MASTERMIND

Play setup

- Open the scene called Level0 which should be located under “Assets/Scenes”
- Set the aspect ratio to 16:9 under the Game tab inside the Unity Editor so that all the UI elements appear as they are intended.

Control/Play Scheme

- Once, Unity starts playing the scene, **enter 4 digits, each in range 1 to 6** for the secret code in the **input box** on the bottom and press **Enter**
- The AI will try to guess the answer
- Once, the AI finds the answer, you can re-enter a new code
- Press **Escape** to **stop** playing

Rules

- Numbers are used to represent 6 different colors from the actual mastermind game
- A Two player game (Codemaker & codebreaker)
- The code maker chooses a pattern in the form of 4 digit secret key
- Duplicate digits are permitted
- The codebreaker tries to guess the pattern, in both order and color (digit), within ten rounds
- Codemaker provides a score once the guess has been made in the form of a black key and a white key
- A black key peg shows for each code digit from the guess which is correct in both color (digit) and position in the answer code
- A white key peg indicates the existence of a correct color (digit) code peg placed in the wrong position
- The number of blank key pegs determines the number of wrong colors (digit)
- The guesses and feedback continue to alternate until either the codebreaker guesses the pattern correctly, or the number of turns that the codebreaker can use to guess crosses ten

My Solution

Codebreaker

Uses Knuth's Algorithm

A) Initialize the codebreaker AI - void Init(getParameters(colors, numOfPegs, rounds, etc.))
   - a) Create a list of all possible answers considering the number of colors and the number of pegs. In our current case it creates 1296 possible codes (1111, 1112 ... 6665, 6666).
   - b) Set current round to 0
   - c) Create a guess-answer-score table. The answer score pair is calculated based on how each answer compares with a guess to generate a score. Thus generating 1296 row entries and each row will have 1296 possible answers along with their respective scores.

B) Generate a guess - Play Guess()
   - a) Increment the current round number (+1)
   - b) If it is the first round, play 1122 as a guess and get the score. As Knuth demonstrated in his paper, the guess 1122 emanates from bruteforcing almost all values to win in the minimum rounds possible.
   - c) Otherwise, get the current answer set and remove all the answers which will not generate the same score that was generated in the previous round.
   - d) Create a list that will hold all possible guesses
   - e) Each guess will contain the following data. Maximum possible probability of a score from all the scores considering all answers, a boolean to indicate whether the answer is present in the current answer set and the actual guess itself.
   - f) From the generated guesses list find the guess that has the least possible probability of it being an answer. Also check if the minimum probable guess is present in the current answer set; if it is then we have found the most accurate possible guess and return the guess, thus eliminating the need to check for other answers. If the previous check was invalid just return the minimum probable guess. That will make sure that most of the impossible answers are eliminated from the current answer set.
   - g) Calculate the score from the current guess and if it is "BBBB" the correct answer has been found.
   - h) Repeat from step (c) to step (h) until the correct answer is found (or current rounds are greater than 10).

Codemaker

A. Score calculation - CalculateScore(guess, answer)

a. For each character in the guess string check if it is at the same position in the answer string. If it is then append a “B” to the score. Otherwise store the guess character in a wrongGuess list and the answer character in a wrongAnswer list.
b. For each character in the wrongGuess list check if it is present in the wrongAnswer list. If it is then append “W” to the score and remove the character from the wrongAnswer list (we don’t want to check for the same character again).
c. Return the score.

Further Improvement

A. Object pooling pattern for the storing the possible answers
  a. In my current solution, I have created a “master version” of all the answers in the start and then when calculating the current possible answers I copy the contents from the master version to the list every time the game resets.
  b. A better version would be to create a master version of all the possible that can be used as an object pool. It can contain a boolean that stores if the answer is inUse.
  c. Once the guess and it’s score is calculated, all possible answers which won't be used can be set inUse = false.
  d. When iterating over the possible answers only the answers with inUse = true can be used for further checks. Thus eliminating the need to create a copy of the huge list.
  e. And when the game is reset all the answers in the possible answers will have the boolean reset, inUse = true.

B. Multithreading
  a. Calculation of the maximum possibilities for the score of a guess is a tedious task if it is given to a single thread.
  b. Instead each thread can find the maximum guess probability independently. Therefore, each thread can be given (total guesses) divided by (thread count) number of guesses to work on.
  c. Thus minimizing the time required to calculate a guess.

References

- A talk on Beating Mastermind by Adam Forsyth:
  RubyConf 2018 - Beating Mastermind: Winning with the help of Donald Knuth by A…
- Mastermind Board Game Wiki: https://en.wikipedia.org/wiki/Mastermind_(board_game)
- The Computer as Mastermind by Donald Knuth:
  https://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf