Homework Assignment 3: Checkers, Continued

For this assignment, you will add a computer player to the checkers program from Homework Assignment 1.

User Requirements

You will need to add to Homework Assignment 1 the ability for the user to choose to play a computer opponent, as either side (i.e., White or Black). The user should be able to select from among several levels of difficulty. The Undo feature should be disabled when a computer opponent is playing. If the user does not select a computer opponent, the program should behave as for Homework Assignment 1.

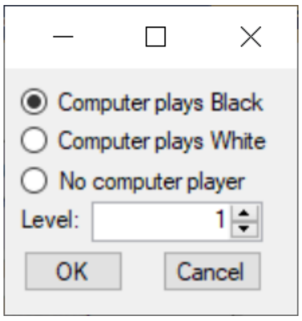
Starting the Assignment

Create a GitHub repository using this URL. This repository contains the model solution to Homework Assignment 1.

User Interface

The demo video shows the desired behavior of your finished program. Here, we will first summarize the GUI design with some instructions on how you will need to set this up. We will then summarize how the finished program should respond to various user actions.

You will not need to change the main GUI at all. However, you will need to add a dialog resembling the following:



Create this dialog by adding a new **Form** to this project as you would add a new class or **UserControl**. Call this **Form** `NewGameDialog.cs`. This will give you a Design window in which you may build this dialog.

Set this **Form**'s **Text** property to "New Game". If the dialog were bigger, this text would then show in its title. As it is, it will appear if you hover over the dialog's icon in the Windows Task Bar while the program is displaying it. Also, to prevent the form from being resized, set its **FormBorderStyle** property to **FixedDialog** and its **MaximizeBox** property to **False**.

The top three lines of the dialog are **RadioButton**s, which you can add from the Toolbox. Because these **RadioButton**s are in the same container, exactly one of them will be checked at any time. Below the **RadioButton**s are a **Label** and a **NumericUpDown**. Set the **NumericUpDown** so that it may contain an integer in the range from 1 to 12, right-aligned (see its **Minimum**, **Maximum**, and **TextAlign** properties). Its value should initially be 1.

At the bottom are two **Button**s. In order to cause them to close the dialog after it has been displayed using its **ShowDialog** method, set their **DialogResult** properties to indicate the respective values that **ShowDialog** should return.

Behavior of the GUI

This section details the behavior that you will need to implement, as described under "Coding Requirements" below (see the demo video for examples).

The program should begin by showing only the New Game dialog as a modal dialog. The user should be able to change which **RadioButton** is checked (exactly one of them should always be checked) and the value in the **NumericUpDown** (an integer from 1 to 12). Clicking either the "OK" button or the "Cancel" button should close the dialog. If the user closes the dialog by clicking "Cancel" or the "X" in the upper-right corner, the program should exit. If the user closes the dialog with the "OK" button, the main GUI should be displayed.

If the checked **RadioButton** when the user clicked "OK" was "No computer player", the program should then function as for Homework Assignment 1, except when the "New Game" menu item is selected. In this case, the New Game dialog should be displayed as a modal dialog. Its behavior should be as described in the above paragraph, except that canceling the dialog should not terminate the program - instead, the current game should continue.

If the checked **RadioButton** when the user clicked "OK" was either of the other two, a game with a computer opponent should begin. The behavior should be much like it was for Homework Assignment 1, except that the program should make plays whenever it is the computer player's turn.

Because the computer player won't click on pieces to move them, squares should not be highlighted to begin a computer player's move. However, when the computer makes multiple jumps on its turn, the GUI should display the result of each move, highlighting the square reached if it is not the final destination of the the turn, and pause for half a second before making the next move in the turn. The highlighting should disappear as the checker leaves a square, and the final square reached should not be highlighted. Note that this highlighting is done in the same way as when a human player is moving, except that the initial location of the moving piece isn't highlighted.

The "Undo" menu item should be disabled whenever there is a computer opponent. The "New Game" menu item should behave as in a game with no computer player.

The value of the **NumericUpDown** when the "OK" button was clicked should govern the quality of the computer's play. Specifically, the computer should play better at higher levels, though it might take longer to play. Whenever the computer has only one legal move, it should make this move immediately, no matter which level is being played. At the higher levels, the computer should be capable of beating a novice player consistently.

