Nidhi Jaju
CID: 01400151
Mastermind Report

Mastermind Report

An explanation and evaluation upon the methods used to program the game for most efficient implementation

For the third computing assignment, we had to write a program for the popular guessing game, "Mastermind". This included writing code for giving feedback on how many black hits and white hits each attempt has, learning from the feedback to eliminate the number of possible attempts, and creating attempts. This game although has one main goal, which is to find the original sequence as soon as possible, has many different methods that it could be implemented through.

Explanation & Evaluation of Functions:

void give_feedback (const std::vector<int>& attempt, int& black_hits, int& white_hits)
For the implementation of this function, I mainly just followed the approach explained by Knuth about how mastermind works. The code for black hits was relatively straight forward in that you just go through each element of the attempt vector and the sequence vector respectively and if they're the same, increment the value of black hits. For white hits, I first tried to write an algorithm to traverse the list and look for each number in the given attempt to provide me with the white hits directly. However, the logic provided in Knuth's paper, although forces one to declare a new variable to calculate total hits, which as a result allocates more memory, it is much more efficient to find white hits this way, as you have to traverse the list of all the possible attempts less times than if you find white hits directly. The total hits is calculated by traversing through the two codes that one wants to compare, and counting how many times each number appears in each code, and taking the smaller count, where n_i be the number of times the "number" appears in the codeword, in the following manner:

$$\min(n_1, n'_1) + \min(n_2, n'_2) + \ldots + \min(n_{num}, n'_{num})$$

The difference between total hits and black hits is then calculated to give white hits.

void init (int i_length, int i_num)
The given function already assigned the values for num and length to the respective variables from the user input. I added some additional implementation that creates a list with all the possible attempts with the inputed length and numbers. I do this by starting with 0, and just adding one to it all the way up until num^{length} in base 10, which is then pushed back into the "possible" vector one by one. I do this to set up a bank from where the create_attempt function can pick an attempt from each time.

I was initially planning on creating a vector of vectors of int values that has the list of all the possible attempts where each digit is stored in a int value. I did this by first starting off with 0, and then adding and as soon as it reached the 'num' value, carry over to the next digit up. While this would have been a perfectly viable approach to creating the list of possible attempts, I realized that not only was this approach using up a lot of memory, therefore making it inefficient, it was also getting complicated.

Therefore, I decided to stick with just creating a simple list which I could then extrapolate numbers from and get them in the correct base using the function, "getnthcode", every time I needed to get access a specific codeword in the list.

I also declared a variable called "total" which calculates the number of possibilities for a given num and length. This is done so that I could set conditions to use different optimization algorithms.

void create_attempt(std::vector<int>& attempt)
In this function, at first I just randomly generated an index from the possible vector and use that as the attempt. This is the most simple implementation which relies completely on the learn function to eliminate possible codewords from the list.

Further optimisations such as specifying the first guess for attempt, using a game theory algorithm called "max parts", and using an incrementer method, which although is relatively inefficient, is much faster in terms of computation period in seconds. These were added to make the approach to selecting an attempt more effective, which are explained in the 'Optimisation' section of this report.

void learn(std::vector<int>& attempt, int black_hits, int white_hits)
The learn function essentially takes in the feedback of input from the attempt that was just inputed into the game and compares it to the feedback of every other possible attempt in the possible vector, and keeping only the codewords that have the same feedback as the attempt just inputed.

For example if num = 4, length = 2,

test against previous attempt = 13 (B = 1, W = 0)

Codeword	B	W	Action
00	0	0	discard
01	0	1	discard
02	0	0	discard
03	1	0	**keep**
10	1	0	**keep**
11	1	0	**keep**
12	1	0	**keep**
13	2	0	discard
20	0	0	discard
21	0	1	discard
22	0	0	discard
23	1	0	**keep**
30	0	1	discard
31	0	2	discard
32	0	1	discard
33	1	0	**keep**

void getnthcode(int num, int length, int n, std::vector<int>& attempt)
As the list of all possible attempts is stored in terms of integers in base 10, this function is implemented to essentially convert to the correct base for the game, and extract the codeword to a vector where each number is stored as an integer value itself. This is done using the mod and divisibility operator, then inserting the remainder into the vector. This is using the same logic for when one extracts a digit from an integer value to a string.

