Checkers AI - NegaMax with Alpha Beta Pruning

Functional Requirements

For this assignment, you will be writing an AI that can be used to play checkers (aka English Draughts). For this implementation, a human versus human version is provided. Your task is to add a computer player that will play against a human. Checkers is a simple game played on a 8x8 checkered board (32 dark squares and 32 light squares).   There are two colors of pieces, red and black, each being played by one player. Each player starts with 12 pieces. The black piece player goes first. Rules of moving, jumping (capturing), and board layout can be found at this link (ignore instructions specific to the site). The objective of the game is to capture all of the opposing pieces.

Finding the Best Play

At the heart of the game engine is an algorithm for trying to find the best move. To understand this algorithm, we need to view the various board configurations as forming a tree. (Note that this tree will not be implemented as a data structure - it is just how we think about the relationships between board configurations in order to guide our algorithm development.) Any board configuration can form the root of a tree. Its children are the board configurations that can be reached by making a single play from this configuration. Thus, a node in this tree may have numerous children.

In principle, we could determine the best move by examining this entire tree. However, this tree is so large that we have no hope of exploring the whole thing. For this reason, we will search only a portion of the tree, and use an *evaluation function* to compute a number estimating the strength of a player's position for each node whose children we don't examine. By combining this evaluation function with a partial search of the tree, we can typically do a better job of finding the best move than if we were to use the evaluation function by itself. In what follows, we will first describe the tree search algorithm, then we will describe the evaluation function.

The Negamax Algorithm

The algorithm we will use to search the tree is called the *negamax* algorithm, which is a more succinct presentation of the *minimax* algorithm. We will assume that the evaluation function attempts to give a higher value for a stronger position for the player whose turn it is to move. A value of 0 indicates a perfectly balanced configuration. The result of the evaluation function acts as a heuristic for our AI player to chose the best move with the information it knows at the time.

Alpha-Beta Pruning

Checkers is a simple game, but the number of board configurations make it very difficult represent the entire tree, even when the depth is limited. To improve our search of the tree, we will use a technique called *alpha-beta pruning*. Alpha-beta pruning improves the efficiency of the negamax algorithm by pruning branches of the tree that will not yield an optimal play given the current state of the game. This pruning uses information that has been obtained by searching other branches in the tree to provide bounds on the score that can be obtained by further exploring children of the current node. Alpha will represent a lower bound on the score the current player can achieve, and beta will represent an upper bound on the score the opposing player can achieve. At any node, we are trying to maximize the value that can be attained by the current player for this node. As this value increases, alpha also increases. However, it doesn't make sense to increase alpha above beta, because the opponent can simply avoid this node if its value is that high. Therefore, if this happens, we can stop exploring from this node.

In a game like checkers, this pruning greatly increases our search speed since the branching factor (how fast the tree spreads out) is very high. If the best moves are searched first, alpha-beta pruning can decrease the running time from $O(b^d)$ to $O(\sqrt{b^d})$ where b is the branching factor and d is the depth.

