Homework Assignment 3: Teaching Mastermind

For this assignment, you are to write a program that teaches optimal play for the game *Mastermind*.

0. Contents

1. User Requirements

Mastermind is a 2-player game in which one player (the *codemaker*) chooses a *code*, and the other player (the *codebreaker*) tries to guess the code using as few guesses as possible. After each guess, the codemaker must provide a *score*, which gives an indication of how close the guess is to being correct. The customer wants a program to help them to break codes optimally. In what follows, we give the rules of the game and define what we mean by optimal play. We then describe the requirements for the program.

1.1. Rules

The game consists of an even number of rounds. In successive rounds, the players alternate between the roles of codemaker and codebreaker.

To begin a round, the codemaker selects a code, hidden from the codebreaker. This code is a sequence of four pegs, each of which can be of one of six colors. For our purposes, these six colors are 0, 1, 2, 3, 4, and 5.

The codebreaker then tries to guess this code by making a series of guesses. After each guess, the codemaker scores the guess with the following two values:

- The number of *black hits*: The number of pegs guessed correctly - the correct color in the correct location.
- The number of *white hits*: The maximum number of additional pegs that would be guessed correctly if the incorrect pegs could be reordered.

For example, suppose the code is `0123`. A guess of `0011` would be scored as 1 black hit (the first peg is the correct color) and 1 white hit (the last three pegs in the guess could be reordered as `101`, yielding 1 additional match).

If a guess is scored as 4 black hits, the round ends, and the codemaker received a number of points equal to the number of guesses made by the codebreaker. Additional rules govern cases in which the codemaker incorrectly scores a guess, or if the codebreaker fails to guess the code within a specified maximum number of guesses; however, these rules are unimportant for this program.

1.2. Optimal Guesses

There are 1296 possible codes. Let's refer to this set of codes as the *initial goal set*. In general, any subset of these codes can be a goal set. We can think of the initial guess as partitioning the initial goal set into nonempty subsets, each corresponding to a different score; i.e., all of the codes that result in a score of no hits are placed in one set, all those resulting in 1 black hit and no white hits are placed in another, etc. Note that because there are 14 possible scores (the score 3 black hits and 1 white hit is impossible), a guess will partition a goal set into up to 14 nonempty goal sets. The score that is actually given for the guess then selects one of these goal sets as the set of possible solutions. Subsequent guesses further partition the set of possible solutions.

Let S be some nonempty goal set. We say that S is 1-solvable if it contains exactly 1 element (we can guess this code immediately). For $n > 1$, we say that S is n-solvable if there is a guess that partitions S into goal sets such that each is either empty or k-solvable for some $k < n$. We define the *optimal score* of S to be the minimum n such that S is n-solvable. We then say that a guess is *optimal* for S if:

- it is the only element in S; or
- it partitions S such that each nonempty goal set has a smaller optimal score than does S.

Example: Let S = {`1122`, `1212`, `2121`, `2211`}, and consider the guess `1210`. This partitions S into the following nonempty goal sets:

- 1 black hit, 2 white hits: {`1122`}.
- 3 black hits, 0 white hits: {`1212`}.
- 0 black hits, 3 white hits: {`2121`}.
- 2 black hits, 1 white hit: {`2211`}.

Each of these goal sets is 1-solvable because it contains only 1 element. S is therefore 2-solvable. Furthermore, it can't be 1-solvable because it contains more than 1 element. Its optimal score is therefore 2, and `1210` is an optimal guess. We also note that none of the elements of S is an optimal guess, as each one partitions S into goals sets such that the set with 2 black hits and 2 white hits has 2 elements.

Our definition of an optimal guess focuses on the worst case. For example, the code that you will write can be used to show that the initial goal set has an optimal score of 5. This means that there is a strategy to break any code in at most 5 guesses. This definition is not the only way of defining optimal guesses. For example, we might consider finding a strategy that minimizes the *expected* number of guesses, assuming codes are chosen randomly with uniform distribution. However, this assumption might not be valid when the codemaker is human. Defining optimality based on the worst case avoids the need to make any assumptions about how the code is chosen.