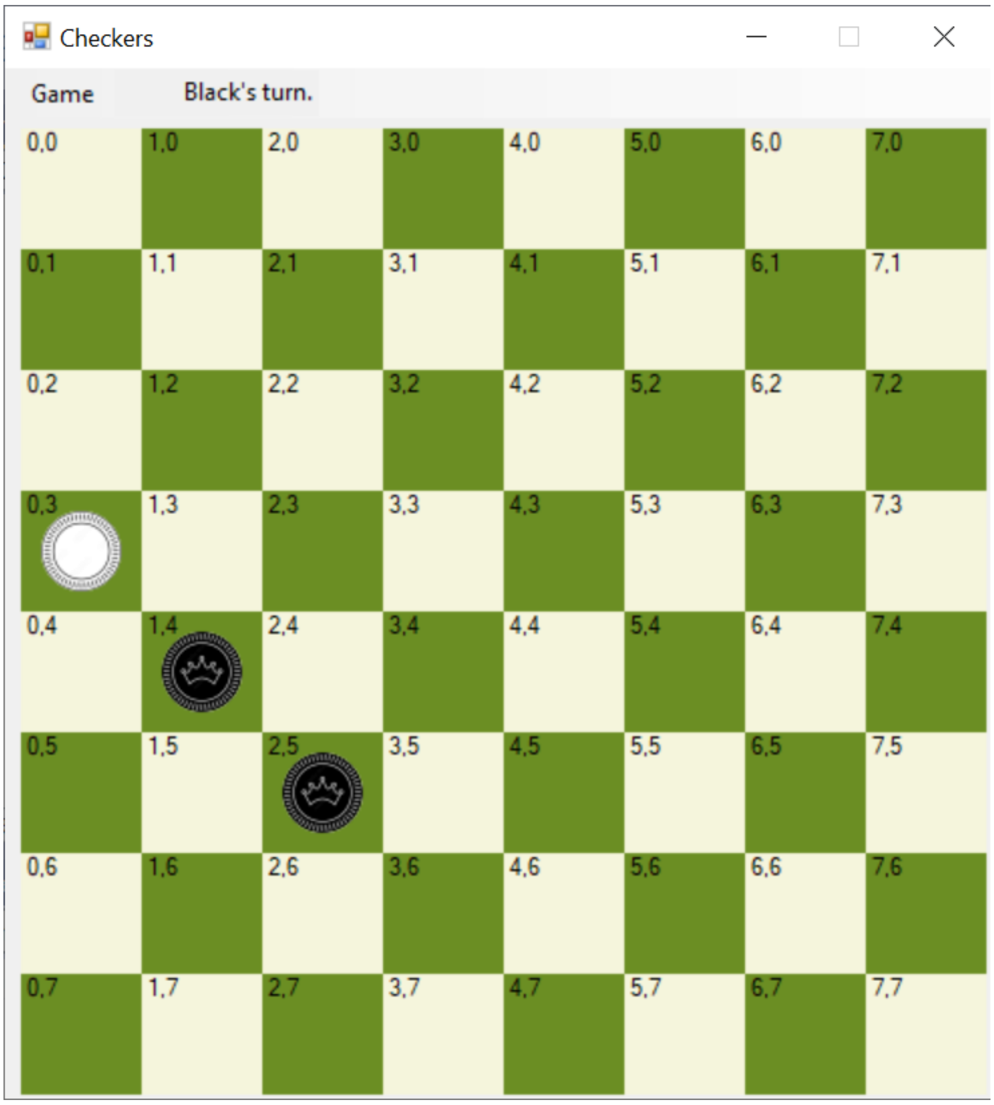
Finding the Best Move

Before we go into the details of the coding changes you will need to make, we will first discuss how the computer player will decide upon its moves. Knowing how we will do this will help us to make design decisions such as which data structures to use. We will first describe an approach that doesn't quite work. We will then show how this can be modified to obtain a useful algorithm. We will then give an additional improvement to address a weakness in the algorithm. Finally, we will describe an optimization that greatly improves the performance.

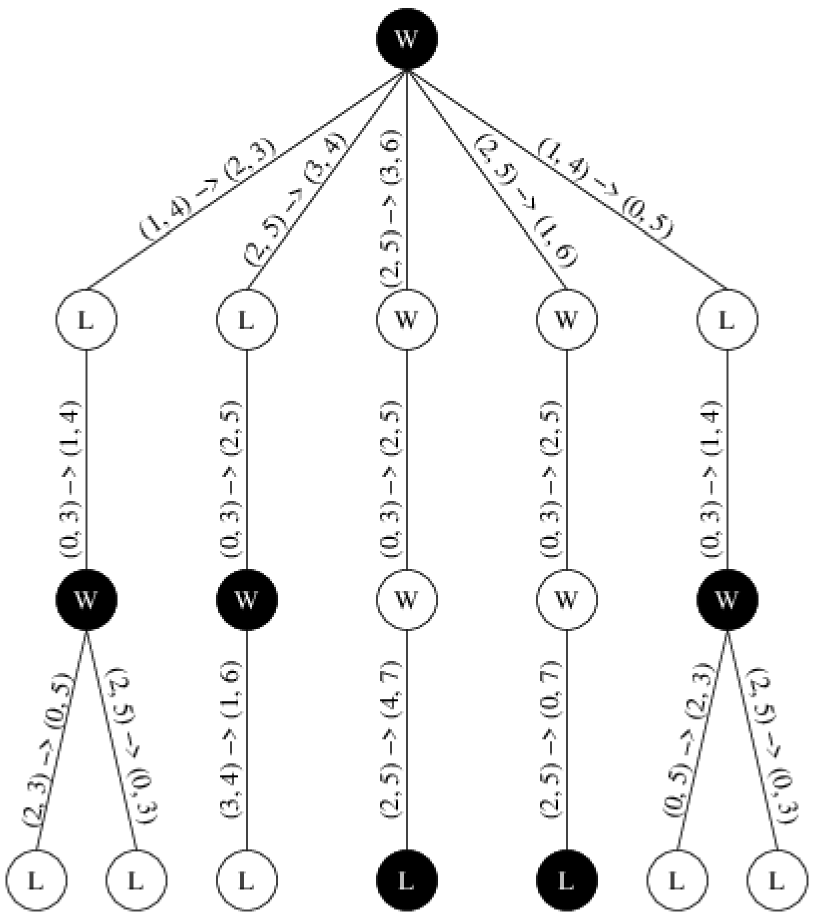
Game Trees

From any position in Checkers, we can define a nonempty *game tree* to allow us to determine the best move from that position. The tree has the game position (including the player whose turn it is to move) as its root. Its children are the game trees for the positions that can be reached in a single move.

Consider, for example, the following position:



Because there are five legal moves (two by the checker at (1, 4) and three by the checker at (2, 5)), the game tree from this position would have five children, each of which is a game tree from one of the five resulting positions. The entire game tree looks like this:



Nodes are colored to indicate the player whose turn it is to move in that position. The move that leads from a parent node to a child node is indicated on the line connecting the parent with the child. Note that there are two white nodes with white children - these nodes indicate the position after the first move of a double-jump turn.

Each leaf is a position in which the game is over; hence, the player whose turn it is to move has lost the game at this point. We designate this fact with an "L" in each leaf. Thus, there are seven possible conclusions to the game, two of which are won by White (when Black first moves the checker at (2, 5) to either (1, 6) or (3, 6), thus allowing White to jump both Black pieces), and five of which are won by Black. We can complete the labeling of the nodes by working our way upward. When a node's children have been labeled, if one of the children is either of the same color and labeled "W" or of the opposite color and labeled "L", we label this node "W"; otherwise, we label it "L". Thus, if a node is labeled "W", the current player at that node has a winning strategy by making a move that leads to a "W" node for the current player or an "L" node for the opponent.

