Minimax Implementation – Othello

Here we extend our Othello example by adding an AI player using minimax. We start with the version where we had a computer version that picks random moves. First, here is a function we can use to compute the heuristic. It simply returns "our score" – "opponent score":

```
int heuristic(char board[][8], char whoseTurn)
{
	char opponent = 'O';
	if (whoseTurn == 'O')
		opponent = 'X';
	int ourScore = score(board, whoseTurn);
	int opponentScore = score(board, opponent);
	return (ourScore - opponentScore);
}
```

We are going to need to apply a move to the board and make a copy of the board in each recursive call to minimax. Here is an efficient way to copy one 2D array to another, as long as the array data is stored contiguously in memory. This is much faster than nested loops to copy each element. This uses a "C" style approach; in STL there is a new array class we will talk about later that lets us do the same thing in a method call, similar to Java's clone method.

```
void copyBoard(char src[][8], char dest[][8])
{
	memcpy(dest, src, 8 * 8 * sizeof(char));  // May need to #include <cstring>
}
```

Now we can implement the minimax decision function. It "returns" the move in parameters x and y, passed by reference. It is instructive to compare the code below to the pseudocode:

```
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]
```

The game state is represented by a board and whose turn it is.

```
void minimaxDecision(char board[][8], char whoseTurn, int &x, int &y)
{
	int moveX[60], moveY[60];
	int numMoves;
	char opponent = 'X';
	if (whoseTurn == 'X')
		opponent = 'O';

	getMoveList(board, moveX, moveY, numMoves, whoseTurn);
	if (numMoves == 0) // if no moves return -1
	{
		x = -1;
		y = -1;
	}
```

```
        else
        {
                // Remember the best move
                int bestMoveVal = -99999;
                int bestX = moveX[0], bestY = moveY[0];
                // Try out every single move
                for (int i = 0; i < numMoves; i++)
                {
                        // Apply the move to a new board
                        char tempBoard[8][8];
                        copyBoard(board, tempBoard);
                        makeMove(tempBoard, moveX[i], moveY[i], whoseTurn);
                        // Recursive call, initial search ply = 1
                        int val = minimaxValue(tempBoard, whoseTurn, opponent, 1);
                        // Remember best move
                        if (val > bestMoveVal)
                        {
                                bestMoveVal = val;
                                bestX = moveX[i];
                                bestY = moveY[i];
                        }
                }
                // Return the best x/y
                x = bestX;
                y = bestY;
        }
}
```

The minimax value function is similar except it doesn't care about the actual X/Y coordinate of a move. It only needs to return a heuristic value. Here is the pseudocode:

```
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[]
```

We pass in the board, whose turn it is (in the search tree, which alternates between X and O) and whose original turn it was that started the move. We need the original turn so we know if it is a minimizing or maximizing move, and we also need it to figure out how to calculate the heuristic (it is relative to whose turn it is). The searchPly parameter controls when we stop searching in the tree.

```
int minimaxValue(char board[][8], char originalTurn, char currentTurn, int searchPly)
{
	if ((searchPly == 5) || gameOver(board)) // Change to desired ply lookahead
	{
		return heuristic(board, originalTurn);
	}
	int moveX[60], moveY[60];
	int numMoves;
	char opponent = 'X';
	if (currentTurn == 'X')
		opponent = 'O';

	getMoveList(board, moveX, moveY, numMoves, currentTurn);
	if (numMoves == 0) // if no moves skip to next player's turn
	{
		return minimaxValue(board, originalTurn, opponent, searchPly + 1);
	}
	else
	{
		// Remember the best move
		int bestMoveVal = -99999; // for finding max
		if (originalTurn != currentTurn)
			bestMoveVal = 99999; // for finding min
		// Try out every single move
		for (int i = 0; i < numMoves; i++)
		{
			// Apply the move to a new board
			char tempBoard[8][8];
			copyBoard(board, tempBoard);
			makeMove(tempBoard, moveX[i], moveY[i], currentTurn);
			// Recursive call
			int val = minimaxValue(tempBoard, originalTurn, opponent,
			                       searchPly + 1);
			// Remember best move
			if (originalTurn == currentTurn)
			{
				// Remember max if it's the originator's turn
				if (val > bestMoveVal)
					bestMoveVal = val;
			}
			else
			{
				// Remember min if it's opponent turn
				if (val < bestMoveVal)
					bestMoveVal = val;
			}
		}
		return bestMoveVal;
	}
	return -1;  // Should never get here
}
```

All we need to do now is modify the main function so it invokes the minimaxDecision function to call the AI player. The following plays randomly for O but uses the AI player for X.

```
if (curPlayer == 'O')
       //minimaxDecision(board, 'O', x, y);
       getRandomMove(board, x, y, 'O');
else
       //cin >> x >> y;
       //getRandomMove(board, x, y, 'X');
       minimaxDecision(board, 'X', x, y);
```

There are lots of enhancements to make – weighting the corners to be worth more will make the AI player much better and encourage taking a corner whenever possible. Experiment with the search depth as well. The optional material on alpha-beta pruning allows much further look-ahead in the same amount of time.