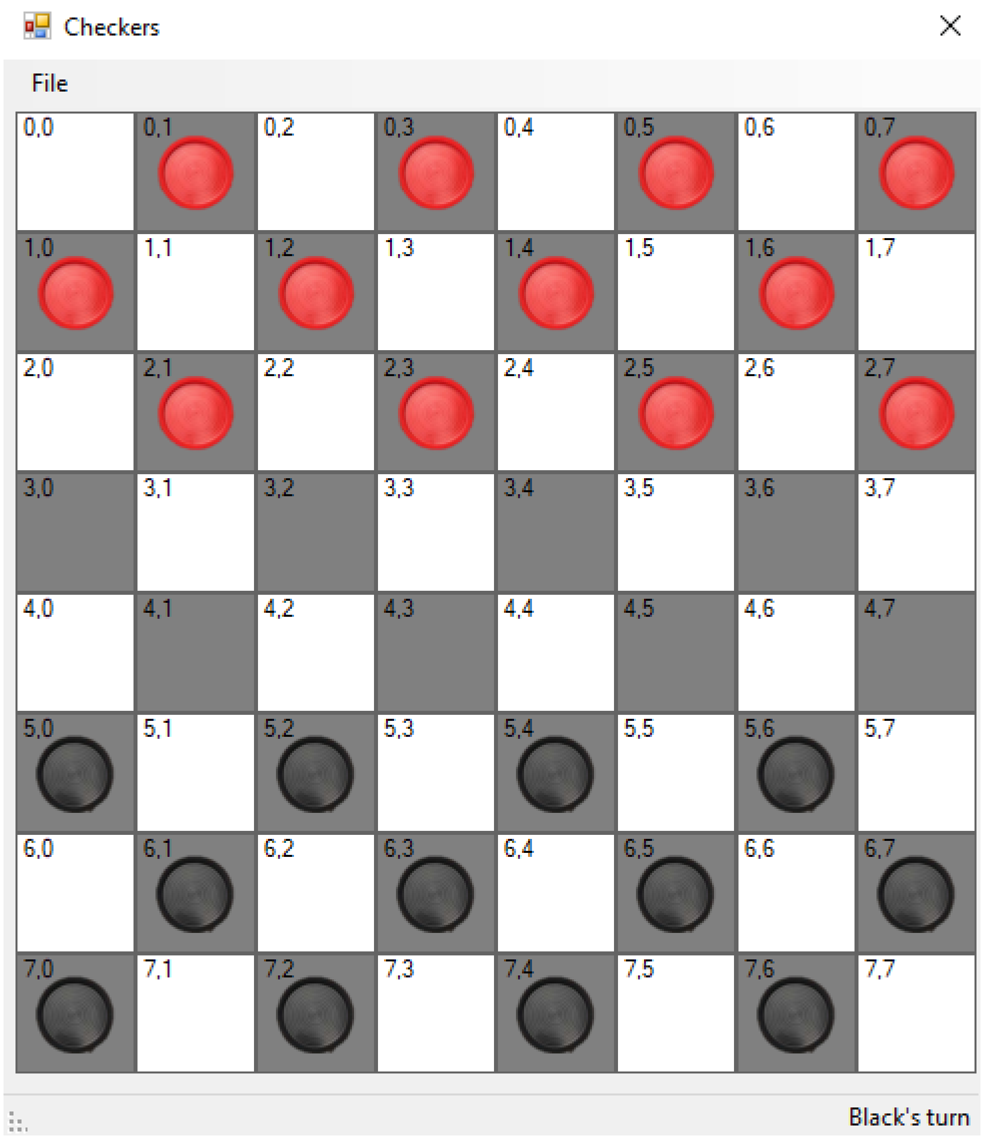
An illustration of the negamax algorithm with alpha beta pruning can be found on the Wikipedia page. Note that the algorithm shown on that page is slightly different from the one described later in the description of the `GameTree` class, specifically the need to flip the sign of the result of the evaluation function.

