Mastermind Report

Issa Bqain

March 22, 2019

1 Contents

1.1 Feedback function

1.1.1 Black hits

1.1.2 White hits

1.2 Algorithm 1: Knuth algorithm explained

1.2.1 Generating the list

1.2.2 Updating the list

1.3 Inspiration for Algorithm 2

1.4 Algorithm 2 Explained

1.4.1 Algorithm 2 code review

1.4.2 Stage 1: Create attempt

1.4.3 Stage 1: Learn

1.4.4 Stage 2: Create attempt

1.4.5 Stage 2: Learn

1.5 Evaluation

2 Feedback Function

The aim of this function is to find the black hits and white hits when comparing the attempt to the secret code which is defined as sequence.

In order to achieve this, the function finds the total number of black hits and the total number of hits. The white hits can then be simply obtained by finding the difference between the white hits and the black hits.

Black hits

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

    black_hits = 0;
    white_hits = 0;
    int total = 0;

    //finds BH
    for(int x = 0; x < length ; x++){
        if( attempt[x] == sequence[x]){
            black_hits++;
        }
    }
```

In order to find the black hits, the value of each number in a specific index starting from the first number in the attempt to the last number is individually compared to the number in the same index in the secret code which is stored in the sequence vector. If they are equal, black hits which is initialised as 0 would increase by 1.

```
For example:

code : 1221
attempt : 1100

Black hits = 0
is 1 == 1 ?
Black hits = 1
is 1 == 2 ?
Black hits = 1
is 0 == 2?
Black hits = 1
is 0 == 1
Black hits = 1
```

White hits

```
    //finds total hits and white hits
    for(int i = 0; i< num; i++){

        int x = 0;
        int y = 0;

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

            if(attempt[j] == i){
                x++;
            }

            if(sequence[j] == i){
                y++;
            }

        }

        if(x >= y){
            total = y+ total;
        }
        else{
            total = x + total;
        }


    }

    white_hits = total - black_hits;
}
```

In order to find the total hits I used the following equation:

$$\sum_{n=0}^{num-1} min(x_n, y_n) \tag{1}$$

- x_n is the number of times that the digit n appears in the attempt
- y_n is the number of times that the digit n appears in the secret code
- Num is the increments of digits the code has for example 1211 has 3 digits , 1111 has 2 digit
- n is the actual digit being tested.

Once the total number of hits has been obtained by subtracting the value of the black hits it would be able to find the value of the whit hits and hence obtain the full feedback of an attempt.

3 Algorithm 1: Knuth algorithm explained

A brief description of how I have implemented my version of this algorithm follows the following steps :

1. A list is made for all possible solutions given Length and Num.
2. An attempt is made which is chosen specifically for certain lengths and randomised from the list for other lengths.
3. Any number in the list which does not have the same feedback when compared to the attempt relative to when the attempt was compared to the actual sequence is removed from the list.
4. A random number is picked from the updated list
5. Steps 3 and 4 are repeated until blackhits = length

3.1 Generating the list

In order to create a list I have used base conversions to create all possible numbers given a value for length and num. This is done in the function generate list and completes step 1.

When the code has a value of num = 5 then the base we will convert to will be base 5. num = base value.

To convert bases this can be done by repeatedly dividing the quotient by the num value until it equals 0 and then combining the numbers in in reverse.

```
Base conversion example:

If the number 17 was to be converted into base 5

17/3 = 5R2
5/3= 1R2
2/3 = 0R1

The value in the new base = All the remainders in reverse order
In this case Remainders = 221
In reverse order = 122
```

When implementing this, I broke it down into two steps. The first step completes the base conversion which was described above.

