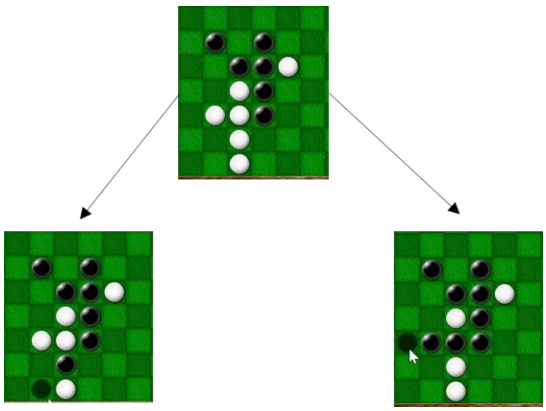
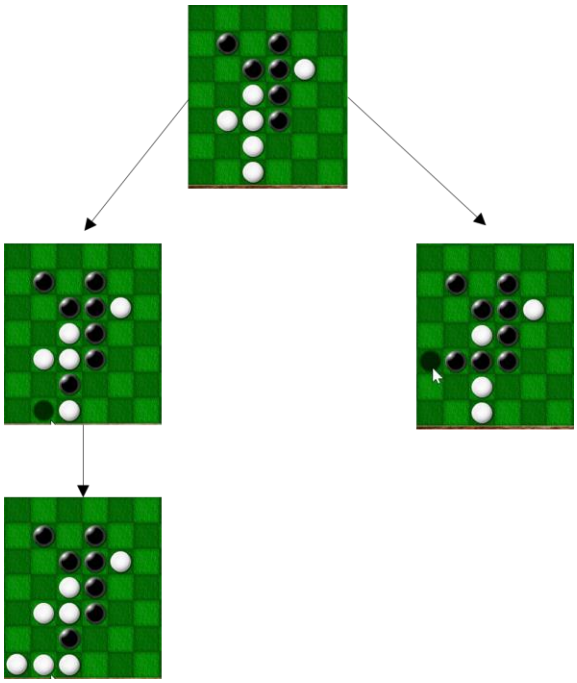
AI Game Playing – Minimax Algorithm

The minimax search algorithm is a recursive algorithm commonly used in the context of two player deterministic strategy games, like chess, checkers, tic-tac-toe, etc. There are lots of resources online that discuss minimax (mostly in the context of tic-tac-toe!)

The basic idea is pretty similar to how humans would play these games. When it is your turn to make a move you consider locations to place your piece. For each of these locations you would consider places your opponent could then place his or her pieces, and so on. For example, in the Othello board below, if it is black's move we might consider the two moves below (there are others as well not shown):



If we consider the move on the left, next it would be white's turn. So we can think an additional move ahead. For example, white could move in the corner and erase our newly placed piece.



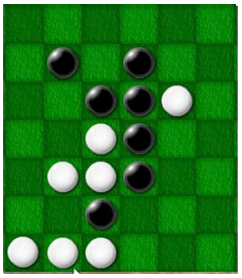
If we mentally generate such a sequence of moves it is called a **search tree**. The tree is upside down, with the root at the top. Note that the tree branches out quickly from the root of the tree. If we had a small, trivial game, then we could search ahead and determine what move will eventually lead to a win, loss, or tie. When this is possible, the term is that we have "solved" the game.

However, for most games of interest, it is not realistic to generate this tree to determine when we win or lose. The branching factor and depth of most games is too high. The branching factor refers to the width of each level in the tree, which corresponds to the number of moves that are possible. In many games, the branching factor is about 35, and the number of pieces that can be moved is about 100 (the depth of the tree). 35^{100} too large to search! That is on the order of the number of atoms in the observable universe…

Some terminology: A **move** is usually considered to be when each player takes a turn. The figure above would be considered one move since black has played and white has played. When only one player takes a turn, this is called a **ply.** In othello, a ply is when each player places a piece and represents a level in the search tree. In upthrust, a ply is when a player moves a piece.