Starting the Assignment

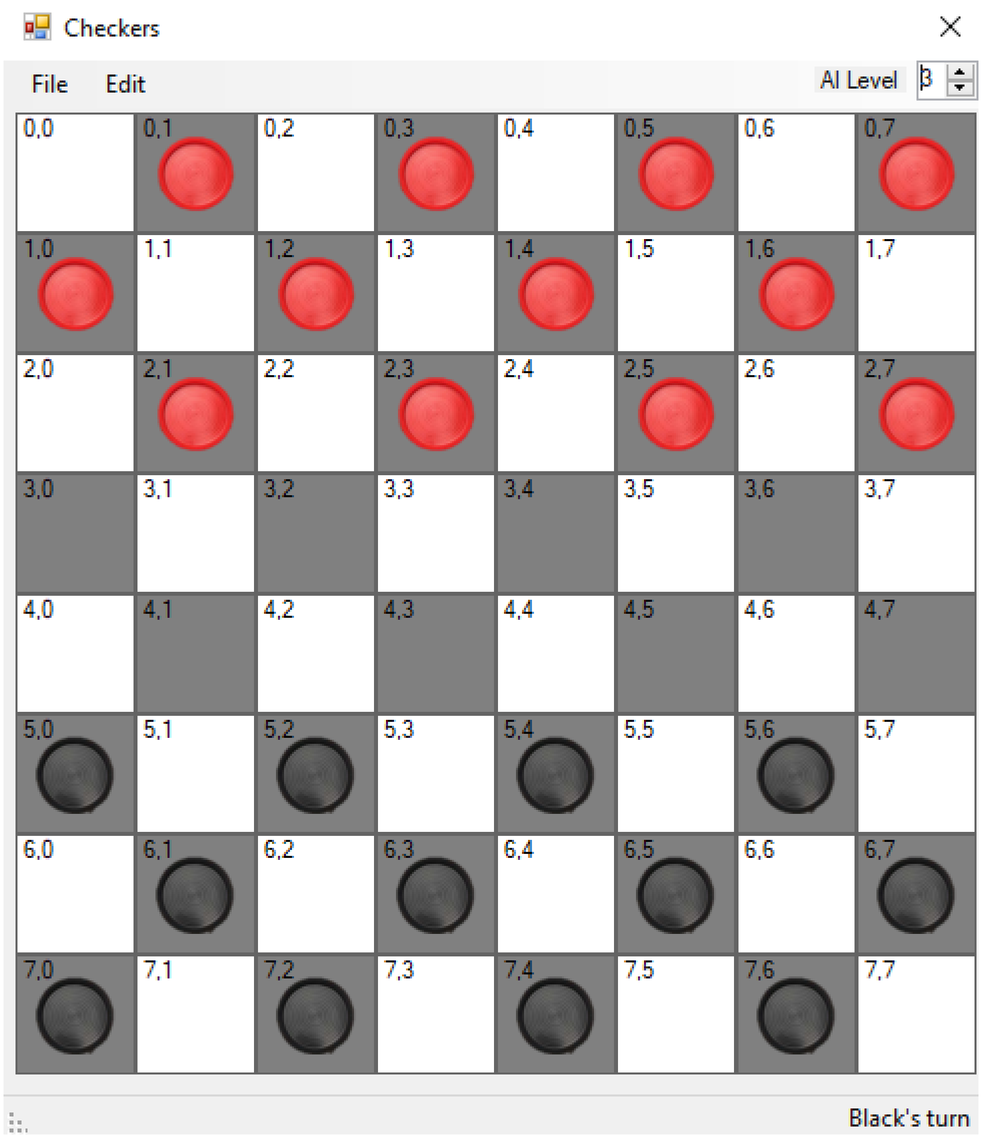
Create a GitHub repository using **this URL** and clone it to your local machine. This solution contains a Checkers implementation that allows for human vs. human play. The pictures used for the checker pieces are in the included **pics** folder.

