Mastermind V2

(AI Edition!)

Joshua Grizzell

CIS-7-48647
Fall 2021

Introduction

Title: Mastermind Version 2

Mastermind is a famous mind game that has a very simple concept: figure out the hidden code. The player gets ten turns to figure out the code. Each turn, the player guesses a code, and the game tells you information based on the guess. In this version of the game, an AI will solve the code by itself.

The AI must solve a code that can have 10 possible digits (0-9) in a code with the length of four.

Summary and Discussion

Part 0: Changes in Version 1a

So, I feel like I should include this because I heard that we were supposed to do a write-up on version 1a and I simply didn't know until after the fact. I'll briefly touch on what was added in version 1a, so feel free to skip ahead if it's of no consequence to you.

Beginning with the final iteration of version 1, I added a new function called `automnd` that the user can call at the beginning when I ask if they want the game to play itself. The function runs through codes 0000 to 9999, incrementing by 1 each time, and then dumps out statistics of its findings at the end.

```
How many times should 4 right come up?: 1
How many times should 3 right come up?: 36
How many times should 2 right come up?: 486
How many times should 1 right come up?: 2916
How many times should 0 right come up?: 6561

How many times should 4 close come up?: 9
How many times should 3 close come up?: 320
How many times should 2 close come up?: 2178
How many times should 1 close come up?: 4404
How many times should 0 close come up?: 3089

How many times should a sum of 4 come up?: 24
How many times should a sum of 3 come up?: 720
How many times should a sum of 2 come up?: 3540
How many times should a sum of 1 come up?: 4420
How many times should a sum of 0 come up?: 1296
```

Figure 0.1: The results at the end of the function automnd.

I had to adjust existing functions and create new ones. Below is a table of what was added and changed.

	Original (v.1)	Modified (v.1a)
	void mstrmnd(int codeLen, bool dupes);	void mstrmnd(int codeLen, bool dupes);
➕	int* genCode(int arr[], int codeLen, bool dupes);	int* genCode(int arr[], int codeLen, bool dupes, int range);
	void clear();	void clear();
	bool chkDgts(int num, int codeLen);	bool chkDgts(int num, int codeLen);
	int* cnv2Arr(int n, int arr[], int codeLen);	int* cnv2Arr(int n, int arr[], int codeLen);
	string cmpArr(int arr1[], int arr2[], int codeLen);	string cmpArr(int arr1[], int arr2[], int codeLen);
	void prnHist(int gHist[], string rHist[]);	void prnHist(int gHist[], string rHist[]);
➕		void automnd();
➕		char* genCode(char arr[], int codeLen, bool dupes);
➕		char* cnv2Arr(int n, char arr[], int codeLen);
➕		int* cmpArr(char arr1[], char arr2[], int result[]);

Figure 0.2: Changes to existing functions and added functions.

You can see that instead of outright removing or changing some of the existing functions like `int* genCode` or `int* cnv2Arr` I created new functions with different signatures, such as `char* cnv2Arr`. This was a bit challenging. I think that's everything, though. Now, onto the actual AI.

Part 1: Brainstorming

After getting the code stub from the class github repository, I found myself having a ton of trouble figuring out how to start. After getting the stub from the class github repository, I sat for hours trying to figure out just how to get the AI to know what to do. I decided I needed to look more into the theory behind Mastermind before trying to code in anything.

I looked into the Knuth algorithm, which is pretty great as a computer algorithm, but it seemed beyond me, and I wasn't sure how I would implement it with my current skill set. (Note: I think, having completed this project, I could definitely do this if I put enough effort into it!) I kept researching methods to solve Mastermind and found a few resources that helped me. However, it still wasn't enough to begin coding.

The best way to figure out how to code the AI, I thought, was to play several games of mastermind myself and take notes on what I was doing, and why I wanted to do it. This was by far the biggest help, because it let me finally see how to begin coding the AI. The key is the amount of correct digits and the amount of close digits. If the AI can react to this, it can solve Mastermind.

Part 2: A Burgeoning AI

Having finally figured out how to start the AI, I began coding. The first thing I wanted the function to do was to cycle from 0000 to 9999, incrementing by 1111 each time. I created the static integer `num` that would increment at the end of the AI function every time it was called. Each character in the char array sGuess would equal num plus 48. This meant each time the function was called, each digit would increment by 1.

Next up, making it actually react to the game. I used the `rr` and `rw` variables (the amount of correct digits and close digits) passed in through the argument to trigger if statements. If all four digits had not yet been found, and `rr+rw` was greater than zero, I would add the current `num` integer to an array called `fgDigit` (meaning foreground digit.)

```
 Guess     rr rw
0000  |    0  0   rr+rw = 0, no digits are added
1111  |    1  0   rr+rw = 1, one digit 1 is added
2222  |    1  0   rr+rw = 1, one digit 2 is added
3333  |    2  0   rr+rw = 2, two digits 3 are added

          static int fgDigit[4] = 1233
```

Figure 2.1: How the AI function reacts to correct digits. Looking back now, I suppose rr + rw is pretty redundant. It only ever had to pay attention to correct digits.