For example, consider the root node. Because there are no jump moves from that position, each move causes Black's turn to end. As we have already observed, two of these moves lead to wins by White; however, because there is a move (in fact, there are three) that leads to a loss for White, Black has a winning strategy. For example, if Black plays (1, 4) to (2, 3), White's only move is to play (0, 3) to (1, 4). At this point, either of Black's moves ends the game, resulting in a win for Black.

Because the root node is a win for Black, the White player should avoid getting into this position, if possible. For example, suppose the root node was reached by White playing (1, 2) to (0, 3). In this parent position, White might have been better off making a different play - hopefully a part of a winning strategy. However, White's only other option was to play (1, 2) to (2, 3), at which point Black's only move is a jump move that wins immediately. Hence, this parent node is a loss for White.

We might try to implement an algorithm that never loses by building the entire game tree from the starting position. However, this is impossible for the game rules described in Homework Assignment 1, as there are no rules that force a game to end (for example, the game might reach a position in which each player has one king, and neither player wants to approach the other player's king). As a result, the game tree is infinite. There are rules that we could add to force the tree to be finite, but doing this in any of the standard ways results in a game tree that is much too large to build, or even to traverse.

The Negamax Algorithm

In order to make use of the game trees described above, we need a way of using just a portion of the tree. Specifically, from a given position, we can generate the tree out to a specific number of turns. However, we can't evaluate the nodes as we did above because the leaves probably are not positions at which the game is over.

To overcome this difficulty, we use an *evaluation function* to estimate the strength of the position at a leaf node. Specifically, if the game is not over in that position, we compute a *score* for each player based on that player's pieces. Let B be the y-coordinate of the player's back row (0 for White or 7 for Black), and suppose a player's piece is at location (x, y). We then add the following value for that piece, depending on whether it is a pawn or a king:

- **A pawn:** Add 80 + $(B - y)^2$. Thus, a pawn is worth from 80 to 116 points, with a higher value if it is more advanced.
- **A king:** Add 195. If $x + y - 1$ is divisible by 12 or if $xy = 12$, add 4 more. Thus, a king is worth either 195 or 199 points, or about twice the value of a pawn. The 4 extra points are awarded if the king is in one of the locations (0, 1), (1, 0), (3, 4), (4, 3), (6, 7), or (6, 7). We will discuss the significance of these locations below.

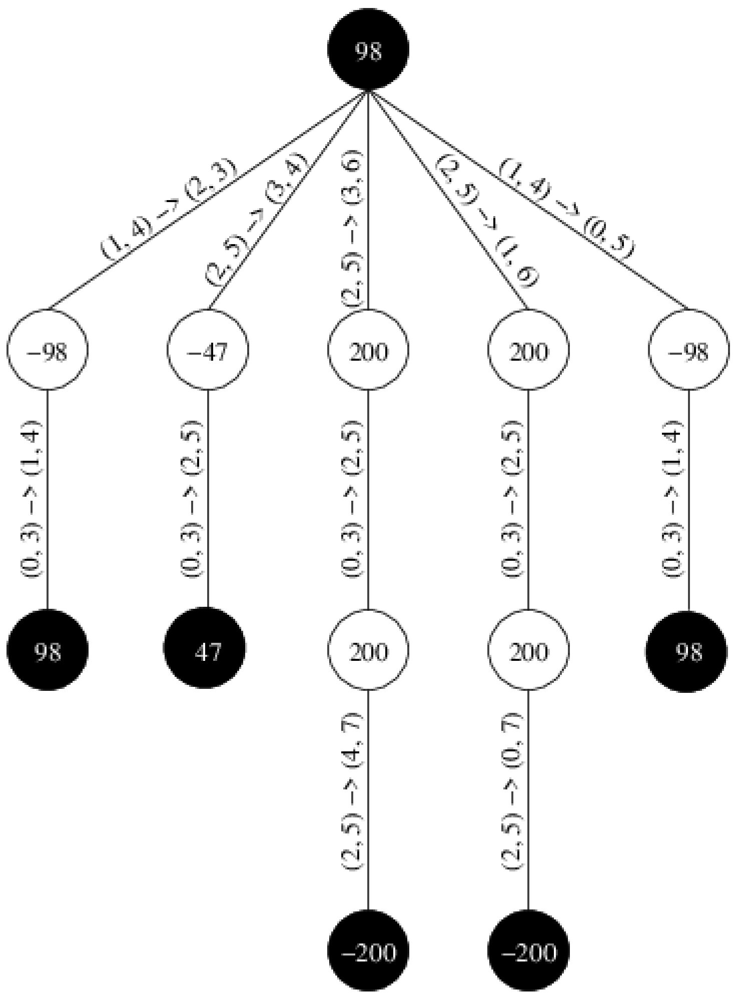
If the current player's score is s, the other player's score is s', and the number of pieces on the board for both players combined is n, the value of the evaluation function is:

$$\frac{s - s'}{n}$$

The numerator grows larger as the score for the current player grows in comparison to the score of the other player. We divide by n to reflect the fact that having fewer pieces on the board favors the player with the material advantage.

If the game is over in a given position, we need to assign it a value smaller than any possible value that can be produced by the above evaluation function. Because each piece contributes more than -200 and less than 200 points to the numerator, the evaluation function's value is also within this range. We can therefore use -200 or any smaller value for a lost game. For reasons we will discuss shortly, we will incorporate the remaining lookahead k in the value of a lost game. Specifically, we will use the value -200 - k.