User Interface

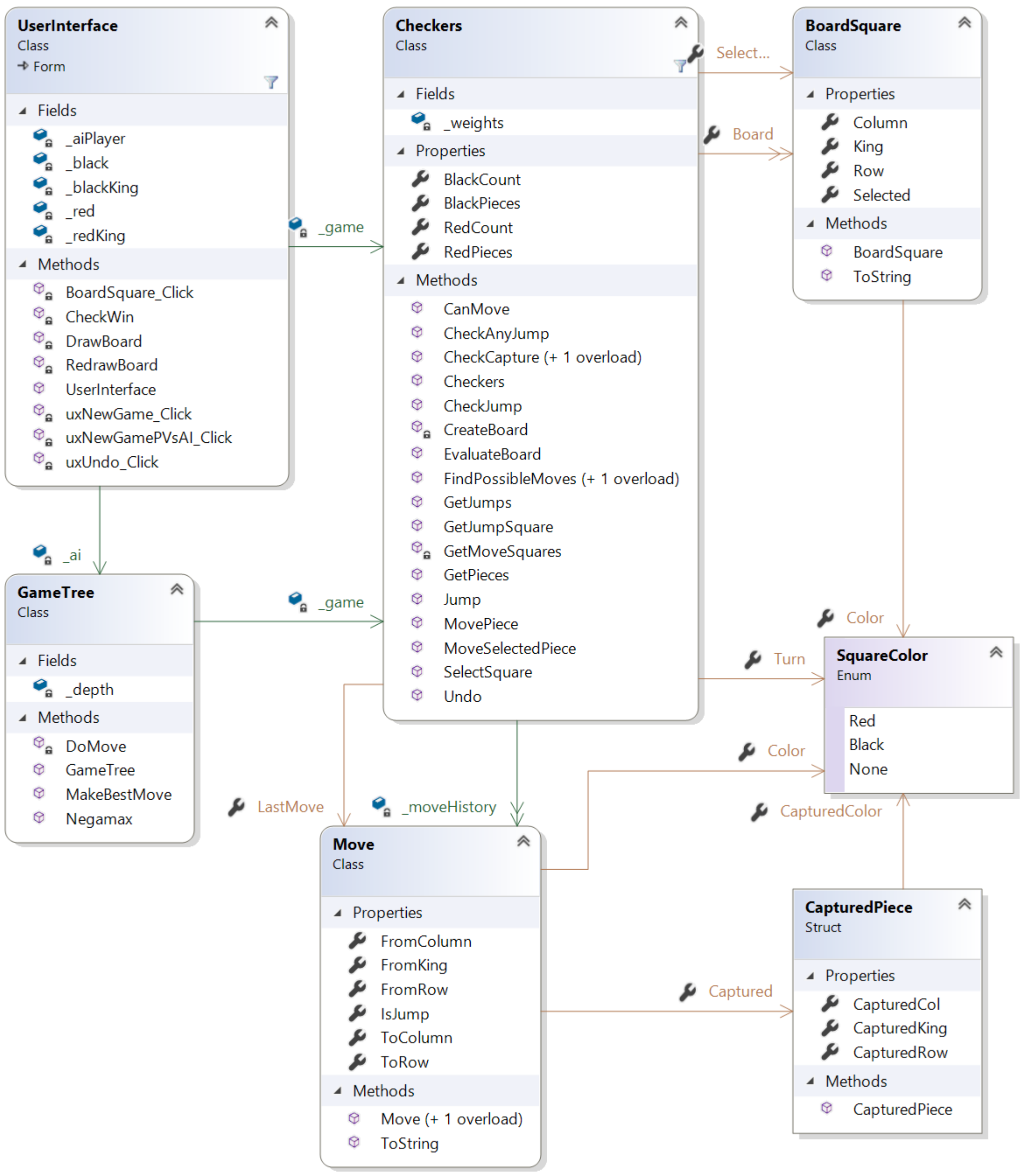
The base interface for a functional checkers game is provided.



To the given interface, you will need to add a few new menu options. Under 'File', add an option for creating a new game versus the AI. Add a new menu called 'Edit' that has a menu item 'Undo' mapped to the key **Ctrl + Z**. Selecting this menu item will undo a single move; specifically, if a multi-jump move has been made, it will undo only one jump. There must also be a new **NumericUpDown** control, in the top-right corner. This has an initial value of 3 and a **minimum** value of 1and **maximum** value of 12. This will control the level at which the AI plays (i.e., the number of plays it looks ahead). It may be changed at any time, but the level of play will only be set when a new player vs. AI game is started (i.e., changes during a game will have no effect).