There was one big problem, though: it would add the wrong digit. Specifically, it would add a digit too late, meaning that when I meant to add 1 to the `fgDigit` array, it would add a 2. One of the reasons the problem was occurring was because I was putting all of my new code AFTER the return value of the function, sGuess, was set. I fixed the problem by changing `num` to `num-1` when adding to the `fgDigit` array, and by moving all of my new code to be above the guess calculations. Additionally, even though this didn't change much, instead of using the `rr` and `rw` values to check for how many digits in the guess were correct, I used the `grr[guess]` and `grw[guess]` values.

A lot to take in. Hopefully that last part made sense. If not, my bad, you can instead check out the difference between version 1 and version 2 in the ZIP file. At this stage, the AI can figure out which digits are in the final code, but not much else.

Part 3: Cracking the Code

Once all four digits have been found, the AI function would then add each of these digits to the `sGuess` array and guess using those digits. They would also switch a boolean variable `solveState` to true. Looking back now, this variable was only used for one thing: ensuring that the AI only guesses the digits it found in the `fgDigit` array, and only guesses them once. I didn't want the AI to only guess one specific combination of digits over and over again, so I made `solveState`.

The AI function then switches from simply incrementing each digit by 1 each time it's called to swapping the existing digits. I created a swap function However, I quickly figured out that blindly swapping digits was not a good idea. I decided to create several new static variables: `int x, y` and `string bGuess`.

`bGuess` stands for best guess, and it keeps track of the guess which yielded the most correct digits. Integers `x` and `y` are both used for indexing different spots in `sGuess` when swapping, and are increased and decreased depending on the situation. It's easier to see how these variables work more specifically in the flow chart, but for now, just understand that they're for swapping digits. Keeping up with all of these variables? Here's a handy guide. Don't worry, I won't add any more after this.

Static Vars	Description
num	Integer that increments upon each call of the AI function.
found	Integer that increments on 2 conditions: grr[guess]+grw[guess] < 0, and found < 4.
fgDigit[4]	Integer array that holds all found digits.
solveState	Boolean that initializes to false. Set to true when all digits are found/found == 4.
x	Integer used for swapping digits in sGuess. It'll be explained more shortly.
y	Integer used for swapping digits in sGuess. It'll be explained more shortly.
bGuess	String that holds the best guess. Determined by the variable rr.

Figure 3.1: Static variables I've added to the AI function.

Part 4: Optimization and Theory

The AI is now essentially complete. It can figure out the code without error, but it could be faster. One thing I did was change bGuess from a string, like sGuess, to an int for indexing. Rather than just copying the string of the best guess and putting it in a variable, I would have bGuess be an index, so as to find all properties of the best guess, such as the amount of digits were in the correct place. Furthermore, if I ever wanted to access the string like before, I could just use aGuess, which has a history of all previous guesses, and use bGuess as an index to find the string. This was a great change, because it optimized the bad criteria for finding the best guess.

Another small optimization: if the two digits to swap during the solving phase were identical, I would make the AI move onto the next possible swapping combination. Every turn counts!

There are so many other things I could do to make this AI even better. I tried to include a few lines of code that made it so that after guessing 8 in all digits, if there are any digits yet to be found, the AI would know that they would have to be 9. However, this would just end up backfiring and I couldn't figure out how to fix it in time. It's a shame, too, cause that would've been an easy turn to save!

Overall, this was an amazing project and I am incredibly proud of my work, however shoddy it may be.

Pseudocode

string AI

Define functions print & swap using lambdas
Declare all static variables.
Save the last guess's correct digits and close digits in grr[guess] and grw[guess].

If all four digits have been found
    If the last guess's correct digits > the best guess's correct digits
    Set the best guess index to the current guess index

If the AI is currently solving (boolean solveState is true)
    Set the current guess to the best guess.
    If x is equal to 2
        Set x and y equal to 0
    While the two digits to be swapped are identical
        Increment y by 1
    If x is equal to y
        Increment y by 1
    If y is greater than 3
        Increment x by 1
        Set y equal to x + 1
    Swap two digits of the current guess using integers x and y as indices.
    Increment y by 1.

If a number has been found while they have not all been found yet
    Set a variable queue equal to the last guess's correct digits + close digits.
    While queue is decreasing to 0
        Set a slot in an array of found digits = num - 1, using total found digits as an index.
        Increment the amount of found digits by 1.

If the amount of found digits is less than 4
    Set each container in the current guess string equal to num. (+48 for char)
Else if the amount of found digits is 4 and the code is not yet being solved (solveState is false)
    Set the current guess string equal to each digit in the found digits array.
    Set the best guess equal to the index of the current guess plus 1.
    Set the boolean solve state to true.

Save the guess string in the history array.

If the amount of found digits is less than 4
    Increase the integer num by 1.

Return the guess string.