# ASSIGNMENT 3 HARSH SAINI 1001571136

**Answer 1**.

For simple games like Tic-Tac-Toe, state space is small enough to exhaustively search for winning states. In such a case, the Max player can always force a win regardless of Min's first move. Min can win only when Max plays unoptimally without any common sense.

* While for complex games like Chess, state space to search is very large and it is computationally very expensive to search entire tree. If we assume that the depth of state space tree is m and there are b legal moves to search for winning states at each point, then the complexity of this algorithm will be O(bm). Hence, depending on the context, it is possible that algorithm doesn't terminate in looking for best move.

**Case 2: Alpha-Beta pruning:**

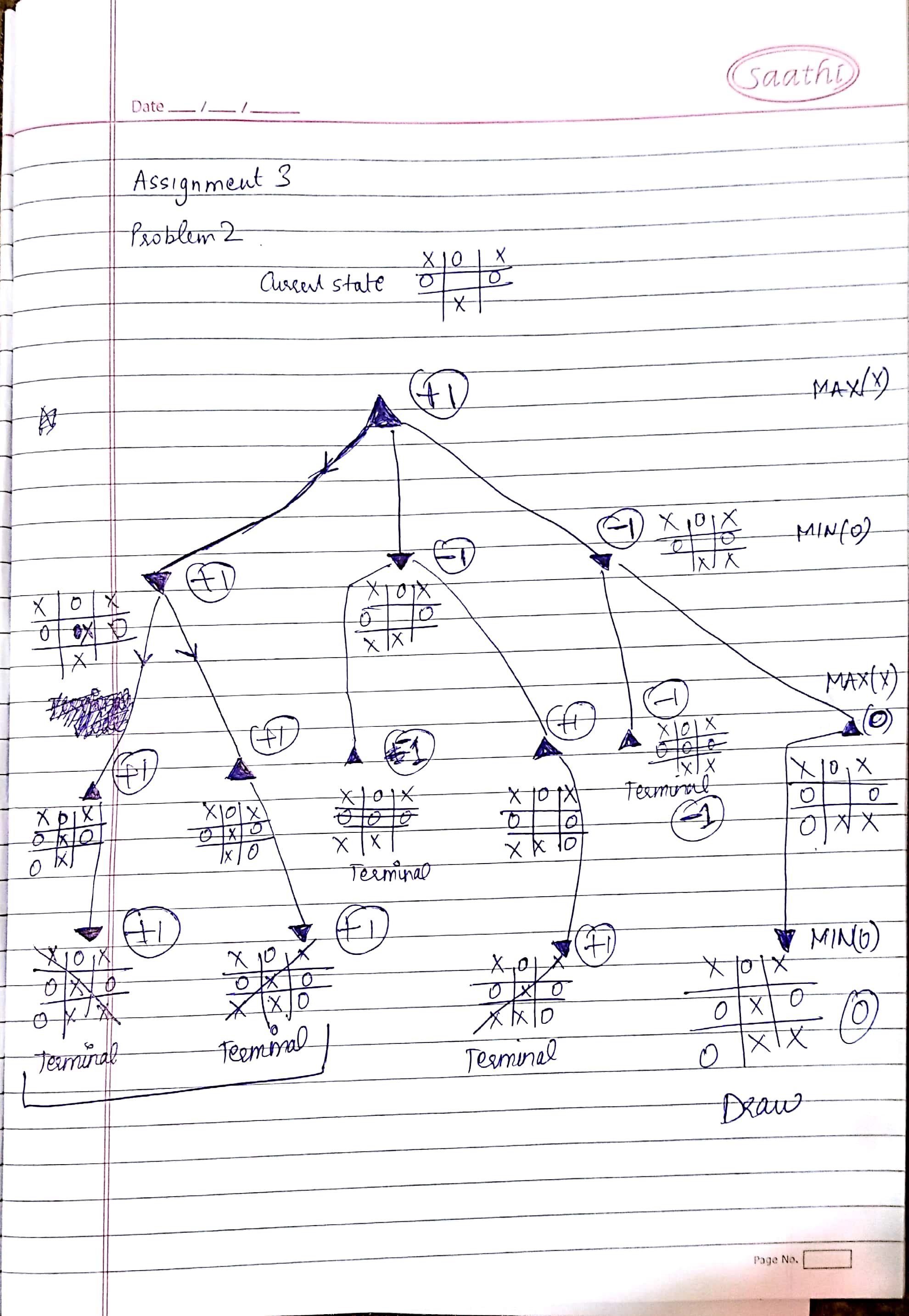
It returns the same move as standard minimax, but prunes the branches not leading to optimal (winning) states.