Software Architecture



The **UserInterface** class is primarily used for drawing/updating the UI for when the game is created and when a piece is interacted with. All of the actual logic for playing checkers will be contained inside the **Checkers** and **GameTree** classes. Each square on the checker board will be represented using **BoardSquare**s. This class also stores whether or not a square contains an actual piece. The enumeration **SquareColor** is used to represent this information. The **Move** class will be used to represent individual moves. If the move is a jump, only a single jump is represented - multi-jump moves will be represented by several **Move** objects, one for each jump. The piece captured by a jump will be represented by the **CapturedPiece** structure.

The start code contains definitions for **UserInterface**, **Checkers**, **BoardSquare**, and **SquareColor** (**SquareColor** is defined in `BoardSquare.cs`). You will need to make some changes to **UserInterface** and **Checkers**, but you will not need to change **BoardSquare** or **SquareColor**. You will need to add the remaining definitions.

Coding Requirements

Specific requirements for the above software architecture are given in what follows. Feel free to add more **private** methods if you feel it improves the code. There is some room for the use of helper methods to reduce code for repeated logic.

The BoardSquare Class

This class has been completed for you. It is used to represent each square on the checkers board. Each square may have a piece (that might be a king), which has a color.

The CapturedPiece Structure

This structure contains all the information regarding a piece that is captured during a move. It is used in the **Move** class to help keep information from a jump organized. For this structure:

Add the following **public** properties, each of which should have a get accessor with default implementation (no set accessors):

- `public SquareColor CapturedColor`
  - Represents the color of the captured piece.
- `public int CapturedRow`
  - Represents the row in the board of the captured piece.
- `public int CapturedCol`
  - Represents the column in the board of the captured piece.
- `public bool CapturedKing`
  - Represents whether or not the captured piece was a king.

Then add the following constructor:

- `public CapturedPiece(int captRow, int captCol, SquareColor captColor, bool captKing)`
  - This is a basic constructor that sets the properties of the struct to the corresponding values.

The Move Class

The **Move** class contains the information needed to describe a move - either a non-jump or a single jump. You will need to add the following **public** properties, each of which should have a get accessor using the default implementation (no set accessors):

- `public bool FromKing`
  - Indicates if the piece being moved was initially a king
- `FromRow`, `ToRow`, `FromColumn`, `ToColumn`
  - Each of these are **int**s. These represent the coordinates on the board where a piece (From) was moved (To).
- `public bool IsJump`
  - Indicates whether this move is a jump.
- `public SquareColor Color`
  - Indicates the color of the piece that was moved.
- `public CapturedPiece Captured`
  - Gives the piece captured, if any.

Then add the following **constructor**s:

- `public Move(int fromRow, int fromColumn,int toRow, int toColumn, bool fromKing, SquareColor color)`
    - This constructor is used to construct a non-jump. It assigns each of the given parameters to the corresponding property, and assigns `false` to **IsJump**. Because **Captured** is unused for a non-jump, this property doesn't need to be initialized.
- `public Move(int fromRow, int fromColumn, int toRow, int toColumn, bool fromKing, SquareColor color, CapturedPiece cap)`
    - This constructor is used to construct a jump. It assigns each of the given parameters to the corresponding property, and assigns `true` to **IsJump**.

Add the following method:

- `public override string ToString()`
    - This method overrides the base **ToString** method in order to get a string representation of a **Move**. If the **Move** is a non-jump, return a string of the form, "*Color*-(*fromRow*, *fromColumn*) to (*toRow*, *toColumn*)", where *Color* is the color of the piece being moved, *fromRow* is the row where the piece started, *fromColumn* is the column where the piece started, *toRow* is the row where the piece landed, and *toColumn* is the column where the piece landed. If the **Move** is a jump, return a string of the form "*Color* Jump--*Move*(capt: (*capturedRow*, *capturedColumn*))", where *Color* is as above, *Move* is the string described above, and *capturedRow* and *capturedColumn* are the row and column of the captured piece. This method is used to display the last move in the user interface. It is also extremely useful when debugging your program.

The Checkers Class

The **Checkers** class holds most of the logic for checkers. All of the code supporting a player vs. player game of checkers has been provided for you. This class will need to be modified in order to allow the AI to play.