```
for (int i=0;i< possibilities; i++) {
    std::vector<int> tmp;
    std::vector<int> tmp2;


    int quotient = i;
    while (quotient!=0) {
1       tmp.push_back(quotient % num);
        quotient = quotient / num;
    }


    int zerosneeded = length - tmp.size();
2


    while (zerosneeded != 0) {
        tmp.push_back(0);
        zerosneeded--;
    }

    for( int j = tmp.size()-1 ; j >=0; j--){
        tmp2.push_back(tmp[j]);
    }

    list.push_back(tmp2);
}
```

The second step calculates if needed, the number of zeros it needs to add to the number and adds them into the temporary vector tmp to produce a possible attempt.

Since the values of the remainders have to be in reverse order, another for loop as added to reverse the values in this attempt and stores it in another temporary vector called tmp2.

Finally this temporary vector tmp2 is stored into a vector of vectors called list for later use in stage 3 of the algorithm.

The number of possibilities for a given value of length and num is num to the power of length. Hence to obtain all the possible codes and store them in the list, this process is set to repeat in a for loop until it is equal to possibilities which is assigned the value of num to the power of length.

3.2 Updating the list

In order to reduce the size of the list, any number in the list which does not have the same feedback when compared to the attempt relative to when the attempt was compared to the actual sequence is removed from the list.

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

  for(int i = 0; i< list.size(); i++){

      std::vector<int> tmp = list[i];
      int black = 0;
      int white = 0;

      get_black_white(attempt, tmp, black, white);

      if( ( list[i] != attempt) && (black_hits == black) && (white_hits == white) ){
          newlist.push_back(list[i]);
      }
  }

  list = newlist;
  newlist.clear();

}
```

In order to achieve this another function called learn algorithm one incrementally takes all the attempts in the list and uses the function get black whit to get the feedback for the attempt from the list compared to the attempt the algorithm just tried. For each one these values are compared to the black and white hits of the attempt relative to the sequence and if they are not equal they are not stored in the list. Once it has gone through all the numbers these new values are copied back into the list and the temporary new list vector of vectors is cleared for later use.

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

  for(int i = 0; i< list.size(); i++){

      std::vector<int> tmp = list[i];
      int black = 0;
      int white = 0;

      get_black_white(attempt, tmp, black, white);

      if( ( list[i] != attempt) && (black_hits == black) && (white_hits == white) ){
          newlist.push_back(list[i]);
      }
  }

  list = newlist;
  newlist.clear();
  std::cout << list.size() << std::endl;

}
```

4 Inspiration for Alogrithm 2

One inefficient way to make the solver mastermind is by simply increasing the value in each index until getting all the black hits needed. Here is an example for length = 7 and num = 5.

```
Example :

    1230304  (B)

1)  0000000  (2)
2)  1000000  (3)
3)  1100000  (3)
4)  1200000  (4)
5)  1210000  (4)
6)  1220000  (4)
7)  1230000  (5)
8)  1231000  (4)
9)  1230100  (5)
10) 1230200  (5)
11) 1230300  (6)
12) 1230310  (5)
13) 1230301  (6)
14) 1230302  (6)
15) 1230303  (6)
16) 1230304  (7)
```

- **If Black hits increase:** Increase index value and reset number to 1
- **If black hits decrease:** Go back to the old vector. Increase index and rest number to 1.
- **If Black hits decrease:** Increase the number by 1

Based on this method, the average number of attempts for a length = 15 and num = 15 codeword is 107. For large values this method is inefficient and hence another algorithm is devised. For codes with a high value for length and num , this can heavily be improved by simply storing the number of times each number appears in the code (black hits) and use this information to refine the algorithm such that when changing the number in an index only values which could exist would be used inctead of going through all possible numbers in for each index.

An example of this process can be seen below. For large values of length and num this has a large improvement in the average attempts taken. For example for a 15x15 this would decrease the average from 107 to 61 when tested over 10,000 attempts. Hence this would give us our algorithm for large values, **Algorithm 2**.