The following figure shows the tree generated from the position shown above using a lookahead of two turns:

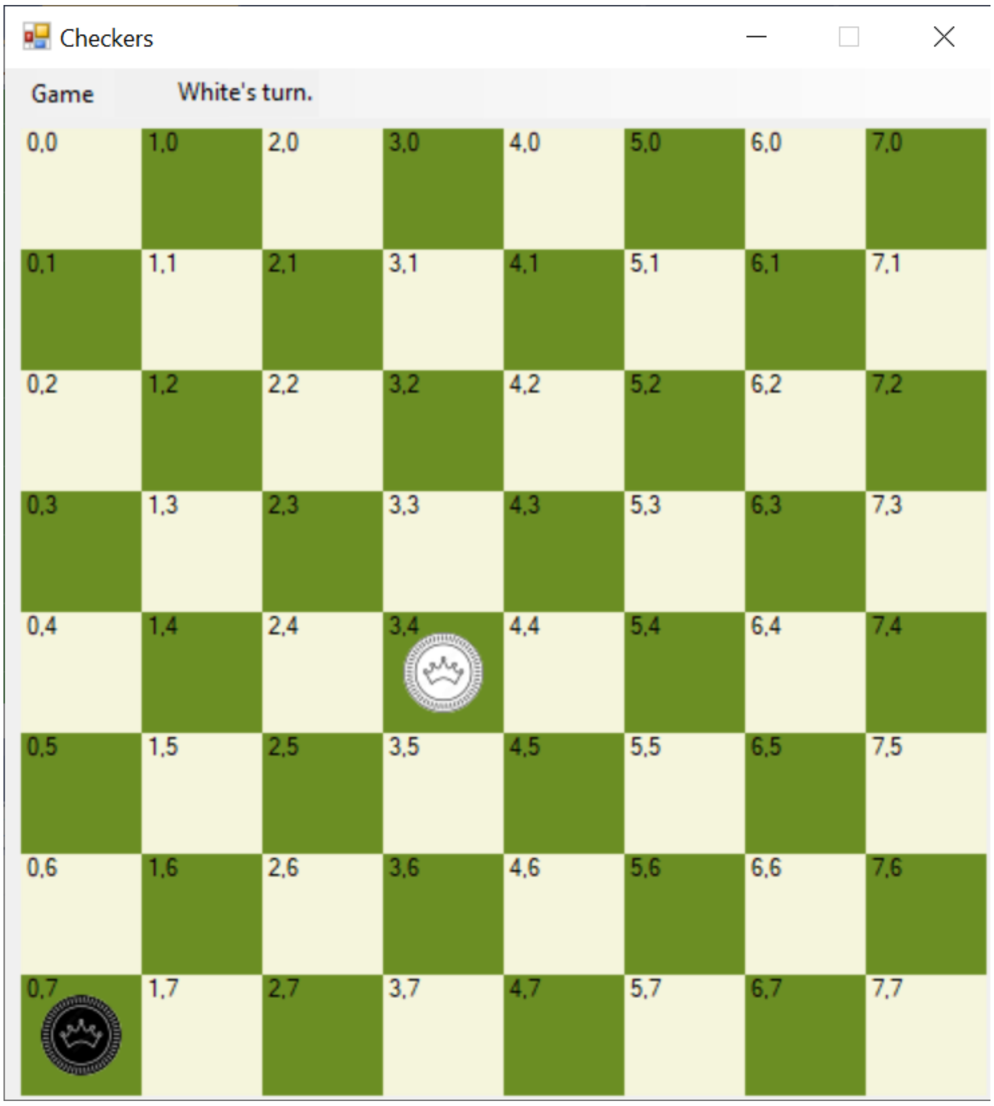


Note that we are specifying the number of *turns* to look ahead, not the number of *moves*. Thus, two of the paths have three moves because they each include a double-jump turn.

Because we are now using values other than just L and W, we need to generalize the way we propagate values to parents. When the child represents a position in which the opponent is to play, we use the negative of its value in order to translate it to the perspective of the current player. For example if a White node has a Black child with a value 15, we interpret that value as being 15 points in favor of Black. This is the same as saying it is 15 points *against* White. If this node is the only child, the parent then has a value of -15. On the other hand, if a child represents a position in which the same player is to play, we don't need to change the value.

To get the value of a node with more than one child, we first negate any values that are for positions from which the opponent will play, then take the maximum of all resulting values. Thus, for example, after we negate the values of all the children of the root of the above tree, 98 is the maximum. Therefore, 98 is the value of the root. The plays that attain this value are (1, 4) -> (2, 3) or (1, 4) -> (0, 5). Even though we have not verified that either of these plays is part of a winning strategy (because we didn't look far enough ahead), they appear to be the most promising based on the portion of the game tree that we have searched.

Now let us return to why we are incorporating the remaining lookahead in the value of a lost position. Consider the following position with White to play:



White can win in three turns by moving to (2, 5) (Black must then move to (1, 6), and White then captures this piece, winning the game). However, it turns out that *any* of Whites initial moves can force a win in at most seven turns. Suppose, for example, that White moves to (4, 3). We then have the following:

- **Turn 2:** Black must move to (1, 6).
- **Turn 3:** White can move to (3, 4).
- **Turn 4:** Black can move to (0, 5), (0, 7), or (2, 7).
- **Turn 5:** White can move to (2, 5).
- **Turn 6:** Black can move to (1, 4), (1, 6), or (3, 6).
- **Turn 7:** In each case, White has only one legal move, which wins the game.

The point is that if each losing position were valued -200, then White would have no reason to choose (2, 5) over any other move. Furthermore, if Black moves to (0, 7) on Turn 4, the position is exactly the same as in the above figure. Thus, the same sequence of four turns could repeat forever, never reaching the win that is within White's grasp.

Subtracting the remaining lookahead solves this problem. Suppose, for example, that the lookahead is initially 8. Then the loss reached after three turns has a remaining lookahead of 5; hence its value is -205 (or 205 from White's perspective). The loss reached after seven turns has a remaining lookahead of 1; hence its value is -201 (or 201 from White's perspective). Thus, the move to (2, 5) gives the best value (205). (The improvement we discuss under "Randomization", below, will also solve this problem, but incorporating the remaining lookahead will cause the win to be reached more quickly.)

The algorithm we have described in this section is known as the *negamax* algorithm. In the traditional version, a child of a node always represents a position in which the opponent is to play; hence, we would always negate all the values, then take the maximum (hence the name "negamax"). We have had to modify this algorithm slightly in order to handle sequences of plays by the same player.

We will refer to the tree searched by the negamax algorithm with a lookahead of *k* as the *k*-negamax tree from a given position. If *k* is understood, we may simplify the name to the negamax tree. We will refer to the value of a node in a negamax tree as its *negamax value*. For each node other than the root, we will refer to the node's value from the perspective of the current player in its parent as its *adjusted negamax value* (in other words, if the current player in the node is the same as the current player in its parent, the adjusted negamax value is the same as the negamax value; otherwise, it is the negative of the negamax value.) Note that in order to compute a negamax value and the move to a child whose adjusted negamax value is this node's negamax value, we don't actually need to build this tree - we can simply traverse it using a representation of the game to keep track of the current node.

Randomization

The problem discussed above isn't limited to positions which the negamax algorithm evaluates as a win. Suppose for example, that White has only a single king remaining, and is trying to avoid a loss by moving that king back and forth between the locations (6, 7) and (7, 6). If the computer player, playing Black, has at least two pieces, a win is possible, but it likely requires more turns than the lookahead. We can value locations (0, 1), (1, 0), (6, 7), and (7, 6) more highly than other locations for kings - in fact our evaluation function does that, as well as valuing locations (3, 4) and (4, 3) more highly. This encourages Black not only to occupy one of these locations, but also to force White out of these locations. However, as we have seen above, the negamax algorithm won't necessarily lead Black to do this, even if it evaluates such a position more highly - it may continue indefinitely in positions that are a few turns away from such a position. Furthermore, we can't solve this problem by incorporating the remaining lookahead into the value of the position because a position that is not the end of the game is only evaluated when the remaining lookahead is 0.

The solution to this problem is to incorporate randomization into the move selection. Specifically, if multiple moves lead to positions having the maximum adjusted value, we randomly pick one of them. We can accomplish this randomization by shuffling the legal moves before using the negamax algorithm. We then let the negamax algorithm select the first move that leads to a maximum adjusted value, just as it normally would. Because the moves have already been shuffled, this selection is a random selection.

Note that shuffling the legal moves is too expensive to do for each node in the tree. However, we don't need to shuffle for each node, because we don't need to choose a move from any node other than the root. Thus, we only shuffle the legal moves before we make the initial call, and just use the legal moves in the order they are for each recursive call.

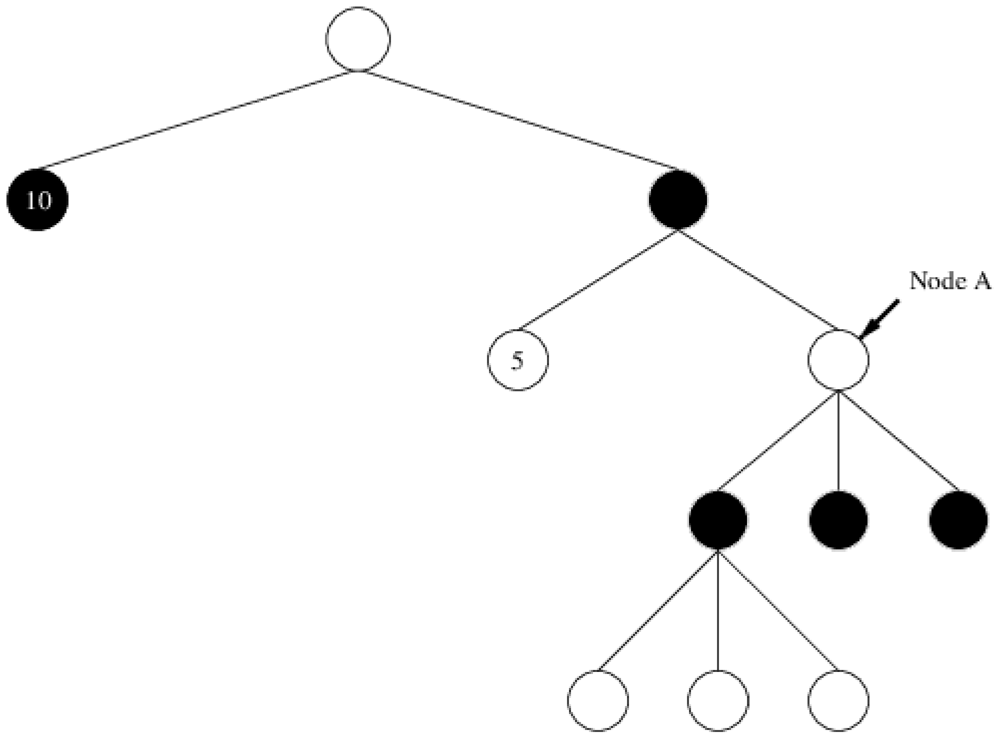
This randomization is a powerful mechanism for leading the computer player to a desired position. The idea is that if we choose randomly enough times from among moves that ultimately lead to a position with a desired value, eventually we will reach such a position (although the process may be painfully slow).

We can therefore see why we award bonus points for the locations (0, 1), (1, 0), (6, 7), and (7, 6). We also award bonus points for (3, 4) and (4, 3) in case a king is stuck in the wrong corner. There is now incentive to move near the center, where it will eventually move toward the correct corner.

Alpha-Beta Pruning

Efficiency is crucial to the effectiveness of the negamax algorithm, as the more turns we can look ahead within a reasonable amount of time, the better the choice of moves. One reason why we chose a fairly simple evaluation function is so that it can be computed quickly. In the next section, we will discuss data structures that will enable the game simulation and the evaluation function to be implemented more efficiently. Here, we will look at an optimization to the negamax algorithm itself.

We first observe that, if we keep track of some contextual information, we can avoid exploring many of the paths. For example, consider the following portion of a negamax tree (i.e., it may contain more nodes that are not shown) whose values have not all been computed yet:



Observe that if Node A achieves a value of at least 5, it will not contribute to the value of its parent, as its parent can already achieve a value of at least -5 based on the value of its child that has already been evaluated. Thus, if evaluating one of Node A's children brings Node A's value to at least 5, there is no point in evaluating any more of its children. Furthermore, in order to contribute to the value of the root, Node A must achieve a value greater than -10, as the root can already achieve a value of at least -10 based on the value of its child that has already been evaluated. At first this fact might seem less helpful, as even if one of Node A's children has a value less than -10, one of the other children might have a larger value that can be used. However, consider the evaluation of one of Node A's children. If the child achieves a value of at least 10, then there is no point in evaluating any more of Node A's children.