Add the following fields:

- `private Stack<Move> _moveHistory`
    - This stack will keep track of moves that are made so that you can either undo your move (if playing human vs human) or allow the AI to backtrack as it generates its game tree.
- `int[] _weights = new int[] { 5, 10, 1, 3};`
    - This array is contains the weights to apply when evaluating a board for the game tree. The weights, in order, are: pawn count, king count, number of pieces in their own back row, and pieces that are in a protected location (any edge of the board).

Add the following properties:

- `public Dictionary<(int, int), BoardSquare> RedPieces` and `public Dictionary<(int, int), BoardSquare> BlackPieces`
    - These two properties have a default get accessor and a default **private** set accessor. They contain each of the red and black pieces respectively on the board. The Key in the dictionary is a tuple (see this documentation) which represents the row and column where that piece is at on the board. A tuple is a data structure that allows you to group multiple pieces of information into one. You can think of it as a light-weight array, except that it is a value type, not a reference type.
- `public Move LastMove`

    - This property only has a get accessor which will return the last element on the `_moveHistory` stack (without removing it). If there is nothing on the stack, it should return **null**.

Modify the default constructor:

- Initialize the `_moveHistory` stack to a new stack.

Modify the following methods:

- `private void CreateBoard()`
    - Update the method to initialize the red and black pieces properties to be new dictionaries. Then, as each piece gets created, add it to the corresponding dictionary.
- `public bool Jump(BoardSquare current, BoardSquare target, SquareColor enemyColor)`
    - To this method, add a new **out** parameter of type **Move**, making the new method signature `public bool Jump(BoardSquare current, BoardSquare target, SquareColor enemyColor, out Move jumpMove)`.
    - Set the **out** parameter to be null initially, and if a jump is made, set the **out** parameter to be a new **Move** object representing the jump made. Once created, add it to the move history. Be sure to do this before the captured piece's color is set to `None`.
    - You will also need to be sure to remove the captured piece from the correct pieces dictionary (the call to **MovePiece** will update the other dictionary).
- `public bool MoveSelectedPiece(int targetRow, int targetCol)`
    - You will need to update the **Jump** call to include the **out** parameter, but you do not need to do anything with the value assigned to it.
- `public void MovePiece(BoardSquare piece, BoardSquare targetSquare, SquareColor enemyColor, bool jumped)`
    - Update the method such that if a jump was not made, you add a new move to the move history using the from and to squares.
    - Depending on the color of the piece moved, be sure to update the corresponding pieces dictionary to reflect the move.

Add the following methods:

- `public BoardSquare GetSquare(int row, int col)`
    - This is a method that will return the board square at the given row and column if the row and column are within the bounds of the board. If not, it will return null.
- `public bool Undo()`
    - This method should undo the last move if there was one. If **SelectedPiece** is not **null**, set its **Selected** property to 'false'. Then if the move history is nonempty, remove the last move. Use the **GetSquare** method to get the "from" square and the "to" square. Reset the **Color** and **King** properties in the "from" square back to their original values and the "to" square color to **SquareColor.None**. Also reset the **SelectedPiece** to **null** and the **Turn** property to the color of the player making the move being undone. Be sure to update the appropriate pieces dictionary to reflect the undo. If the move is a jump, get the square of the captured piece, set its **King** and **Color** properties to those of the captured piece, and update the appropriate pieces dictionary and pieces count. The method should return `true` if a move was undone or `false` otherwise.
- `private List<BoardSquare> GetMoveSquares(BoardSquare piece, int distance)`

    - This method returns a list of squares that the given square can move to. The distance parameter should be 1 for a normal move and 2 for jumps. Remember that kings can move on each diagonal, but a normal piece cannot. Helper methods are very useful here to reduce repeated code.
- `public List<Move> GetJumps(BoardSquare piece, SquareColor enemyColor)`
    - This method returns a list of all jump moves possible from the given piece. The enemy color is the color of the piece we are trying to capture. You will need to utilize the **GetMovesSquares** method (passing in 2 for the distance) to get all of the possible jump targets, then use the **Jump** method to try to execute the jump. If the jump is successful, add the jump move to the list of possible moves, then call the **Undo** method to reverse the jump before trying the next target. If no jumps are found, the method should return an empty list.
- `public List<Move> FindPossibleMoves(BoardSquare piece, SquareColor enemyColor, out bool seenJump)`
    - Given a piece's board square, this method should return a list of all possible moves for it. If there are any jumps to be made, the list returned should *only* contain jumps since the game enforces the rule that if a jump is possible, it must be made. If there are no jumps possible, use the `GetMoveSquares` (passing 1 for distance) to get all of the valid squares the piece can move to and add each of those as a valid move to the list you return.
- `public List<Move> FindPossibleMoves(out bool seenJump)`
    - This method is similar to the one previously described; however, it returns a list of possible moves for all pieces of the current player. Similar to before, if any piece has a jump available, this method should only return moves that are jumps.
- `public int EvaluateBoard()`
    - This method evaluates the current board to create the heuristic value for the Negamax algorithm. For each of the red pieces and black pieces, count the number of pawns, kings, pieces on their own back row, and pieces that are protected (edges of the board). Note that a piece can be counted in more than one category. A helper method is useful for counting each piece so logic does not have to be duplicated for red and black. Once counted, the weights should be applied. Then return the sum of the weighted counts of the current player minus the other player.

The GameTree Class

The **GameTree** class contains the logic for how the AI player in the checkers game makes its move.

Add the following fields:

- `private Checkers _game`
    - The reference to the Checkers game that is currently being played with this AI.
- `private int _depth`
    - The depth represents the maximum number of individual player turns the AI will look ahead in the game tree before making a move.

Add the following constructor:

- `public GameTree(Checkers game, int depth)`
    - The constructor of this class only assigns the given values to the corresponding fields.

Add the following methods:

- `private void DoMove(Move move)`
  - This method will execute the given move on the board. You can check if it is a jump by looking at the **IsJump** property. If it is a jump, you can call the game's **Jump** method, passing the from and to squares from the given move. If it is not a jump, you will call the **MovePiece** method.
- `public int Negamax(int alpha, int beta, int depth, out Move bestMove)`
  - This method will execute the negamax algorithm with alpha-beta pruning as it was described previously. `depth` is how deep this algorithm will search the tree from the current node. (Remember, we are not storing the tree anywhere - it is just how we are thinking about the board positions as we conduct our search). The algorithm is as follows:
    1. Set `bestMove` to **null**
    2. Base case: depth is 0
       1. return the heuristic of the current game (use the `EvaluateBoard` method)
    3. Declare two variables, one of type **Move** (referred to as `localBest` below) and one of type **int** (this is initially `int.MinValue` and is referred to as `score`). These will keep track of what the current best score for the tree node is and the current best move. Remember that each "node" of the tree represents the state of a checkers game.
    4. For all moves in the current game (for the current player)
       1. store the current player's turn in a temporary variable so we can tell if the turn changes after the move is made (this is for the multi-jump scenario)
       2. Do the move
       3. Find the score of the current state of the game by recursively calling NegaMax. Be sure to store the result of the recursive call in a local variable as this will be the best score for the move that was made for the current state. There are two recursive cases:
          1. The move did not change whose turn it is. Therefore pass to NegaMax `alpha`, `beta`, `depth`, and `localBest`.
          2. Else, the move changed the turn, therefore we are now trying to maximize the score of the board for the other player. To do so, we will flip the alpha and beta parameters by calling **NegaMax** with `-beta`, `-alpha`, `depth-1`, and `localBest`. Notice that `alpha` and `beta` get swapped and their signs flipped. Finally, we negate the returned value.
       4. If the value obtained by step 3 is greater than or equal to `score`, then update `score` with this value and store the current move into `bestMove`.
       5. Update `alpha` to the maximum of `alpha` and `score`
       6. Undo the move that was made so we do not corrupt the state of the board with the next move.
       7. if `alpha` is greater than or equal to `beta`, there is no point in evaluating any more moves from this node, so exit the loop
    5. Return `score`
- `public bool MakeBestMove()`
  - This method is used by the **UserInterface** class to make the best move for the AI when it is the AI's turn. Simply call the **Negamax** method, with `int.MinValue`, and `int.MaxValue` as `alpha` and `beta` and `_depth` for the depth of the game tree. If the