1.3. Specific Requirements

The customer is only interested in a single round of play, where the user is the codebreaker. The code should be chosen randomly from a uniform distribution and kept hidden from the user. The user should then be allowed to make a series of guesses. As each guess is made, the program should determine if the guess is optimal, as defined in Section 1.2. Optimal Guesses. If not, the user should be given the option of trying a different guess instead. For any guess, the user must also be given the option of allowing the program to make an optimal guess. Once the user makes a final decision on a guess, the score of that guess should be reported.

2. Starting the Assignment

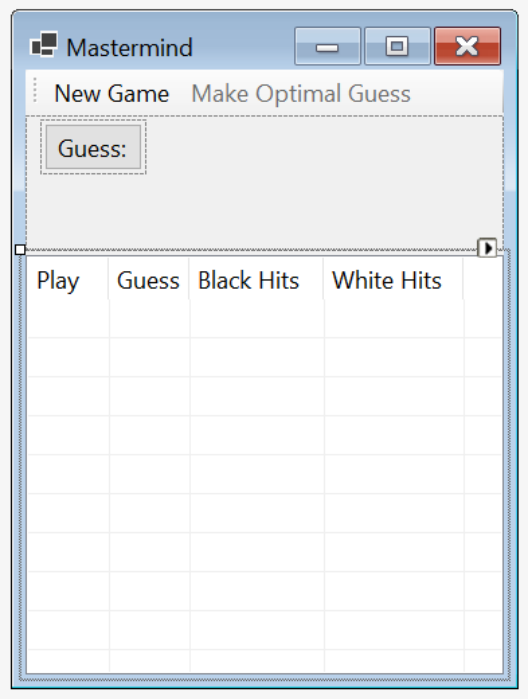
Create a GitHub repository using the link provided in the Canvas assignment and clone it to your local machine. This repository should contain a Visual Studio solution containing a new Windows Forms Application with an additional unit test project, **Ksu.Cis300.MastermindTeacher.Tests**. Note that no code is provided other than the unit test code. You will need to write the entire program, including designing the GUI.

3. User Interface

In this section, we will describe the look and behavior of the GUI that you are to design. We will first outline what you will need to build using the Design window. Then we will describe the required behavior of the GUI. You will need to provide code to implement this behavior (see Section 6. Coding Requirements). The demo video also illustrates the expected look and behavior.

3.1. Manual User Interface Design

Using the Design window, design a GUI resembling the following:



The above only shows a part of the GUI. Modifications will be made by code (see Section 6. Coding Requirements).

At the top is a **ToolStrip** containing two **Button**s. The "New Game" button is initially enabled, but the "Make Optimal Guess" button is initially disabled. Filling the remainder of the window is a **SplitContainer**. Adding it to the form will cause it to fill the remainder of the window. To cause it to split horizontally, set its **Orientation** property to **Horizontal**. You can then drag the bar that splits the container as you wish, but its final location will be set by code (see Section 6. Coding Requirements).

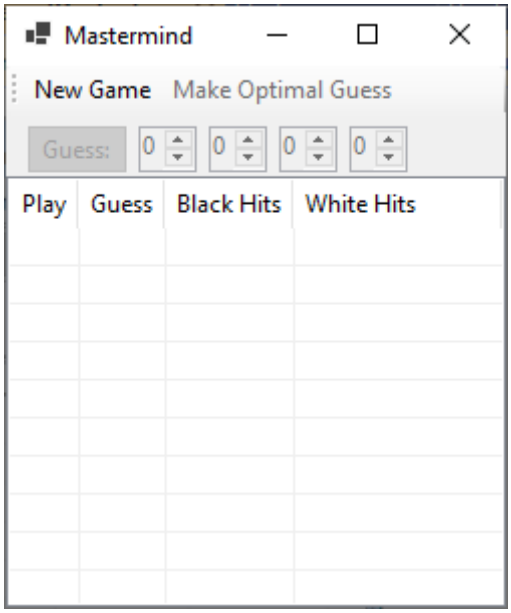
In the upper-left corner of the upper panel of the **SplitContainer**, add a **FlowLayoutPanel**. To this **FlowLayoutPanel**, add the "Guess:" button. Then set the **FlowLayoutPanel**'s **AutoSize** property to **True** and its **AutoSizeMode** property to **GrowAndShrink**.