* It can be applied to trees of any depth (small like Tic-Tac-Toe as well as big like Chess) and it is often possible prune entire subtrees rather than just leaves. The procedure updates the values of Alpha and Beta as it goes along and prunes rest of the branches at a node (termination) as soon as the value of current node is found to be worse the current Alpha or Beta for Max or Min.
* It can be shown that this procedure will always look for absolute best move in any given position first. And the effectiveness of this algorithm will depend on the depth of the tree.

**Case 3: Depth-limited search:**

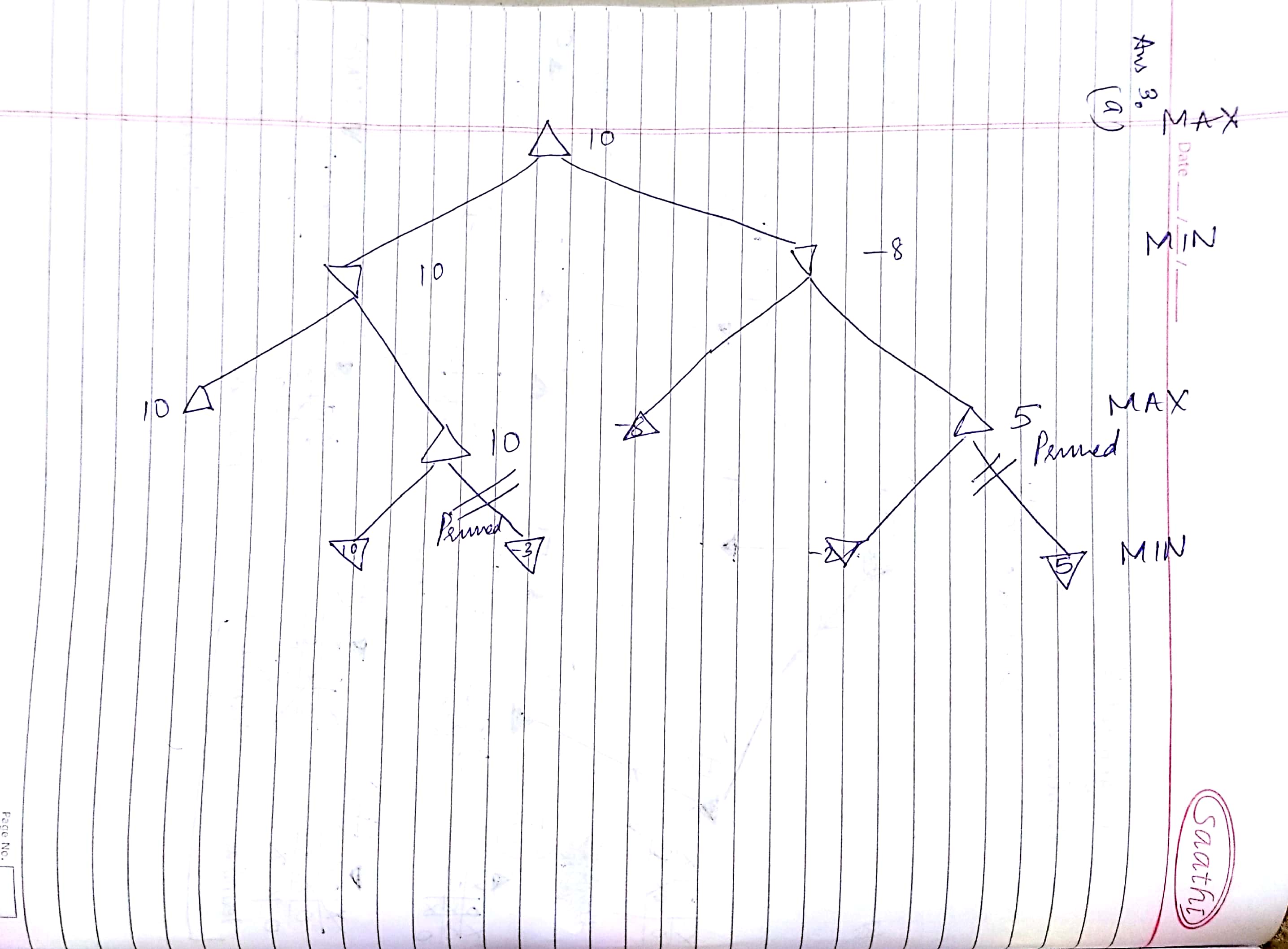
* The evaluation of advantageous or non-advantageous state is based on some heuristic. When this heuristic is applied with a limited look-ahead, then depth of look ahead may not be able to deduce that a promising path may lead to a bad situation later.
* And if such a path is followed further it is possible to lose the entire game. For this reason, usually larger depths deeper from good looking states are explored. But this may lead to a situation when procedure looking for a best next move never terminates. But search has to stop somewhere as per limited depth strategy. And an after effect of this strategy is that search will not be able to find out good looking states (ultimately winning states) beyond the permitted depth. But this is a trade-off and ensures that algorithm is terminated.

ANSWER 2.

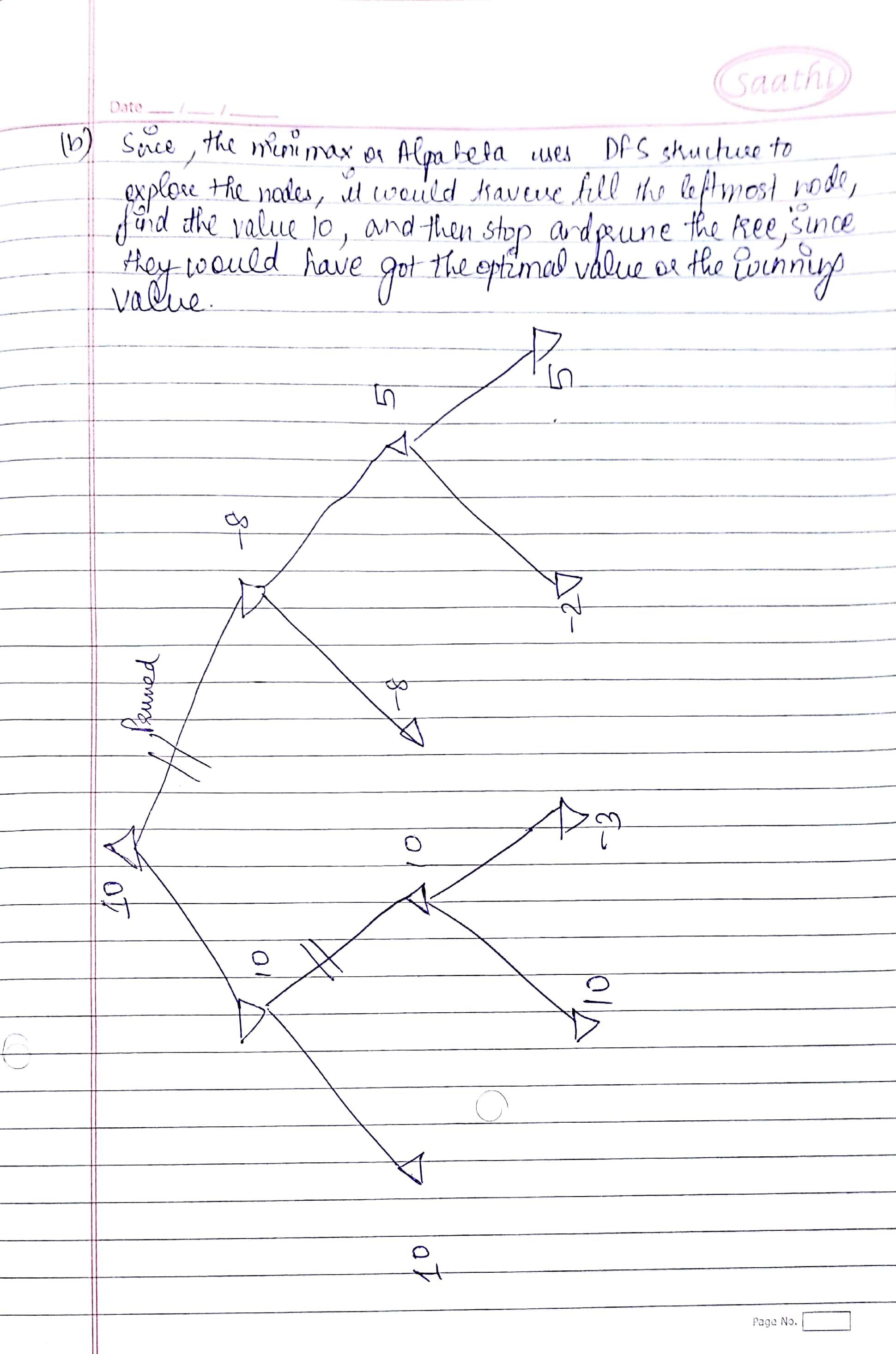


ANSWER 3.

1. The Algorithm prunes the tree as shown in the figure below.



1. The Minimax or Alpha beta uses DFS Structure to explore the nodes. Hence, the deepest node on the left would be traversed which has optimal value. This would prove that this would result in a winning move therefore pruning the rest of the tree.



ANSWER 4.

**Minimax simple version:** If Max plays first, when both are using the same plain version of minimax, there could be no doubt that Max player can easily draw or win over the opponent if he plays optimally since due to the limited amount of state space and the only case where min could win could be the wrong move of Max.   
Briefly: Playing vs a perfect player would always be a draw and against a regular player would always result in either a win or a draw.

**Using a Modified Version or Other than Minimax:** When both aren’t using the same algorithm, we can determine a function that brute forces the complete tree to reach the optimal or trace the winning path. Since, the minimax determines on heuristics which themselves can be misinformed, using a brute force algorithm can ensure to decide the winning path. Hence, by adding brute force.

ANSWER 5.

* As we know Minimax algorithm always uses the best possible search strategy assuming the opponent is always rational and always optimizes its behavior according to ours. Then the minimax algorithm determines the best move anyhow under any condition.
* The win however would depend on the strategy of the player and the current state of the we are in. If both play their optimal moves, then the game would lead to a draw.