Thus, we can see that having lower- and upper-bounds on the useful range of a node's value can be useful in avoiding searching parts of a negamax tree. This pruning can greatly improve the performance when the lookahead is several turns. Because we will refer to these bounds as α and β, respectively, we call this technique *alpha-beta pruning*.

We can now specify the negamax algorithm with alpha-beta pruning. Suppose we are given a game position, a lower bound of α, an upper bound of β, and a lookahead of *k*. We will assume that $\alpha < \beta$ (hence, we must make sure this is true whenever we call the algorithm). Then the negamax algorithm will return the following value:

- If the negamax value of the position is less than or equal to α, the value returned will be *some* value less than or equal to α (knowing the exact negamax value is unimportant - we

only need to know that it is no more than α).
- If the negamax value of the position is greater than α and less than β, the negamax value is returned.
- If the negamax value of the position is greater than or equal to β, the value returned will be *some* value greater than or equal to β.

We also need to return a best move (we can return this through an **out** parameter). We specify this move as follows:

- If the position is not the end of the game, *k* is at least 1, and the negamax value *v* of the position is greater than α and less than β, a move to a child whose adjusted negamax value is *v* will be returned.
- Otherwise, any legal move or **null** may be returned.

Note that because the maximum value we have achieved at any point for a given node is a lower bound on any useful value, we can use α to keep track of the maximum value we have achieved so far. We need to consider how to set up the recursion to find the value for a child. If the move to the child does not end the current player's turn, then we just use the same bounds and lookahead. If it does end then turn, then the child is finding a value from the opponent's perspective; hence, we will need to negate the value it returns. We can use returned values *x* such that:

$$\alpha < -x < \beta$$

Or, multiplying by -1 and rearranging:

$$-\beta < x < -\alpha$$

The lower bound therefore needs to be -β, and the upper bound needs to be -α. Furthermore, because a turn has ended, the lookahead needs to be *k*-1.

The algorithm then operates as follows:

- Initialize the best move to **null**.
- If the position is the end of the game, return the value for a loss (utilizing *k*) and the best move.
- If *k* is 0, return the value computed by the evaluation function described under "The Negamax Algorithm" above, and the best move.
- For each legal move:
    - Play the move.
    - If the current player is the same as before the move was made:
        - Recursively get a value for the resulting position, using a lower bound of α, an upper bound of β, and a lookahead of *k*. We will refer to this value as the adjusted value of the child.
        - Otherwise, recursively get a value for the resulting position, using a lower bound of -β, an upper bound of -α, and a lookahead of *k*-1. Negate this value to get the adjusted value of the child.
    - Undo the last move.
    - If the adjusted value of the child is greater than α (which implies that the adjusted negamax value of the child is greater than α):
        - Update α to the adjusted value of the child and the best move to the current move.
        - If α ≥ β (which implies that the adjusted negamax value of the child is at least β), return α and the best move.

- Return α and the best move (if α has increased since the start of the algorithm, it is the negamax value of this node).

Data Structures

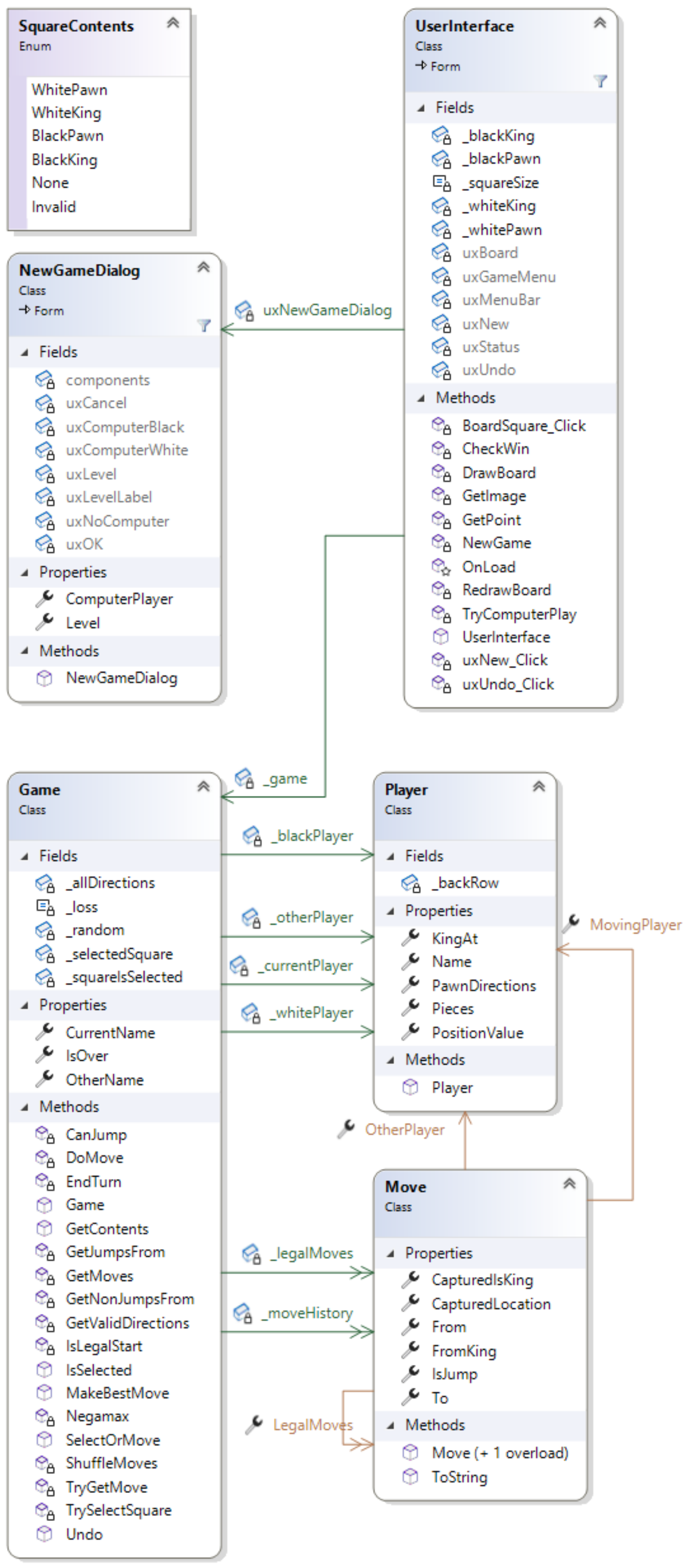
We don't need to represent the negamax trees as data structures. However, because a reasonable lookahead will result in a rather lengthy computation, it makes sense to implement the game representation in a way that facilitates efficient computation. Specifically, the alpha-beta algorithm will make and undo a large number of moves. The most expensive part of making a move is finding all legal moves that result. The implementation for Homework 1 required that every square on the board be examined to determine whether it begins a legal move. We can improve the performance significantly if we can replace the **Checker[,]** with data structures that allow us to access each checker for the current player efficiently.

