Effective Mastermind Solver Algorithms

by Baptiste Provendier

1. Introduction and Overview

1.1. The game of mastermind

Mastermind is a code breaking game played by two players, one is the code maker and the other one is the code solver. The modern version of the game played with pegs of color was invented in 1970. The code maker chooses a code of 4 colors, each of them having 6 possibilities. The number of possibilities is defined as the number of possibilities to the power of the length of the code, in that case 6^4 which is 1296 possibilities. The code solver tries to break that code by submitting codes and getting feedback. After each code submitted by the code solver, the code maker gives back a series of black pegs, indicating one of guess peg appears in the same position as one of the code pegs, and white pegs which means it matches the color but not the position of the code peg. The aim of the code solver is to find the code in the least number of attempts. Several papers and algorithms have been published since to find the most effective way to solve the game of mastermind. One of the first and most famous one is from Donald Knuth, which reveals a method to solve any traditional game of mastermind in 5 attempts or less.

1.2. Overview of our algorithm

In our case, we will not only consider the traditional game of mastermind but one with varying parameters for the length of the code and the numerical values each peg can be assigned up to 15 for both of them. The effectiveness of the program will be measured in terms of attempts it needs to solve a code under a 10 seconds mark. The algorithm used in our program is made of two different algorithms. The first one, made for small numbers is pretty similar to the method described in Knuth's paper. The second one, made for bigger values, is an optimized version of the "Brute force" method. This report will explore the functions and algorithms used while investigating all the improvements and optimizations made over time supported by worked examples and comparisons.

2. Algorithms and functions

2.1 Feedback

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

black_hits = 0;
white_hits = 0;
std::vector<int> occ_att;
std::vector<int> occ_seq;

for(int i = 0; i < num; i++){
  occ_att.push_back(0);
  occ_seq.push_back(0);
}

for(int j = 0; j < length; j++){
    occ_att[attempt[j]] ++;
    occ_seq[sequence[j]] ++;

  if(attempt[j] == sequence[j]){
    black_hits++;
  }
}

int hits = 0;

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

  if(occ_att[k]> occ_seq[k]){
    hits += occ_seq[k];
  }
  else{
    hits += occ_att[k];
  }
}
white_hits = hits - black_hits;
}
```

```
[dyn3129-11:C++ baptisteprovendier$ g++ mm.cpp -o mm
[dyn3129-11:C++ baptisteprovendier$ ./mm
enter length of sequence and number of possible values:
4 6
attempt:
1 1 2 2
black pegs: 0  white pegs: 0
attempt:
0 0 0 0
black pegs: 2  white pegs: 0
attempt:
3 3 0 0
black pegs: 4  white pegs: 0
the solver has found the sequence in 3 attempts
the sequence generated by the code maker was:
3 3 0 0
Time to run: 0.007183 seconds
```

The first function is the feedback one. It will take as an input the attempted code by the solver and return the number of black pegs (or black_hits) and white pegs (or white_hits) compared to the secret code previously made. Two vectors of size equals to the number of possibilities for each digit (num) are filled with zeroes. These vectors are going to act as "counters" for the occurrence of each digit in both the attempt and the code (sequence). For example, if there are 3 ones in the code, the value of the first object in the occurrence vector will be equal to 3. If at some point, the value at the same position is the same for the attempt and the code we increase the number of black pegs. To calculate the total number of hits, both black and white, we use the method described in Knuth's paper. He says that you select the minimum occurrence of each number between the attempt and the code and sum them all together. To get the number of white hits we only need to sub subtract the number of total hits by the number of black hits.

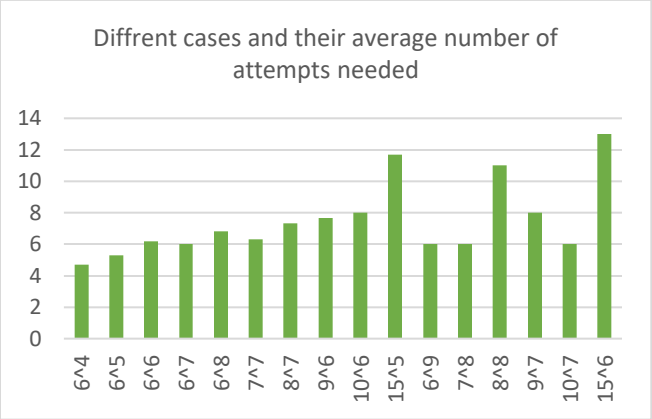
2.2 First Algorithm: Knuth's minmax

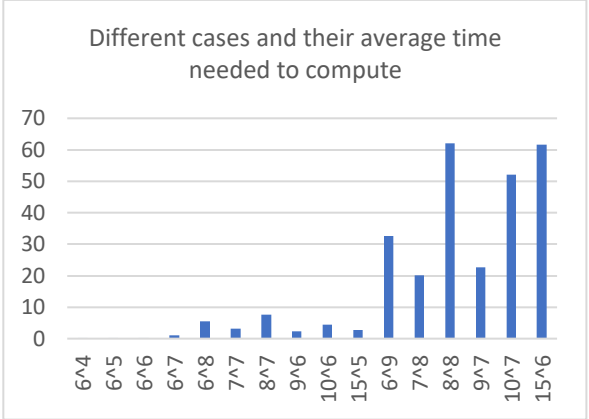
The first algorithm is used for small numbers, which was defined for a maximum of 8^7 possibilities (around 2.1 million) and that number will be explained later on. This algorithm is a slightly simplified version of the minimax method explained in Knuth's paper. The first attempt is a code composed of a first half full of 1s and the second half full of 2s. For a code of 4, Knuth describes this code to be the most efficient. After we receive the feedback the algorithm will test every possible number against the attempt (receiving black and white hits) and will store only the numbers that would have given the same result in terms of black and white pegs if the attempt was the secret code. Hence, the learn is using the same feedback functions to test the possibilities (stored into the vector of that name) and test them against

the attempt. If they usefully obtain the same score, they are kept into the choices vector. While in the minimax, the next attempt would be chosen in terms of the maximum number

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

int max_num = pow(8, 7) +1;
if(total_num < max_num) {
    if(first_learn == true) {
        choices.clear();
        // creates set of every possible outcome
    for(int i =0; i < total_num; i++) {
        possibilities.clear();
        int power = i;

        // push back every number in the set
        while(power!=0) {
            possibilities.push_back(power%num);
            power /= num;
        }
        // add leading zeroes
        int zeroes = length - possibilities.size();
        while (zeroes != 0) {
            possibilities.push_back(0);
            zeroes --;
        }
        int test_black = 0;
        int test_white = 0;
        std::vector<int> freq_poss;
        std::vector<int> freq_att;

        for(int i = 0; i < num; i++){
        freq_att.push_back(0);
        freq_poss.push_back(0);
        }

        for(int j = 0; j < length; j++){
            freq_poss[possibilities[j]] ++;
            freq_att[attempt[j]] ++;

        if(possibilities[j] == attempt[j]){
            test_black++;
        }
    }
    int test_hits = 0;
    for(int k = 0; k < num; k++) {

    if(freq_poss[k]> freq_att[k]){
        test_hits += freq_att[k];
    }
    else{
        test_hits += freq_poss[k];
    }
    }
    test_white = test_hits - test_black;

    if(test_black == black_hits && test_white == white_hits) {
            for(int m=0; m < length; m++) {
                choices.push_back(possibilities[m]);
            }
    }
}
first_learn = false;
}
```

of guesses it might eliminate from choices for every possible output of black and white pegs, and the minimum numerical value, our algorithm only selects the lowest numerical value in choices. This means in a code of size 4 and 6 nums the maximum number of attempts is not limited to 4 but it limits the number of calculations which is going to help for slightly higher number. To this original program, a timer function was added in the main function to calculate the time need to compute the program. We obtain these results.





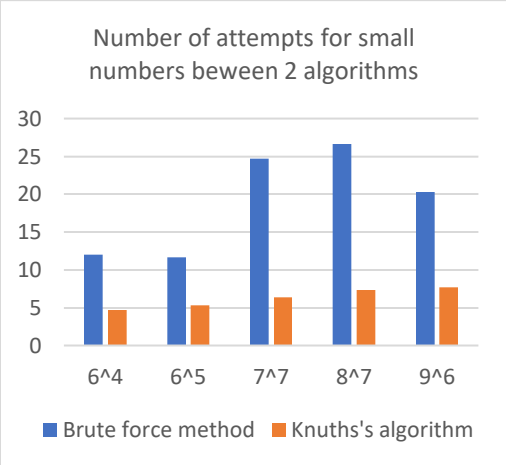
All the results were made from average of 25 consecutive attempts. (The first number being num and the second one the size of the code length). The effectiveness of this program is undeniable as it can run codes with up to 17 million possibilities in under 14 attempts every time. However, the major issue will be the exponentially rising time needed to compute the program for large numbers. Small numbers are then run instantaneously in few attempts, but bigger numbers can take minutes and sometimes hours to compute and this rise of time needed is exponential. This is where the limit of 8^7 possibilities, or 2.1 million, comes in. From