```
Example :

    1130223  (B)

1)  0000000    (1)
2)  1111111    (2)
3)  2222222    (2)
4)  3333333    (2)
5)  4444444    (0)

\\since 4 returned no black hits this means we can reduce the number of attempts by not using this in out increments

6)  0000000    (1)
7)  1000000    (2)
8)  1100000    (3)
9)  1120000    (3)
10) 1130000    (4)
11) 1132000    (3)
12) 1130200    (5)
13) 1130220    (6)
14) 1130223    (7)
```

Figure 1: Algorithm 2 example

5 Algorithm 2 Explained

This algorithm is very simple and has two main stages:

Stage 1) The number of times each number is repeated in the sequence is stored.

Stage 2) Starting with the first index the value is increased to the first available number which appears in the list atleast once.

Stage 2.1)

- **If Black hits increase:** Increase index value, decrease number of black hits left for that number by 1. Reset number to next available number which does not have 0 black hits left.
- **If black hits decrease:** Go back to the old vector. Increase index and rest number to 1.
- **If Black hits remain the same:** Increase the number to the next number which does not have 0 black hits left.

5.1 Algorithm 2 Code overview

In order to achieve the algorithm explained in the previous example I have broken it up into two stages and implemented the following.

Stage 1: Create attempt

The aim of this is to be able to find how often each number appears in the code, hence this can simply be done by filling the attempt with same number and storing how much black hits this achieves for all numbers as shown in the example in figure.

```
if(first_stage){

    attempt.resize(length);

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

        attempt[i] = finding;
    }

    finding++;

}
```

Figure 2: Stage 1 of create attempt algorithm two function

Finding : This is the number which we are using to find the black hits for to see how often it appears in the sequence. This is initialised as zero.

Initially we resize the vector attempt such that it would be able to access the elements in the vector. After this we fill the vector with same value which is stored in finding. This allows us to create an attempt which we can get feedback for.

Stage 1: Learn

Having created the attempt, the feedback will be stored and this information will be used later on in stage 2.

```
if(first_stage){
    bh_each_number.push_back(black_hits);

    if(bh_each_number.size() == num){
        first_stage = false;
        second_stage = true;
    }
}
```

Figure 3: Stage 1 of the learn algorithm two function

bh each number: This vector is used to store how many times each number appears in the code.

Having created the attempt and received feedback for it, the black hits would be stored here for later use. For example:

```
code : 1289
Attempt: 0000

Black_hits = 0;

\\this value is then stored in the vector bh each number.
\\If this was the first attempt this vector would now have one values{ 0 }

attempt: 1111

back_hits = 1

\\ it would now {0,1}

and so on.
```

Figure 4: Exampe of how stage of of learn algorithm two function works

Once we have gone through all the numbers we need and obtained the information for the next stage. The boolen first stage becomes false and the boolean second stage becomes true hence it would no longer enter these loops for stage 1 in the create attempt and learn functions. Now the algorithm would begin stage 2 where it will find what the sequence is.

Stage 2: Create attempt

This simply copies the attempt created in the learn function from the vector new attempt into the vector attempt to obtain the feedback.

```
if(second_stage){
    attempt = new_attempt;
}
```

Figure 5: Stage 2 create attempt algorithm 2 function

Stage 2: learn

```
if(second_attempt){

    num_digit = 1;

    while(bh_each_number[num_digit] == 0){
        num_digit++;
    }

    new_attempt[0] = num_digit;

    second_attempt = false;
    fall_in_bh = false;
}

if(first_attempt){
    new_attempt.resize(length);

    for(int i = 0; i<length; i++){
        new_attempt[i] = 0;
    }

    first_attempt = false;
    second_attempt = true;

    fall_in_bh = false;
}
```

Figure 6: Stage 2 learn algorithm two function

Having just made stage 2 true, this if condition is met and the code is immediately going through the next set of if conditions. Here first attempt is initialised as true and second attempt is initialised as false. In the first attempt it would not enter the first for loop shown above and then enter the second for loop as shown here. It would create an initial attempt of all zeros and make first attempt false and second attempt true to be able to enter the first if loop and make the second attempt. On the second attempt it would then change the value of the first index to the next smallest number which appears in the code atleast once.

This leads to an increase in average attempts by one but for correctness and ease of readability it has been kept as such.

The reason as to why the second attempt is above the first attempt is because if the first attempt condition was above, after second attempt becomes true, it would enter the loop immediately after and hence the attempt would immediately be changed and the algorithm would not work as intended.