To this end, we will use, for each of the players, a **Dictionary<Point, bool>** to represent that player's pieces. We will represent a checker by storing in this dictionary a **bool** indicating whether this checker is a king. The key for each of these **bool**s will be a **Point** indicating where the checker is located. Thus, we can efficiently iterate through all of a player's pieces by iterating through the key-value-pairs in the dictionary. Furthermore, we can quickly determine what piece (if any) is at a given location by looking up that location in both dictionaries. Adding and removing pieces can also be done quickly.

Be sure you take advantage of the efficiencies provided by these dictionaries. Specifically, don't iterate through a dictionary unless you need all of the keys and/or values. To access the contents of a single location on the board, use indexing or one of the methods **TryGetValue** or **ContainsKey**.

Software Architecture

The revised software architecture is shown in the following class diagram:



Most of the types shown above were used in Homework 1. **NewGameDialog** represents the dialog shown to start a new game. The **Checker** class is no longer needed - you may remove it if you wish. You will need to modify the **Player**, **Move**, **Game**, and **UserInterface** classes, but not the **SquareContents** enumeration. The required changes are described in what follows.

Coding Requirements

In what follows, we will describe the coding requirements for each of the classes you need to modify. You do not need to use the same names as shown in the class diagram above, provided you follow the naming conventions for this class. If you feel it improves the maintainability of the code, you may break some of the methods into additional **private** methods.

The Player Class

You will modify this by adding the dictionary containing this player's pieces, as described under "Data Structures" above, and code to compute this player's score, as described under "The Negamax Algorithm" above. Specifically, you will need to add a **private int** field storing the *y*-coordinate representing this player's back row. You will need to modify the constructor to initialize this field.

You will also need to add the following **public** properties:

- A property that gets a **Dictionary<Point, bool>** containing this player's pieces (see "Data Structures" above). Use the default implementation.
- A property that gets the player's score (an **int**), as described under "The Negamax Algorithm" above. You will need to implement the **get** accessor using a block of code to compute the score. Be sure to do this efficiently. To square an **int**, do **not** use **Math.Pow**; instead, multiply the value by itself.

The Move Class

Because the **Checker** class will no longer be used, you will need to change the **CapturedPiece** property to a **bool** indicating whether the captured piece is a king (it would also be a good idea to rename this property). As a result of this change, you will also need to change the parameter list for the second constructor so that the **Checker** parameter is now a **bool**.

The Game Class

You will first need to remove the `_board` field and update the code to use the dictionaries in the **Player** objects instead. You should also make the following modifications to the parameter lists of the following methods and use these changes to avoid doing redundant lookups in these dictionaries:

- Add a **bool** parameter to the **GetNonJumpsFrom** method to indicate whether the checker at the given location is a king.
- Add an **out bool** parameter to the **CanJump** method to indicate whether a king will be captured.
- Add a **bool** parameter to the **GetJumpsFrom** method to indicate whether the checker at the given location is a king.

Once you have completed the above changes so that no syntax errors remain, you should thoroughly test your code to ensure it still satisfies the requirements for Homework Assignment 1. Be sure to test the Undo. Ensuring that the human vs. human code works correctly now will make debugging the AI much easier.

At this point, you will need to add a **private const int** field to store the base value of a loss (-200). Then you will need to add a **private** field storing a **Random** to be used for generating random numbers. Initialize this by passing a value of 0 to the constructor. This value is a *seed* for the random number generator. Using a seed will make your code easier to debug because the same sequence of random numbers will be generated each game.

You will then need to add at least three methods to implement the AI. These are described in what follows.

A private method to shuffle the legal moves

This method should take no parameters and return a **Move[ ]** containing the shuffled moves. First copy the stack of legal moves to a **Move[ ]** using the stack's **ToArray** method. Then iterate through the array locations from the end of the array to the beginning. For each location, generate a nonnegative random **int** less than or equal to that location. Use your **Random**'s **Next** method, which takes an **int** as its only parameter and generates a random nonnegative **int** less than this parameter. (Be careful here - the **Next** method generates a value less than its parameter, but you need a value less than *or equal to* the current location. If you can't generate the current location, there are some permutations that can't be generated.) Then swap the value in the current location with the value at the random location you generated. This algorithm will ensure that each permutation of the moves is equally likely.

A private method to implement the negamax algorithm

This method should take the following parameters:

- A **double** giving the lower bound on useful game values (i.e., α from the section, "Alpha-Beta Pruning" above).
- A **double** giving the upper bound on useful game values (i.e., β).
- An **int** giving the number of turns to look ahead (i.e., *k* from the section, "Alpha-Beta Pruning").
- An **IEnumerable<Move>** giving the legal moves. **IEnumerable<Move>** is an interface implemented by both **Stack<Move>** and **Move[ ]**. A **foreach** loop may be used to iterate through an **IEnumerable<Move>**.
- An **out Move** through which the best move will be returned.