* **If Min plays a bad move, this would result in a win.**
* **If Min plays optimally, it would result in a draw.**

ANSWER 6.

Explanation: The below algorithm uses the current state to know about the response of the algorithm of the opponent. We can ensure our victory or draw by using its value to ensure we play the optimal move always.   
We will easily know about its move and if it plays any move that was out of bounds of our knowledge, we will use it to append our heuristic so that we remain optimal always(hypothetical case but is useful to ensure).

MinMax(GamePostion game){

Return MaxMove(game);

}

MaxMove(GamePosition game){

If(GameEnded(game)){

return EvalGameState(game);

}

else{

best\_move<-{};

moves<-GenerateMoves(game);

For Each moves{

Move<-MinMove(ApplyMove(game));

If(Value(move)>Value(best\_move))

Best\_move<-move;

}}

return best\_move;

}}

MinMove(GamePosition game){

best\_move<-{};

moves<-GenerateMoves(game);

check\_DeepGreen(moves);

ForEach moves{

Move<-MaxMove(ApplyMove(game));

If(Value(move)>Value(best\_move) && Value(moveFromstatebyDeep)<Value(Best\_move))

Best\_move<-move;

}

else{

updateheuristic(moveFromstatebyDeep);

//MinMove(game);

}}

return best\_move;

}

DeepGreen(State s){

Return moveFromstatebyDeep;

}

updateheuristic(betterMove){

//this will depend on the game rules and heuristics given

//we calculate why was this chosen and update the value at node either to update heuristic or to find a new //path which was either hidden due to limited horizon or any other unforeseeable factor.

}