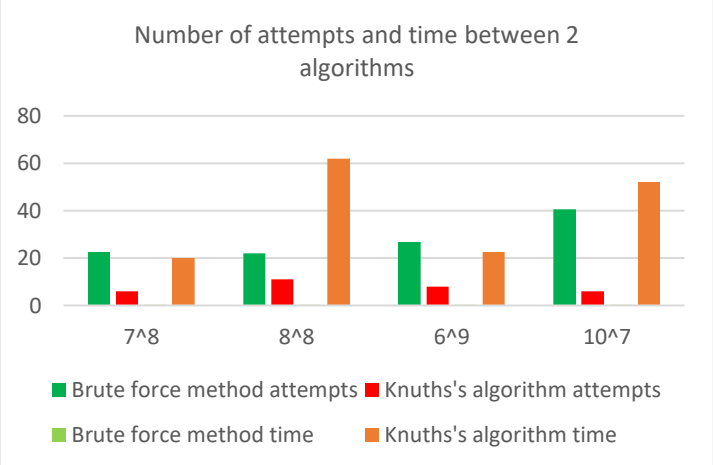
the tests ran, it is the maximum value of possibilities this code can compile in less than 10 seconds every time. Hence, we will use this code for all the combinations that have less than 2.1 million possibilities, but we need to find another way of solving codes with more possibilities than that.

2.3 "Brute force" method

```
else{
    if(first_learn == true) {
        blackh = black_hits;
        index = 0;
        first_learn = false;

        choices.clear();
        for(int j = 0; j < length; j++) {
            choices.push_back(0);
        }
    }

    if(black_hits > blackh) {
        index ++;
        blackh = black_hits;
    }

    if(black_hits < blackh) {
        choices[(length-1) - index] --;
        index ++;
    }
    choices[(length-1) - index]++;
}
```

The "brute force" method is a well-known method for hacking: it consists of entering every single possible value until we get the right one. However, in mastermind, each attempt received some feedback from the actual code so no need to try every value. We are going to see two versions of this method and the improvement between the two. The first one consists of sending a code full of 0 and then increase each digit one by one until the number of black hits increases by number. If it is the case, then it goes to the next digit. If it decreases that means 0 was a black peg and so the digit goes back to 0 and we move on. This method is able to compute all the combinaisons in a fraction of time but is pretty unconsistent. Depending on if the digit values are high or low the number of attempts can vary a lot. Another important change to note is the change of class for total_num. When using the first algorithm, the total number of possibilities was defined as an integer while now it has been changed to a double. The reason for this change is that an intenger can contain 32 bits, and hence the maximum value it can store is expressed in expressed in billions. This is fine for the small values we were using (maximum of 2.1 million) but now that we are manipulating much bigger numbers (maximum for a 15-15 is approximately $4*10^{17}$) we need more bits to store these values. The class "double" can contain 64 bits which is now capable of handling such calculations.



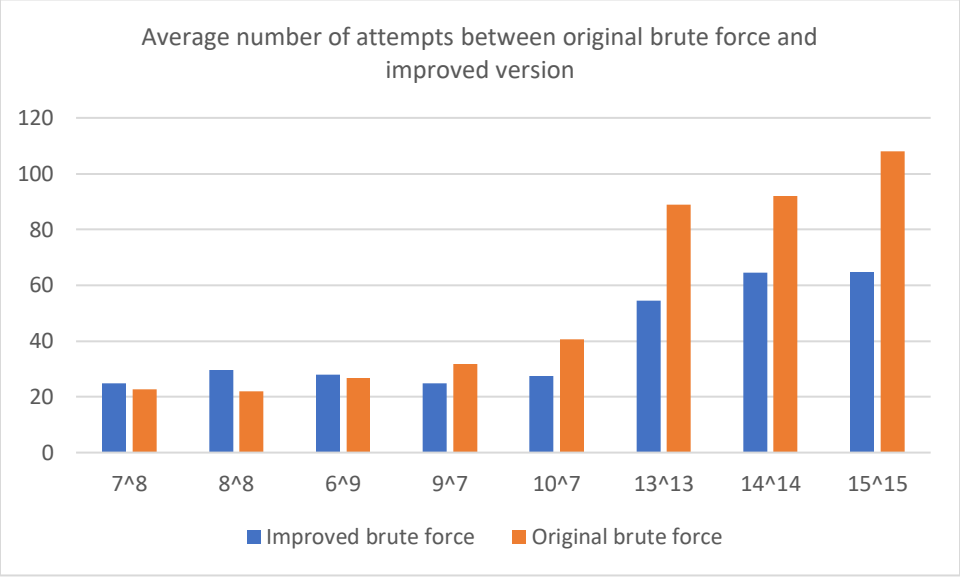


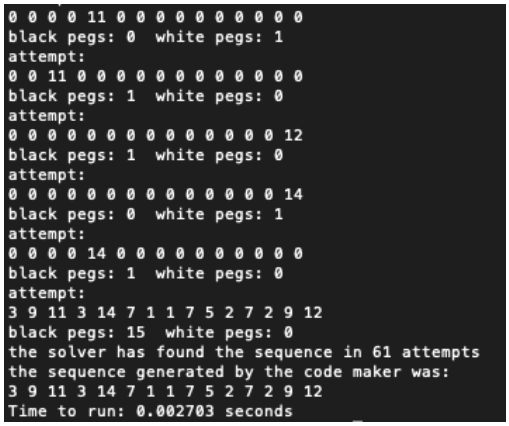
From these comparisons from the first algorithm, we can clearly see that in the case of small combinations, the Knuth's algorithm is much more effective than the brute force. For higher combinations it is still the case but the time necessary to run it is incredibly faster. We cannot ever see the column for the time of the brute force method as it is extremely small compared

to the time for the Knuth's algorithm. Hence, we have an algorithm that is useless for small combination, but we are able to run every code under one tenth of a second. Now that we have focused our algorithm in getting under the ten seconds mark, the aim is to minimize the number of attempts.

The first thought to improve our brute force method could be to use the white hits as well as the black hits to solve it quicker. However, that would be tedious and not make that much difference. Instead, the idea is to test codes full of the same numbers, so we know all the numbers in the actual code but not their order. It is, then, only a matter a rearranging the code to find the actual solution. The actual code is pretty messy but here is the basic idea. First, we test vectors full of the same numbers for all the numbers and store the value of black pegs for a vector full of 0. We store into a vector each digit as many times as they have black pegs. We have a vector with all the numbers of the actual code in it but not in the right order. For each number going from 1 to (num-1), then occurrence is counted, and the number is tested on each digit until we find all the positions where it makes the number of black pegs increase. Every time a value is found it is stored in a vector and when all the solutions of a number are found we do the same thing with the next number. This vector is also useful because we want to test only on indexes that are empty in this vector, and if it is not the case, we increase the index until we find an empty spot. Finally, some functions are called multiple times in the algorithm and this is because, otherwise, when a number was found it had to wait for the next loop to get to the next number and hence was sending a useless attempt to the feedback. The same think was happening when the whole code was found, and this is why the next_num function and the check if we have the complete code are called twice.

The advantages of this method compared to the traditional brute force is that all the secret numbers are known and only their positions are to be found. Also, if a number is not present in the code it is never sent in an attempt. This improvement can be seen on the experimental values of the number of attempts between the two brute force method. (Time is not taken into consideration as all the combinations are of the order of a milliseconds.)





The difference is not very big for smaller combinations and it may even seem that the original brute force is more effective but as the combinations increase there is a big gap between the original version and the improved one. The 15 by 15 codes can now be solved in 65 attempts in average compared to 110 attempts with the original brute force.

3. Conclusion & Improvements

In conclusion, the algorithm to solve the mastermind is divided in two parts. The first one used for small combinations of numbers, the tests have shown that this is when the number of total possibilities is under 2.1 million, is a simpler version of the minimax method used by Knuth. The second algorithm is an optimized version of the "brute force" method as it tries to get all the numbers of the code, so its role is only to find the order. The whole program works great and is able to get a result in under 10 seconds all the time (and usually under 10 attempts in the case of the first algorithm). However, there is a better and more effective method to solve a mastermind and this is by using genetic algorithm. Many papers and studies have been made on that since the modern computer era, each of them making improvement compared to the Knuth method. This method would consist of using a population of individual solutions and select random numbers to be the parents of that generation. These parents would then create the children of the next generation and over generations, the population "evolves" toward an optimal solution making a more effective and more optimized algorithm.