It should return a **double** satisfying the specification given under "Alpha-Beta Pruning" above. This method should implement the algorithm given in "Alpha-Beta Pruning". When computing the evaluation function, you will need to cast either the numerator or the denominator to **double** in order to avoid doing an integer division. Be sure to use the **const** you defined for the value of a loss, but incorporate the remaining lookahead as described under "The Negamax Algorithm" above. Also, don't shuffle the legal moves when doing recursive calls. Note that this method doesn't use the **out** parameter returned from its recursive calls (the method below will use it). Be sure your recursive calls don't change the best move.

A public method to make the best move

This method should take as its only parameter an **int** indicating the number of turns to look ahead. It should return nothing. It is responsible for making the best move it can by looking ahead the given number of moves. You will need to use the above method to find this move. In order for that method to return the best move (and specifically, not return **null**) and to avoid wasting time, you need to make sure of the following:

- The game is not over. If it is, throw an **InvalidOperationException**.
- The given lookahead is at least 1. If not, throw an **InvalidOperationException**.
- There is more than one legal move. If there is only one legal move, don't call the above method, as this move is clearly the best one.
- The lower- and upper- bounds for the above method are respectively smaller than and larger than any possible negamax value. **double.NegativeInfinity** and **double.PositiveInfinity** will work.

Shuffle the legal moves using the appropriate method above when you use the above method. Once you have found the best move, call the appropriate method to make this move.

The NewGameDialog Class

You will need to add two **public** properties to this class:

- A property that gets a **string** giving the computer player selected by the user. This selection can be obtained by seeing which **RadioButton**'s **Checked** property is **true**. Return either "Black", "White", or "" (to indicate there is no computer player). It is easiest to implement the **get** accessor as a block of code.
- A property that gets an **int** giving the level selected by the user in the **NumericUpDown**.

The UserInterfaceClass

You will need to modify this class to use a **NewGameDialog** in setting up a new game and to handle the computer player's play if there is one. First, you will need a **private NewGameDialog** field to store a **NewGameDialog**. You will then need to modify three of the existing methods, override one method, and add at least two other **private** methods. These are described in what follows.

The RedrawBoard method

You need to modify this method so that after it has drawn everything, it pauses for at least a half-second in case the computer player is making multiple jump moves. To do this, first call the Refresh method, which is similar to the **Invalidate** method in that it is inherited from **Form** and causes it to be redrawn. The difference in the two methods is that **Invalidate** signals that the form needs to be redrawn at the next opportunity, but **Refresh** causes it to be redrawn immediately. After calling this method, cause the program to pause for a half-second by calling the **static** method **Thread.Sleep**. You will need a **using** directive for the **System.Threading** namespace in order to access the **Thread** class. The method takes as its only parameter an **int** giving the number of milliseconds to wait.

A method to try to play the computer player's turn

This method should take no parameters and return nothing. It is responsible for playing the computer player's entire turn if it is the computer's turn to play. Specifically, if the game isn't over, do the following as long as the name of the current player stored in the **Game** object matches the name of the computer player stored in the **NewGameDialog** object:

- Make the best move using the level stored in the **NewGameDialog** object as the lookahead.
- Redraw the board.

The BoardSquare_Click event handler

Modify this method so that after redrawing the board, it tries to play the computer's turn using the above method. Note that the above method should be written so that if it isn't the computer player's turn, or if there is no computer player, it will have no effect.

A method to manage the "New Game" dialog

This method should take no parameters and return a **bool** indicating whether the user started a new game. It is responsible for showing the **NewGameDialog** and, if the user chooses to start a new game, setting up the new game. Treat this dialog in the same way you would a file dialog: use its ShowDialog method (which it inherits from **Form**), which returns a **DialogResult** indicating how the user closed the form. If the user closes the form with the "OK" button, start a new game by doing the following:

- Construct a new game, as is done in the **uxNew_Click** event handler.

- Make sure the main GUI is visible by calling the **Show** method, which is inherited from **Form**. This method takes no parameters and returns nothing.
- Redraw the board.
- Enable the "Undo" button if there is no computer player; otherwise, disable it.
- Try to play the computer player's turn.

The uxNew_Click event handler

Because the above method now does everything that needs to be done when the "New Game" menu item is selected, replace the code in this method with a call to the above method.

The OnLoad method

You will need to override the **OnLoad** method as you overrode the **OnPaint** method for Homework Assignment 2. This method will be called when the GUI loads as the program is starting. After calling **base.OnLoad**, it needs to display the "New Game" dialog using the appropriate method above. If the user cancels this dialog, exit the program by calling the **static** method **Application.Exit**, which takes no parameters.

Testing and Performance

A program such as this requires a significant amount of testing. Here, we summarize the testing that should be done, but see the demo video for more details. The performance of your program should be comparable to that shown in the video, taking into account the speed of your machine.

- Make sure the New Game dialog works correctly, both at the start of the program and in response to the "New Game" menu item. Be sure to test the "Cancel" button in both cases.
- Make sure the "No computer player" option satisfies the requirements of Homework Assignment 1.
- Test two complete games at Level 1, one game with the computer playing the black pieces, and one with the computer playing the white pieces. This will allow you to test the implementation of the evaluation function.
- Test Level 2 for several moves on both sides of the board. This will help you to determine whether the negamax algorithm has been implemented correctly (although the alpha-beta pruning, if implemented correctly, won't be done below Level 3).
- Test Level 10 against the program at https://gametable.org/games/checkers/ at Expert level. If your program is done correctly, it should usually win, although the games will typically be close. Your program shouldn't take more than a few seconds for each move, although response times will be somewhat slower late in the game.

Submitting Your Assignment

Be sure to **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 nine .cs files (not counting **Checker.cs** if you haven't removed it) within the **Ksu.Cis300.Checkers** folder, and that it is the version you want graded. This is especially important if you had any trouble committing or pushing your code.