Project Description: CSCI 350

1. Goal

Build an expert Mastermind player in Lisp to play in our tournaments. The system chooses a "secret code" and your program tries to guess that code. You can read about *a simple version of* Mastermind on the Wikipedia page on Mastermind.

2. Environment

The standard 4,6 version of the game uses a code with 4 pegs, each of which is one of 6 different colors (denoted here by letters). In a single game, your player will get up to 100 guesses and 10 seconds to determine the secret code, whichever comes first. The system returns your guess with a scoring list: the number of exact pegs (correct color in the correct position) and the number of "almost" pegs (correct color in the *wrong* position). For example, the guess (B B A C) might receive the system response (2 1). This means that two of the pegs are the right color in the right position and one more peg is the right color in the wrong position. (Ah, but which ones?) Actually, you will often get a 3-element list in reply to a guess in our tournaments; this is explained in Section 5.

The game can be made arbitrarily more difficult by increasing the number of code pegs and the number of colors. The tournament engine provides for any (positive integer) number of pegs, and any (positive integer) number of colors up to 26. Your program will not just be playing the 4,6 version. Your job is to write a Mastermind guesser that can ***play any size game well against any secret-code generator***. (More about those generators appears below.)

3. Evaluation

Your program will play a set of tournaments. Each tournament is a set of 100 games that pits your program against the same Secret-Code Selection Algorithm (henceforward, **SCSA**) for some fixed number of pegs and colors. Your program can determine the name of the SCSA during the tournament.

The program with the highest score in a tournament wins that tournament. Winning in fewer guesses is better. If your program makes an illegal guess (wrong length or illegal colors), the game ends and you lose 2 points. The scoring function is

$$5\sum_{wins}\frac{1}{\sqrt{guesses-to-win}}-2(\#\,rounds-with-illegal-guess)$$

4. Guessing strategies

For 4 pegs and 6 colors, there is a 5-guess solution by Donald Knuth, described on the Wikipedia page on Mastermind. The 5-guess approach exhaustively enumerates all possible codes, and eliminates those inconsistent with the responses to the guesses as they are received. It then chooses a guess with the highest score, the one that maximizes the possibilities that could be eliminated by any of the possible responses. Of course, **this strategy does not scale**. There are 6^4 possibilities for the standard game, but in general $colors^{pegs}$ possibilities. Like many fun AI problems, Mastermind is NP-complete. (See Stuckman and Zhang.)

Here are 4 **baseline strategies** that may not scale either:
#1: Exhaustively enumerate all possibilities. Guess each possibility in lexicographic order one at a time, paying no attention to the system's responses. For example, if *pegs* = 4 and *colors* = 3, guess (A A A A), (A A A B), (A A A C), (A A B A), (A A B B), (A A B C), and so on. This method will take at most $colors^{pegs}$ guesses.
#2: Exhaustively enumerate all possibilities. Guess each possibility in lexicographic order *unless* it does not match some previous response. For example, for *pegs* = 4, if guess (A A A A) got (0,0) then you would not guess anything with an A in it.
#3: Make your first *colors*-1 guesses monochromatic: "all A's," "all B's,"… for all but one of the colors. That will tell you how many pegs of each color are in the answer. (You don't need to actually guess the last color; you can compute how many of those there are from the other answers.) Then you can generate and test only answers consistent with that known color distribution.
#4: SIGART article by Rao has a heuristic algorithm that is probably not optimal. I've posted it on the class website.

These four are good baselines against which you can compare the performance of your more sophisticated strategy. The web is filled with ideas, but then again, so are your own heads.

5. Challenges

#1: Implement a general-purpose player. Your program should be able to play for any number of pegs and colors. It should also make reasonable guesses based on the responses it has received. Feel free to read and look about on the Web, and draw inspiration from it. You must, however, *cite your sources properly*. **See the handout on how to avoid plagiarism.**
#2: Implement a scalable player. As the number of pegs and colors increases, any given algorithm will take longer to make each guess, and more guesses will be required. Scalability is measured in time and in number of nodes expanded. *You do not want to make the same guess more than once*, so you probably want to generate a list (or a partial list) of guesses and work through them methodically. (You could also generate one guess at a time, but that might be hard to do methodically.) To facilitate this, *the return on a legal but non-winning guess is a three-element list*, where the third element is the guess number. That is (2 3 5) means that your 5th guess had 2 correct-color correct-position pegs and 3 right-color wrong-position pegs. Feel free to define variables and/or data structures to help you remember what has happened in the current game, and don't forget to initialize them every time a game begins.
#3: **Learn the system's strategy for choosing the secret code.** In some tournaments, the system may use a biased (i.e., not purely random) SCSA. Your program should try to detect the SCSA's code-generating strategy so that it can make better guesses. The SCSA may only use certain colors, or always choose codes with exactly three colors, or always choose codes that alternate colors, or never use the same color more than once, or always put the same color in the first and last places, or always place colors in a certain order (e.g., A is always before B), or prefer codes with fewer colors but occasionally use more colors to mislead you, or have a probabilistic preference for fewer colors (e.g., with probability 0.5, use exactly 1 color; with probability 0.25, use exactly 2 colors, …). SCSA strategies will include, *but not be limited to*, those described above. *Think carefully* about your learning feature space and algorithm.

6. How to use the code

To set up a game, first run the function Mastermind with the number of pegs, number of colors, and SCSA of your choice. For example, `(Mastermind 7 5 'two-color-alternating)` builds a global variable *Mastermind* that describes a 7-peg, 5-color version that alternates exactly 2 colors. When you are playing, the answer you are trying to guess is stored in *Mastermind*. It won't be there when I test your code, of course.

In the code file you will also find:
• Eight SCSAs: insert-colors, two-color, ab-color, two-color alternating, only-once, first-and-last, usually-fewer, and prefer-fewer. If you execute an SCSA, you get a hidden code.
• The function `SCSA-sampler`, which will give you secret code samples from all the ***provided*** SCSAs. You can use it to generate a set of codes from the same SCSA and then practice on them. For example, you can get 25 7-peg codes on 5 colors generated by two-color-alternating with `(SCSA-sampler 25 'two-color-alternating 7 5)`.
• The play-tournament function so you can run your own practice sessions just the way I will run the tournament. To watch the team called RandomFolks play 25 games of 7-peg, 5-color Mastermind against the SCSA called two-color-alternating, for example, you would use `(play-tournament *Mastermind* 'RandomFolks 'two-color-alternating 25)`, *but every time you run it this way, it will be using different codes*. If you want to test against the same codes repeatedly, you will need to store them. *Read the comments on* `play-tournament` *carefully*.

To make things more interesting, there are also five **hidden SCSAs**. A set of 100 sample codes for 7 pegs and 5 colors that each of them generated is on the website, along with the *names* of the hidden SCSAs that generated them. You will want to practice on these codes too. Your algorithm should be parameterized to work against each of these by name. (In other words, any learning you do should be *before* the tournaments.)

All teams should design a guessing strategy that is expected to do well on challenge #1. However, I expect that different teams will choose to focus more of their energy on either challenge #2 or #3. Of course, you should have *some* solution that is plausibly scalable (works for any size problem at least in theory), and *some* learning approach (even if it is quite simple and limited). Teams of 3 or more people will be expected to have good designs for all three categories. Also, *all of the project writeups (as discussed below) must address all three challenges and what approaches you designed to try to meet them*.

7. Project requirements and grading

You will be primarily graded on the *thoughtfulness and clarity of your design and presentation*, and not primarily on your algorithm's performance. This gives you the freedom to try a risky approach that is interesting from a design perspective but might not work very well. An approach that does not work very well, and is also naive, trivial, or not well-motivated will receive a correspondingly trivial grade. The most important part of this project is to ***produce a working program with a strong justification and a clear design beyond the baseline strategies*** for your player. Winning is nice but it won't earn you many points.

• **Baseline player implemented in Lisp (5 points** for each of the 4 baseline strategies listed in Section 4). *You are only required to implement one.* Extra credit will be given for a *correct* implementation of any others.
• **Your team's tournament player implemented in Lisp** (**35 points**):
15 points for correctness (whether the implementation matches the solution described in your paper)
15 points for design (generality, clarity, and elegance)
5 points for code readability (indentation, comments, modularity)
• **Project report (50 points)**: Each team must submit *one copy* of a clearly-organized, *typed* project report that describes your approach, your experience in designing and implementing the approach, and the performance of your system. This must be a *minimum of 5 pages* including:
5 points for a **survey** of any background reading you did on the game and strategies, *with citations*.
10 points for a **discussion** of how you constructed guesses and how you learned for biased SCSAs. *Provide explicit citations* for any ideas you drew upon or borrowed directly from the literature.
10 points for some **theoretical analysis** (mathematically formal would be nice, but is not required) of the computational complexity of your algorithms, and the number of expected guesses (which could be based on the degree to which each guess is expected to reduce the size of the remaining solution space) in terms of the size of the problem.
25 points for an **experimental evaluation** of your program with respect to the three challenges. Report performance *results for your program and at least one of the 4 baseline strategies* in *all* of the following:
• The random SCSA (insert-colors) and at least 5 of the known biased SCSAs in tournaments of 100 rounds for the 8,10 problem. ***You may find this is quite slow.*** If you cannot get results in a reasonable time, you may reduce this to 7,9 or even 6,8, but that means you have not thought your guessing strategy through very well.
• The scalability challenge in a series of tournaments of increasing difficulty (*pegs* × *colors*). You choose the problem sizes. *You will understand scaling better if you start with small values for pegs and colors and work your way up.* You should do this for more than one SCSA.
• The learning challenge, using data from the SCSAs provided, both the known and the hidden ones.
Present all results clearly using tables and/or charts, scatterplots, and/or statistics (e.g., means, standard deviations, confidence intervals). In each case *provide information on both CPU time and guesses*. (Hint: Use the Lisp function `time`.)
• Class tournament (10 points)
5 points for a program that runs successfully during the tournament
5 points for how well it performs in the tournament
All tournament rounds must run on the G lab machines, so you should be certain to **test your code there** even if you develop it on a different machine.

8. Deadlines

DEADLINE 1: One functioning baseline player with a sample of its tournament output. It doesn't have to win but it does have to run. Send both the code and the output by **email** to susan.epstein@hunter.cuny.edu **before class on the date specified on the class website**.

DEADLINE 2: Your near-final tournament player with successful (4,6) and (4,7) tournament *output against 2 non-trivial players*. Send both the code and the output by **email** to susan.epstein@hunter.cuny.edu **before class on the date specified on the class website**.

DEADLINE 3: The full project is due with the **report in hardcopy and the code in email before class on the date specified on the class website.** I must be able to compile and execute your code to run the tournaments.