Since the search space is usually too large to exhaustively search all the way to the end of the game, what are we to do? The solution is to search as far as we can in some reasonable amount of time and then apply a **heuristic function** to each board state. The heuristic is our best-guess or estimate of how favorable the game board is for us. If we win the game the heuristic might be a large positive number. If we lose the heuristic might be a large negative number. If we have a lot more pieces than the opponent maybe it is a smaller positive number.

A simple heuristic for Othello is to subtract the number of opponent pieces from the number of our pieces. The more piece we have, the larger the number. If our opponent has more pieces, then it is a negative number. This function gives us a number to estimate how favorable the game state is. It's not perfect though, as it is possible there is some sequences of moves that might flip a bunch of pieces, so a couple of ply later what was once a positive heuristic value might result in a negative value.

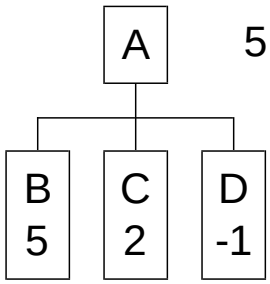


H = 7 – 7 = 0

We will discuss example heuristics in more detail later.

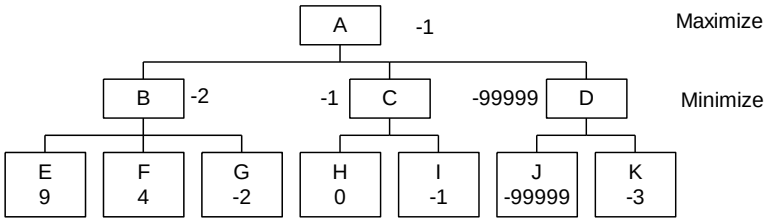
We can put this together to generate the Minimax algorithm. We look ahead from the current game state as many moves as possible. In our case we are limited by memory and time. We'll just pick a maximum search depth. Apply the heuristic function to these positions and choose the best one.

For example, consider a 1-ply search. The letters represent game states and the numbers represent the evaluation of the heuristic function to the board:



Starting at board state 'A' we can make three moves to either state B, C, or D. Applying the heuristic to each of these states results in the values shown. Large values are good, negative values are bad. It looks like choice "B" is the best, so node A would get the value of 5 as the best and select B as the move. A is selecting the **maximum** value from the values returned to it.

What if we want to look ahead another ply? We have to take into account that the next move will be our opponents move. Instead of picking a state with a high value, we will assume our opponent will use the same heuristic function as us and pick the state leading to the smallest value. This is called a **minimizing** move; when it is our move, this is called a **maximizing** move.



In the example above, if we search one more ply and then apply the heuristic function, we get different results. If we assume that the opponent will choose to pick the move resulting in the smallest heuristic value, then we have to propagate back the minimum of the children at a min node. State B turns out to be not so good due to G, while C is slightly better. Move D turns out to lead to a lost game for us at state J.

Note: Minimax assumes an opponent as smart as we are. Sometimes we may want to make a move like B, and hope that the opponent won't choose G, but will instead choose E or F. Since G is close to I, but there are better other moves at B, we could take move B and hope the opponent will pick one of the other moves.

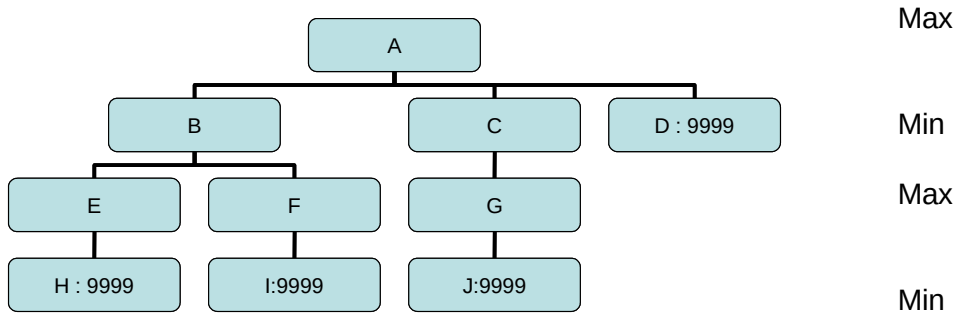
We can now describe pseudocode for the algorithm (an actual implementation will need additional variables to account for things like whose turn it is):

```
Function minimaxDecision() returns Move
        moveList ← moveGenerator(gameState)
        for each move M in moveList do
                value[M] ← minimaxValue(applyMove(M, gameState), 1)
        return M with the highest value[M]

Function minimaxValue(state, currentSearchDepth) returns a heuristic Value
        if currentSearchDepth == desiredDepth or terminal(state) then
                return heuristic(state)
        else
                moveList = moveGenerator(state)
                for each move M in moveList do
                        value[M] ← minimaxValue(applyMove(M, state),
                                                currentSearchDepth+1)
                if whoseTurn==myTurn then
                        return max of value[]
                else
                        return min of value[]
```

This algorithm assumes that the heuristic function doesn't flip for the opponent. Some algorithms assume that it does.

Note: You may want to explicitly check for the WIN state; if you can win, make the move immediately. Consider the following:



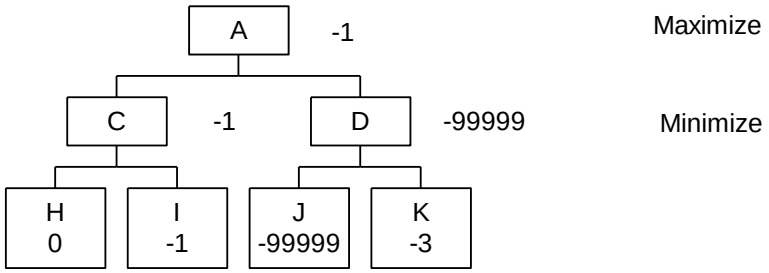
If the maximizing player takes the move to B, then looking ahead we see we are guaranteed a win. But we could directly win if we make move D. The downside is we may take the move to B instead, which could potentially result in an infinite loop if this

type of move is repeatable. A similar problem occurs if we don't check and exit for win states (the maximizing player may then go for multiple-win states if that increases our heuristic value)

Optional Material if you want to make a better AI player: How to search deeper in the search tree Alpha-Beta Cutoffs

A strategy called alpha-beta pruning can significantly reduce search time spent on minimax, allowing your program to search deeper in the same amount of time. In general, this can allow your program to search up to twice as deep compared to standard minimax. The modified strategy also returns the exact same value that standard minimax would return.

Consider the tree below:

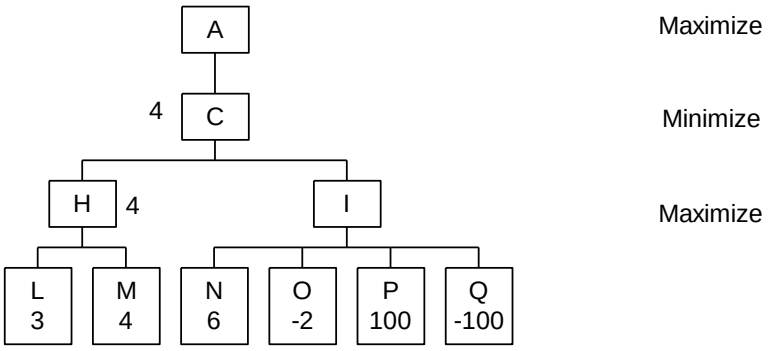


The minimax algorithm will search the tree in a DFS manner: A to C to H, to I, A to D to J, then A to D to K. Notice however, when what happens when we're at node D and have just examined J. We know that A wants to maximize its move. It can already get a value of -1 by making move C. D wants to minimize; it can choose at LEAST -99999, and maybe less. So we don't even have to examine node K, because there is no way A will pick branch D, since D can choose a value < -1.

This is called an alpha prune; on a min node, we came across a value LESS than the current max. At this point we can stop searching because we won't want to make a move where the opponent can force a worse board state.

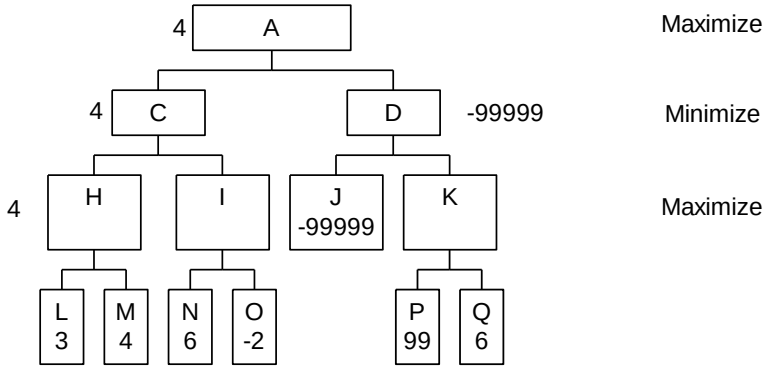
We can also do the opposite prune, a beta prune; on a max node, if we come across a value GREATER than the current min, then we can stop searching because our opponent won't want to make a move where we can get a better board state.

Here is an example of a beta prune:

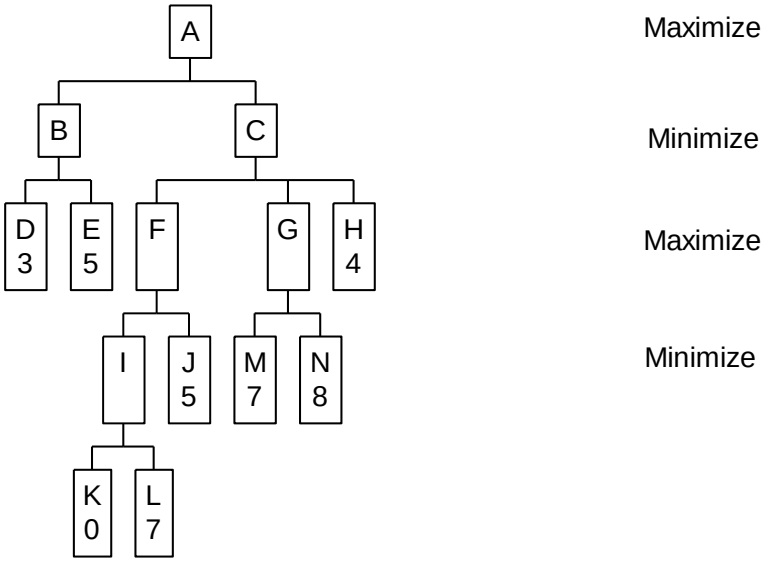


In this case, after examining node N we don't have to examine nodes O,P, or Q at all, since we can choose a value of at least 6. However, the opponent can limit us to 4 by choosing move H. So moving to I would be bad and any other values can be discarded.

Here is another example with both an alpha and a beta prune:



One more example:



At min nodes, compare to alpha (max so far). At max nodes, compare to beta (min).

Can prune out nodes L and N. Why? What is the final backed-up value? Note that in these examples only a few nodes were pruned, but if the pruned nodes are entire trees, then a significant amount of pruning can be achieved, enough to search several layers deeper in search.

Algorithm: Call with Max-Value(state, -MaxValue, MaxValue)

```
Function Max-Value(state, alpha_max, beta_min) returns pair: minimax value of state, move
        If current_search_depth == Desired_Depth or OutOfTime or Terminal(state) then
                Return Heuristic(state), any move
        Move_List = MoveGenerator(state)
        BestMove ← Move_List[0]
        For each move M in Move_List do
                Value ← Min-Value(Apply_Move(M,game_state), alpha_max, beta_min)
                If Value > Alpha_max then
                        Alpha_max ← Value
                        BestMove ← M
                If Alpha_max >= Beta_min then return Alpha_Max, BestMove
        Return Alpha_Max, BestMove

Function Min-Value(state, alpha_max, beta_min) returns minimax value of state
        If current_search_depth == Desired_Depth or OutOfTime or Terminal(state) then
                Return Heuristic(state)
        Move_List = MoveGenerator(state)
        For each move M in Move_List do
                Value ← Max-Value(Apply_Move(M,game_state), alpha_max, beta_min)
                Beta_min ← Min(Value, Beta_min)
                If Alpha_max >= Beta_min then return Beta_Min
        Return Beta_Min
```

In this algorithm, we only care about the best move from the initial invocation to Max-Value. The rest of the algorithm only requires the heuristic values. Consequently, the BestMove code is only present in the Max-Value routine.