best move returned through the **out** parameter is not **null**, then a move is available. Do that move and return `true`. Return `false` otherwise to indicate to the UI that there are no more moves available for the AI to make (this is a deadlock scenario).

The UserInterface Class

The **UserInterface** class is responsible for drawing and updating the checker board. This class is mostly finished for you; however, you will need to make some modifications:

- Add two private fields `private bool _aiPlayer` and `private GameTree _ai` which will indicate if the game is player vs. AI and hold the AI's game tree.
- `private void RedrawBoard()`
    - Modify this method to update **uxTurn**'s **Text** property to also display the last move that was made (if one was made) along with whose turn it is. If there was a last move, the text should be updated in the form: "(Last Move: *moveGoesHere*)---Now *turnGoesHere*'s turn"
- `private void BoardSquare_Click(object sender, EventArgs e)`
    - Modify this method so that it handles both player vs player and player vs AI games. Note that when it is the AI's turn, this will have to be in a loop until it switches to the other player in order to handle multi-jump moves. The method should handle situations when the AI run's out of moves (deadlocks) and show a winner (if there is not, show a tie).
- `private void uxNewGame_Click(object sender, EventArgs e)`
    - Modify this method to switch the `_aiPlayer` to false and enable the undo feature in the user interface.
- `private void uxNewGamePVsAI_Click(object sender, EventArgs e)`
    - This is a new click event handler for the menu option for creating a game vs an AI. The method should set the `_aiPlayer` to be true and the `_ai` (the game tree) to be **null**. It should then draw the board and disable the undo feature in the user interface.
- `private void uxUndo_Click(object sender, EventArgs e)`
    - This is a new click event handler for the undo menu item. It simply need's to call the **Checkers** object's **Undo** method, and if that method returns `true`, call the **RedrawBoard** method.

Testing Your Program

Be sure to test win conditions for both sides (human and AI, and each kind of move/jump. To assist in testing your program, board squares are labeled with their row and column. Adding the last move made to the status bar where the current player is displayed can be helpful as well (the AI can make very fast moves). Overall, testing this homework can be tricky. Your AI should not blindly throw pieces away (especially at levels higher than 3 or 4) and be more difficult to beat as you increase the AI level. Note that the AI is not the most sophisticated in choosing moves at times (due to the evaluation function), but it should play a decent game of checkers at higher levels.

Performance

There will be a cascade effect when creating a new game, but moves will be made instantly for player vs player. For games with the AI, moves should be made instantly up to around level 9 or 10. At 10, you will start to see some delay in each move, but it should be around 1-2 seconds for some moves and 3-4 seconds for others. At level 12, you will start to see longer delays (5-10 seconds). Your AI should play a good game of checkers and be harder to beat as you increase the level.

Submitting Your Assignment

Be sure to **refresh** your Team Explorer, **commit** all your changes, then **push** your commits to your GitHub repository. Then submit the *entire URL* of the commit that you want graded.

Important: If the URL you submit does not contain the 40-hex-digit fingerprint of the commit you want graded, **you will receive a 0**, as this fingerprint is the only way we can verify that you completed your code prior to submitting your assignment. We will only grade the source code that is included in the commit that you submit. Therefore, be sure that the commit on GitHub contains all of the ".cs" files, and that they are the version you want graded. This is especially important if you had any trouble committing or pushing your code.