For example, for converting 1046 from the list of all possible attempts, when num = 4, length = 6,

1st iteration: 1046 % 6 = **2** 1046 / 6 = 174	possible: 2
2nd iteration: 174 % 6 = **0** 174 / 6 = 29	possible: 0 2
3rd iteration: 29 % 6 = **5** 29 / 6 = 4	possible: 5 0 2
4th iteration: 4 % 6 = **4** 4 / 6 = 0	possible: 4 5 0 2

The next while loop is just implemented as a way to add zeros to the beginning of the the possible vector if it is shorter than length, to make sure the codeword is the length that it is supposed to be according to the game.

void feedback_attempt(const std::vector<int>& temp, const std::vector<int>& attempt, int& blacktemp, int& whitetemp, int num)
This function is almost the same as the give_feedback function, except I had to declare it again as a global function. This is because the give_feedback function is in the maker structure, but the learn and create_attempt function in the solver structure need to access it. Thus, it is the same with just one difference of passing num as a input parameter as well.

Optimisation:

First Attempt

After doing some research, I found out that it is proven by many different people that if the first attempt is specified by the program itself, instead of randomly generating the attempt, it can help in making the solver program more efficient in guessing the right attempt. I did this by running the program with different inputs each time and seeing that there was a pattern in the code that resulted in lower number of attempts taken to guess the original sequence:

Length	Attempt
2	02
3	012
4	0012
5	00112
6	000112
7	0001112
8	00001112
9	000011112
. . .	. . .

Determined Pattern for push_back order for first attempt:

For length is <u>even</u>,

Digit	No. of the Digit
0	length / 2
1	(length / 2) - 1
2	1

For length is <u>odd</u>,

Digit	No. of the Digit
0	length / 2
1	length / 2
2	1

Max Parts

To make my program more efficient in terms of number of attempts taken to guess the sequence, I used a game theory strategy by Barteld Kooi called "max parts". In this strategy, I compared two copies of the "possible" vector and got the feedback comparison for the first codeword in the first list with all of the codewords in the second list. I repeated this for all codewords in the first list. This helped me to calculate the number of "parts", which is the number of different combinations of black and white hits I got for that particular codeword with all other codewords in the second list. While going through the first list, I keep track of the position of the codeword that provides the maximum number of parts, as this is the one that provides the most information about the attempt. Therefore, this will eliminate the most codewords from the list, resulting in a shorter list to choose an attempt from, and hence less guesses needed to decode the sequence.

"Incrementer" Algorithm

While testing, I realized that the simple code and the max parts algorithm took too much time to execute for big integers such as num = 9 and length = 9. As a result, the incrementer algorithm was implemented to ensure the program could run for larger sequences as well, without timing out. One tradeoff was the number of attempts it takes for the sequence to be correctly guessed though. The main logic for the algorithm is the following:

For example for num = 4, length = 4:

1. Start with 0000 and compare the feedback from the codeword (0000) and the previous attempt
2. If the number of black hits is the same, increment the first digit (e.g. 1000), and repeat step 1. If the number of black hits increased, it means the number is correct, and you move on to the next digits place. If the number of black hits decreased, it means the number is 0, and you move on to the next digits place.
3. Keep incrementing the digit, and act accordingly to the conditions in step 2, until you have four black hits.

The main aim of this algorithm is to test each digits place with each number in order from 0 to num in a very systematic manner. This results in larger number of attempts, than comparing the feedback for each codeword with the list as done for max parts and the simple method. However, as less memory is allocated, as there is no temporary lists or vectors to collect information being created, the computation time is extremely fast.

Performance Statistics:

(I ran my code for 100 games to get the average attempts and time per game.)

Length	Num	Algorithm	Average Attempts	Average Time (s)
2	**6**	maxparts	3.708	0.004788
3	**6**	maxparts	3.972	0.009394
4	**6**	maxparts	4.391	0.100519
5	**6**	random	5.054	0.020553
6	**6**	random	5.478	0.116190
7	**6**	random	5.900	0.678820
8	**6**	random	6.470	4.096790

Length	Num	Algorithm	Average Attempts	Average Time (s)
2	**6**	increment	6.101	0.002163
3	**6**	increment	8.876	0.002135
4	**6**	increment	11.277	0.001579
5	**6**	increment	14.095	0.001604
6	**6**	increment	16.783	0.002535
7	**6**	increment	19.399	0.003232
8	**6**	increment	23.453	0.004310

As it can be seen from the above tables, although the elimination method give lower averages, it takes an extremely long time for each game to be completed for larger sequences, which is why I decided put a condition on the code, so that it runs the increment algorithm for larger games.