MASTERMIND SOLVER USING C++

1. INTRODUCTION

1.1. MASTERMIND GAME-

Mastermind is a code-breaking game for two players. The famous board game version was invented in 1970 by Mordecai Meirowitz, an Israeli postmaster and telecommunications expert. It is thought to have been based on the century old pencil and paper game Bulls and Cows. Invicta Plastics Ltd bought the intellectual property rights to the game in 1972 and subsequently brought the game to the market. In its standard form, Mastermind is played with 4 pegs and 6 colours, while another popular version exists, Super Mastermind, played with 5 pegs and 8 colours. The game is very flexible, in that it can be played with any combination of pegs and colours. However, applications of Mastermind go further than this, existing in various fields such as teaching, artificial intelligence, and security.

The two players in the game are the code-maker or encoder, and the code-breaker or decoder. A general game begins with the code-maker setting a code that the code-breaker cannot see. This code will be a sequence of N colours, whose length will equal the number of pegs used in the game, chosen out of a set of K colours, with repetitions in the sequence allowed.e.g If N = K = 6, then the code is 6 in length, and each colour can be one of 6 colours. In this project, the colours will be represented by the numbers 0,1,2,..,K-1. The code-breaker will try to guess what the secret code through a number of guesses. The number of guesses required to guess the codes value, and the methods used, is the focus of analysis in this study. After each guess, the code-maker will return two numbers in response; X, the number of black pegs and Y, the number of white pegs. X represents the number of digits in the guess which exactly match the codes digits both in value (colour) and in position, while Y represents the components of the guess which are the right colour but in the incorrect position. The game will continue in this manner until the guess exactly matches the secret code, at which point it will end.

1.2. ABSTRACT-

For this project, we have a Mastermind Solver using C++ language. It lets us to input the length, l and a number, n to generate a sequence using the set of

numbers from 0 to (n-1). It then creates an attempt and checks it with sequence until the attempt is equal to the sequence in least average number of guesses. One could investigate the number of guesses required to solve the game, the number of evaluations needed to reach a solution and therefore the speed of the algorithm, and how these statistics change when the game is scaled up. This project will focus specifically on the number of guesses taken to solve the game and the speed and efficiency of the program.

1.3. FUNCTIONS USED-

a. void set_random_seed();
b. int randn(int n);
c. int compare(std::vector<int> attempt, std::vector<int> table, int num);
d. void init(int i_length, int i_num)
e. void give_feedback(const std::vector<int>& attempt, int& black_hits, int& white_hits)
f. void create_attempt(std::vector<int>& attempt)
g. void learn(std::vector<int>& attempt, int black_hits, int white_hits)
h. int main()

2. DEVELOPMENT OF STRUCTURES AND FUNCTIONS

2.1. struct mm_code_maker

It is structure that contains all the functions and variables required to generate a sequence and check it with the attempted sequence to give the number of white and black hits.

It contains the following variables and vectors-

1. **int length**- the size of the code maker's sequence
2. **int num**- number of possible values
3. **std::vector<int> sequence**- vector used to store the code maker's sequence

It contains the following functions-

1. **void init(int i_length, int i_num)**- Used to initialize length and num with the user-entered values when it is called in the int main() using the statement, maker.init(length, num);

2. **void generate_sequence()-** Used to generate a random sequence for the code maker. A for loop is executed from i=0 to i=length. It calls the **int randn(int n)** function which inputs random numbers from 0-(n-1) in the sequence vector until the end of the for loop.

3. **void give_feedback(const std::vector<int>& attempt, int& black_hits, int& white_hits)-**
   This function compares the attempt vector to the sequence and gives us the number of white and black hits. We execute a for loop that goes from **i=0** to **i< attempt.size()**. Within the loop, it checks the number at each position in attempt that in sequence. If they are equal the **black_hits** gets incremented by 1. Therefore, this gives us the number of black hits. If the numbers are not equal, they get stored separately in vectors **store** and **store1**.
   In the next loop, it checks whether a number in **store** has its duplicate in **store1**. If it finds a duplicate**, white_hits** gets incremented by 1 and the number gets replaced by -2 and -1 in store and store1 respectively. This has to be done so that the number does not check its duplicate again, else we will get incorrect number of white hits.
   As a result, we get the number of black hits and white hits, which gets passed by value when the give_feedback function is called in the int main() function.

   Apart from this method, we have an alternate method where we can calculate the total number of hits and number of black hits. We then subtract the number of black hits from the the total number of hits to get the number of white hits. Here is the code for the following method-

```
void give_feedback(const std::vector<int>& attempt, int& black_hits, int& white_hits)
{
  black_hits=0;
  int min=0;

  for (int i=0; i < attempt.size(); i++) {
       if (attempt[i] == sequence[i]){
         black_hits = black_hits + 1;
       }
     }

     for (int number = 0; number <= num; number++) {

       int attempt_counter = 0;
       int sequence_counter = 0;
       for (int i=0; i < attempt.size(); i++) {
         if (attempt[i] == number){
           attempt_counter = attempt_counter + 1;

         }
         if (sequence[i] == number){
           sequence_counter = sequence_counter + 1;

         }
       }
       if (attempt_counter <= sequence_counter) {
         min = min + attempt_counter;
       }
       else if (attempt_counter > sequence_counter) {
         min = min + sequence_counter;
       }

     }
     white_hits = min - black_hits;
}
```

By applying both methods to my code, I found my program to be more efficient using the former method

2.2. struct mm_code_solver

It is structure that contains all the functions and variables required to create an attempt sequence, form a permutation table and update it until we get the solution in as least number of guesses as possible.

It contains the following variables and vectors-

1. **int length**- the size of the code maker's sequence
2. **int num**- number of possible values

3. **std::vector<int> attempt**- vector used to store the code solver's sequence

It contains the following functions-

1. **void init(int i_length, int i_num)-** Used to initialize length and num with the user-entered values when it is called in the int main() using the statement, solver.init(length, num);.

   It also creates a permutation table having all the numbers possible with the user-entered length and num. We convert each number from 0 to std::pow(num, length) to the base num and push it into the **perm_table** vector.
   This is done the following way-

```
int f=s%num;
permutation.push_back(f);
s=s/num;
t--;
```

   We are pushing the mod of the number with **num**(f=s%num) into **permutation** and each time we are updating the number by dividing it with **num**(s=s/num). We run the following procedure in **a while loop** so that we can have the number of the size length. For eg. If we have 12 as the converted number, it will push in 1200 into **permutation.** We run a for loop in the reverse order to push the elements into **perm_table** vector. Then finally we clear the permutation vector for new elements to be inserted into it. This process continues until the for loop terminates. This helps us to generate the required permutation table.

2. **void create_attempt(std::vector<int>& attempt)-**
   This function creates an attempt sequence by randomly selecting a number from the permutation table. We find out the starting address(index, in the function) of any random number in the permutation table by using **randn(int n)** function. Then we use a for loop to push the number into the **attempt** vector.

```
void create_attempt(std::vector<int>& attempt){

        for(int i = 0; i < length; i++){

            attempt.push_back(randn(num));

        }

    }


    /// this version just doesn't do anything when it's called

    void learn(std::vector<int>& attempt, int black_hits, int white_hits){



    }



    int length;

    int num;


};                                      mm.cpp
```

The above is the code that was provided in **mm.cpp** in the context and requirements sheet. But we had to modify our function because this method is highly inefficient. We get our solution after hundreds and thousands of guesses. So to optimize our code, we use the former method which gives us the solution at the least average number of guesses.

3. **void learn(std::vector<int>& attempt, int black_hits, int white_hits)-** This function is used in to update the permutation table by reducing the number of possibilities to get to the solution. Therefore, it helps us to reach a solution in minimum number of guesses.

Basically, we extract every number from the permutation table at a time and push it into the vector, **table**. We compare it with the attempt sequence to find the number of black and white hits by calling the **compare** function (which is explained later). It returns the value **(black*10 + white)** for the number in table vector. Now we the value of **(black*10 + white)** for both the numbers, and we push the extracted number into another vector, ie, **correct** vector, if they are equal and the attempt vector is not equal to table vector. This procedure is repeated for all the numbers in the permutation table. Eventually, we copy the **correct** vector into **perm_table** vector. This helps to reduce the number of possibilities and we reach the solution in lesser number of guesses.

2.3. **int randn(int n)-** This function returns a random number from 0 to (n-1)

2.4. **int compare(std::vector<int> attempt, std::vector<int> table)**
This function compares the two vectors passed by value and finds out the number of white hits and black hits. It then returns the value of **(black*10 + white)**

2.5. **int main()-** It calls all the functions and also calculates the average time and average number of attempts, maximum and minimum value for the number of attempts.

3. RESULTS AND EVALUATION

LENGTH, NUM	AVERAGE NUMBER OF GUESSES				AVERAGE TIME/GAME(msec)			
	TRIAL 1	TRIAL 2	TRIAL 3	MEAN	TRIAL 1	TRIAL 2	TRIAL 3	MEAN
2,4	2.958	2.916	2.939	2.937	2.169	2.996	3.215	2.793
3,5	3.811	3.832	3.773	3.828	2.888	4.577	5.302	4.249
4,6	4.643	4.639	4.575	4.619	6.087	6.019	6.224	6.110

Table (1)

LENGTH, NUM	AVERAGE NUMBER OF GUESSES				AVERAGE TIME/GAME(msec)			
	TRIAL 1	TRIAL 2	TRIAL 3	MEAN	TRIAL 1	TRIAL 2	TRIAL 3	MEAN
5,7	5.413	5.479	5.496	5.462	63.263	64.11	66.272	64.548
5,8	5.835	5.880	5.874	5.863	120.377	130.033	134.957	128.455
6,8	6.330	6.290	6.320	6.313	1060.4	1278.94	1078.27	1139.20

Table (2)

LENGTH, NUM	AVERAGE NUMBER OF GUESSES				AVERAGE TIME/GAME(msec)			
	TRIAL 1	TRIAL 2	TRIAL 3	MEAN	TRIAL 1	TRIAL 2	TRIAL 3	MEAN
6,6	5.62	5.49	5.40	5.50	215.64	202.20	213.60	210.48
7,7	6.51	6.43	6.48	6.47	3475.2	3492.69	3464.56	3477.48

Table (3)

We calculate the average number of attempts and the average time taken to calculate the solution for 3 trials and then calculate the mean value for them. The results are stated in the tables above. All the data in table(1) and table(2) is calculated for 1000 trials and table(3) for 100 trials. My code is not optimized enough to give the results for (6,6) and (7,7) quickly for 1000 game number, therefore, I had to change the number of games to 100.

4. OPTIMIZATION

When we used **mm.cpp** code in the context and requirements sheet, we got our solution in huge large number of attempts and realized we have to make the program efficient by bringing down the average number of guesses along with maintaining the speed of the program. Therefore, we updated the **give_feedback**, **create_attempt**, **init** and **learn** functions to give us the required output. As mentioned above, we had two methods for **give_feedback**

function, and we used the most efficient one. Instead of randomly generating attempts, we made a permutation list of possible solutions in the **init** function and created guesses randomly from the list using the **create_attempt** function. And finally, in the **learn** function, we kept on checking the number of black and white hits for attempt sequences and permutation list elements, and remove the elements that did not have the same number of white and black hits as the attempt sequence. This way we got an updated permutation list with a reduced number of possible solutions, hence reducing the number of guesses. Eventhough we are able to optimize the code to a large extent, there is still scope to further increase its efficiency by making changes in the functions or by using different algorithms. But due to lack in time and knowledge at present, it is hard to optimize the code any further.

5. CONCLUSION-

We learned to code one of the classical board games, Mastermind, using C++. Since there were many different paths to take with the work, apart from development of functions, a good understanding of the context was also required to make an efficient program. Papers and materials, not only the ones provided to us by college had to be understood thoroughly, but further research was also required to which solve Mastermind. The code did not only require the implementation of a basic logic, but also urged us to use various algorithms and complex computation to give us the best possible output. Consequently, a timeless, computational problem with great history, Mastermind represents a very interesting optimization and satisfactory problem.