In lower panel of the **SplitContainer**, add a **ListView**, and set its **Dock** property so that it fills the panel. Set its **View** and **GridLines** properties to display the contents as a grid whose rows and columns are separated by lines. Then edit the Columns property, either through its entry in the Properties list or by clicking "Edit Columns..." below the Properties list. In the resulting dialog, use the "Add" button to add a column. Change the **(Name)** property of each column header so that it follows the naming convention for GUI controls, and change each **Text** property to the text to display in the column header. You can also adjust the **Width** properties to make each header's text fully visible, but the final column widths will be set by code (see Section 6. Coding Requirements).

Finally, resize the form to an appropriate size.

3.2. Behavior of the GUI

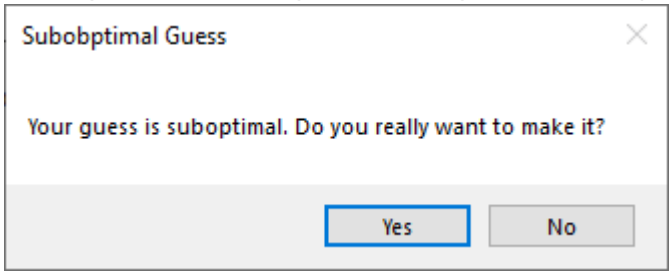
Once the program is finished, it should initially open a window resembling the following:



At this point, the only enabled control is the "New Game" button. Clicking this button at any time should enable the other two buttons and all four **NumericUpDown**s and generate a code randomly, keeping this code hidden from the user.

The user may then repeatedly do one of the following actions:

- Make a guess by entering the four colors into the **NumericUpDown**s and clicking the "Guess:" button. If the guess is not an optimal guess (see Section 1.2. Optimal Guesses), a **MessageBox** resembling the following should be displayed:



  If the user closes the **MessageBox** with the "No" button, nothing on the GUI should change. If the user closes it with the "Yes" button, or if this **MessageBox** wasn't shown because the guess was optimal, the following changes should be made to the GUI:

  - A value 1 greater than the last value in the "Play" column (or 1 if the "Play" column is empty) should be added to that column.
  - The user's guess should be added to the "Guess" column.
  - The score of this guess should be added to the last two columns. If the number of black hits is 4, the "Make Optimal Guess" and "Guess:" buttons, as well as all four **NumericUpDown**s, should be disabled.
- Click the "Make Optimal Guess" button. The program will then choose an optimal guess and update the GUI as above, using the chosen guess in place of a guess by the user.

4. Algorithms

In this section, we'll give the algorithms for finding optimal guesses and for determining whether a given guess is optimal. We first discuss the concept of a *game tree* for a closely-related game. We then present an algorithms for traversing a game tree to find an optimal guess or to determine whether a given guess is optimal. Finally, we give some optimizations to the algorithms.

4.1. Game Trees

Consider the following variation of the game. Instead of selecting a code initially, the codemaker delays selecting the code as long as possible. For each guess, the codemaker's responsibility is to assign a score such that at least one code satisfies all of the scores up to that point. Thus, while the codebreaker is trying to minimize the number of guesses, the codemaker is actively trying to maximize the number of guesses. To find an optimal guess, we need to determine which guess the codebreaker will make, on the assumption that both players are playing optimally.

Defining the game in this way makes it similar to other two-player perfect-knowledge games. Specifically, the players alternate making plays, with one player (the codebreaker) trying to minimize the score, and the other player (the codemaker) trying to maximize the score. The main difference from most two-player perfect-knowledge games is that the two players make their plays under different rules.

Having defined this game variation, we can now define a game tree for any position in this game. When it is the codebreaker's turn to play, the game position will be the goal set consisting of all codes consistent with the scores given to all the guesses made up to this point. When it is the codemaker's turn, the game position will be the partition of the goal set implied by the codebreaker's last guess (see Section 1.2. Optimal Guesses). A node representing a position at which it is the codebreaker's turn to play and the goal set has more than one element will have 1296 children - the partitions resulting from all possible guesses. A node representing a position at which it is the codemaker's turn to play will have from 1 to 14 children - the goal sets selected

by all legal scorings of the guess. The leaves are the nodes representing positions at which it is the codebreaker's turn and the goal set contains only one element.

Example: Suppose it is the codebreaker's turn to play, and that the current goal set is {`1122`, `1212`, `2121`, `2211`}. Here is a portion of the game tree from this position: