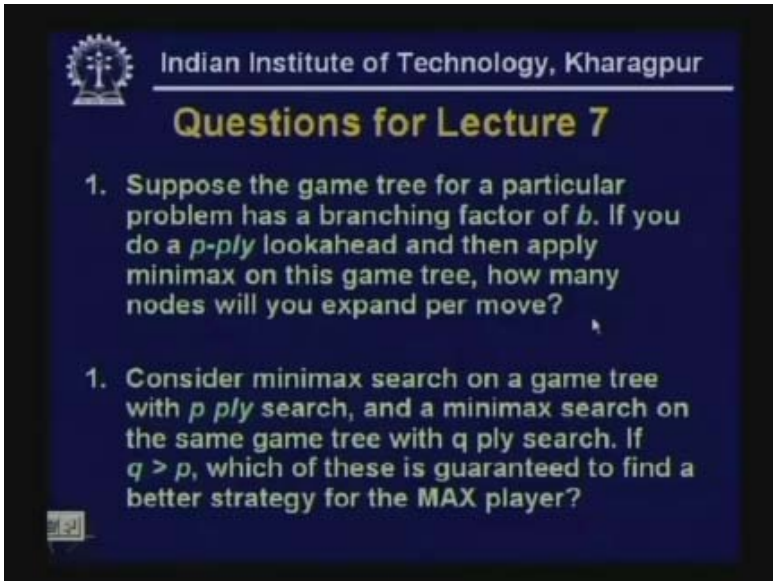
(Refer Slide Time: 52:36)



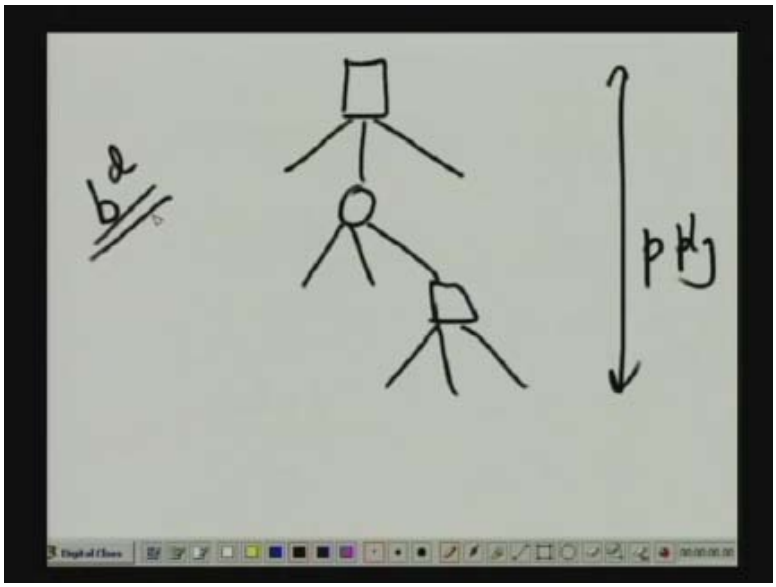
2) You are given this game tree. You are going to apply MINIMAX with alpha beta pruning on this game tree and identify the best possible move for max.

Question number 1 for lecture 7 was:
Suppose the game tree for a particular problem has a branching factor of b.
If you do a p-ply look ahead and then apply MINIMAX on this game tree how many nodes will you expand per move. This is actually quite easy. You have a tree whose branching factor is b and the tree is to the depth of p-ply. So this is p-ply. So, for every move you need to do a p-ply full MINIMAX search where b power d nodes are expanded. For each move you need to expand b power d nodes.

(Refer Slide Time: 52:55)

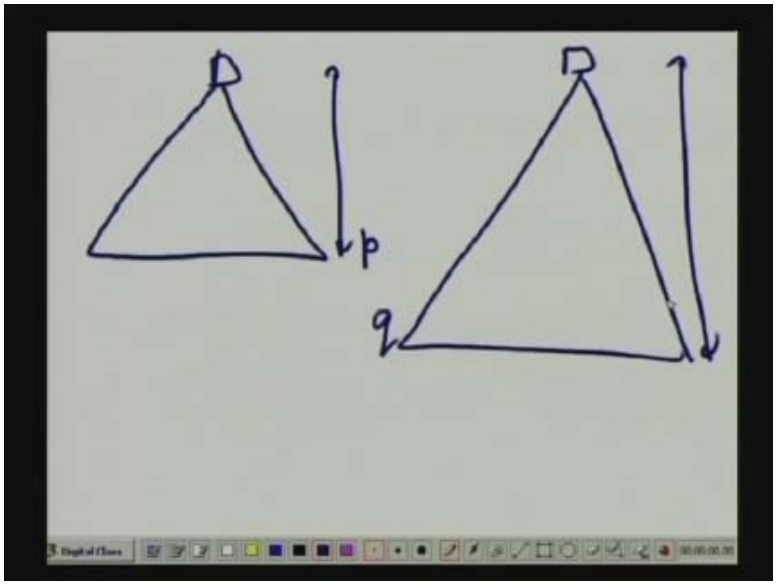


(Refer Slide Time: 53:58)



2) Consider MINIMAX search on a game tree with p-ply search and another MINIMAX search on the same game tree with a q-ply search. If q is larger than p which of these searches is guaranteed to find a better strategy for the max player?

(Refer Slide Time: 55:00)



So the question is, if we do a p-ply search and find the best move in those situations and we do a q-ply search and then again find the best moves of find the MINIMAX value which of these searches gives rise to a better strategy?
The answer is we normally think that a q-ply search is better than a p-ply search if q is larger than p. And the intuition is that, we think that if we go deeper down the tree we will have more accurate evaluation functions as we are nearer to the terminal positions.

If we are at a terminal position then we know the correct value of a node. And we assume that if we are nearer to the terminal position we have a better estimate of the correct value. So we expect that the leaves evaluated for a q-ply search will have a better estimate and based on these evaluations when you do MINIMAX we may find a better strategy than what we would find in at search having p-plies. However there is no guarantee here. So we cannot guarantee that a q-ply search will be larger than a p-ply search even though we expect a q-ply search to be better. So the answer is, it is not guaranteed that one of them will be better than the other even though the q-ply search is expected to be better than p-ply search.