Some strategies to increase speed:

1. Search the potentially best moves first. If you can find good moves early on, these moves help prune more branches.
2. Prune search tree by eliminating some moves that are obviously bad to make.
3. Copying board may be time consuming; consider incremental approach, undoing moves on the same board. If you are copying a board, consider using memcpy instead of a loop copying each element.
4. Use opening library of book moves, if you have analyzed and found opening moves. In book moves, you store an exact board state and the move you have determined to be best for that state. If the state arises, make that move. Use a hash or lookup table to retrieve the moves quickly.

5. Perform search while opponent is moving (predict what move they will make, do squashing). This is not allowed in the tournament, but if you can get it to work I will give you extra points for the project!
6. Optimize your code for speed wherever possible ☺

Alternatives to Minimax?

Some alternatives so far have been:

1. Rule-based systems. These essentially do no search or lookahead but rely on a large number of rules to make the move. This is analogous to a complex heuristic function and doing only one move lookahead.
2. Machine learning systems such as neural networks. The machine plays many games, adjusting its weights and parameters of what makes a move good; Tesauro has implemented this technique into a world class backgammon program.

Hints on Designing Heuristic Functions for Board Games

Ideally you want a heuristic to give an accurate measure of how far away you are from a win. This can be very difficult for some games, especially ones where the board state may change radically in a single move. For example, in Othello it is possible for many game pieces to flip color in a single move.

Here are some simple sample heuristics used for various games:

Othello:

A surprisingly good heuristic is just the ratio of pieces: Mine/His, or the difference in pieces, Mine - His. Since the goal is to win by maximizing your pieces until the board is full, this works very well. Strategic locations are usually given a higher weight. For example, the corners are very strategic since they can never be captured again. The sides are slightly less strategic, and pieces in the middle even less strategic. A better scheme is a weighted heuristic based on board placement:

Value = A*#corner_pieces + B*#edge_pieces + C*#other_pieces
Heuristic=Value(Mine)/Value(His)

In general, this approach is very common. A weighted heuristic based upon different features of varying importance:

Heuristic = Const1*Feature1 + Const2*Feature2 + …

Quoridor:

Quoridor is played on a 10x10 board with a pawn starting at each end. Each opponent has a number of fences that can be placed (each occupying two squares) to block the pieces. Each player may either play a fence or move the pawn one square. The first player to reach the other side of the board wins.

My heuristic was to compute the shortest path for each player to the end of the board. This is actually a somewhat expensive heuristic; it requires finding the shortest path. I used an algorithm called A* to find this path, using a subheuristic of the current Y coordinate. The final heuristic was:
Heuristic = C1(C2*MyShortestPath – C3*OpponentShortestPath) + C4(C5*MyFences-C6*OpponentFences)

Pente:

The goal of Pente is to get 5 in a row on a 19x19 grid. If two stones are sandwiched, then they are removed. If 5 pairs are captured, then you win. Our heuristic was a weighted heuristic:

Value = A*(HisCaptured-MyCaptured) + B*(1_to_win) - C*(1_to_lose)
+ D*(2_to_win) - E*(2_to_lose) + F*(3_to_win) - G*(3_to_lose)
+ H*(4_to_win)

We had different weights for moves away from winning and moves away from losing. We made the moves-away-from losing weight to be high, so that the program would be more defensive. If we captured 4 of his pieces, or if 4 of our pieces were captured, then the weight A was made very high so that our program would go for the fifth capture.

Upthrust:

Since we have an actual numerical score for the game, a simple heuristic could be MyScore – HisScore. You might consider another factor that rewards moving pieces up the game board.

The general tradeoff in all heuristics is an accurate expensive heuristic vs. a more inaccurate, but inexpensive heuristic. In general the cheaper heuristics win out if this lets you search one or two ply farther than the expensive heuristic. Since you will not be given a lot of time to make a move, it is up to you to determine the tradeoff that will be best for you!