```
if( !first_attempt && !second_attempt){
    analyse = true;
}
```

Figure 7: Stage 2 learn algorithm two function

At the end of the second attempt, the boolean analyse becomes true and this allows us to enter the process by which we can begin to use all the information we have stored until now in stage 2.1 which occurs when first attempt second attempt are false and analyse is true.

```
if( !first_attempt && !second_attempt && analyse){
```

Figure 8: Stage 2.1 learn algorithm two function

Stage 2.1: learn

In this stage we will be creating a new attempt based on the changed in the value of black hits. There are 3 different scenarios. Black hits increasing, black hits decreasing and black hits staying the same as highlighted previously.

Black hits decreasing

```
if( black_hits < old_black){
    index++;
    new_attempt = old_attempt;

    num_digit = 1;


    while(bh_each_number[num_digit] == 0){
        num_digit++;
    }

    new_attempt[index] = num_digit;

    fall_in_bh = true;
}
```

Figure 9: Attempt adjustments for decrease in black hits

When black hits decrease this means that the previous attempt had more values in the correct place than previously. Hence in order to make adjustments for this the code would restore the new attempt with the previous one. It would then increase the value of index and restore the value of num digit to 1, where it begins to find the next smallest number which appears in the code atleast once and assigns it to the index.

```
code :    0123 (BH)
          0000 (1)
          1000 (0)
          0100 (2)
```

Figure 10: Example to illustrate correction for a decrease in black hits

When there is a fall in black hits the algorithm would not want to store these black hit and attempt values for comparison later on as this information would be redundant and not useful since it lead to a decrease in black hits.

In addition to this consider this scenario: without this correction, there would be a scenario which leads to two or more black hit decrease in a row. This would mean that the loop will try to adjust for this by going back to the old attempt and incrementing the number to the next value but what happens when the previous attempt also had a decrease in black hits? it would never be able to access the vector before it which was more accurate and the algorithm would never be able to find the code as the black hits for the previous indexes where not found.

To adjust for this a new boolean called fall in bh is introduced such that the values for old attempt and old black hits (called old black) are only stored when fall in bh is false as shown below.

```
if(!fall_in_bh){
    old_black = black_hits;
    old_attempt = attempt;
}
```

Figure 11: Example to illustrate correction for a decrease in black hits

Black hits increasing

```
if(black_hits > old_black){

    bh_each_number[num_digit] = bh_each_number[num_digit] - 1;
    index++;
    new_attempt = attempt;
    num_digit = 1;
    while(bh_each_number[num_digit] == 0){
        num_digit++;
    }

    if( index < new_attempt.size() ){
        new_attempt[index] = num_digit;
    }

    fall_in_bh = false;
}
```

Figure 12: Example to illustrate correction for a decrease in black hits

When black hits increase, this means that we have found the number in that certain index. As a result, the algorithm will decrease the number of times this number appears in the code by 1 such that when looking for numbers to place in the next index we will have to go through less numbers. The algorithm will then increase the index value and find the next smallest number which appears

in the sequence atleast once and assign that value to the index.

Since there was an increase in black hits, this information is valuable and hence fall in bh becomes false such that the algorithm will be able to use this information for comparisons in creating future attempts.

```
code :    1123 (BH)
          0000 (0)
          1000 (1)
          1100 (2)
          1120 (3)
          1123 (4)
```

Figure 13: Example to illustrate correction for a decrease in black hits

Black hits remain constant

When Black hits remain constant, this means that we have not yet found the number for that index. Therefore to find the number present in that index the algorithm continuously increments the number to the next smallest number which appears in the sequence atleast once until it is found. This has the same logic as previously shown and can be seen below.

```
if(black_hits == old_black){

  num_digit++;

  while(bh_each_number[num_digit] == 0){
      num_digit++;
  }

  new_attempt[index] = num_digit;

  fall_in_bh = false;
}
```

Figure 14: Example to illustrate correction for a decrease in black hits

6 Evalauation

I have collected the following data From these algorithms

Algorithm 1 : Knuth

length/dig	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1.55	1.8	2.5	2.87	3.38	3.84	4.29	4.72	5.95	5.86	6.23	6.32	6.91	8.36
2	1	1.92	2.45	3.2	3.32	3.81	4.31	4.71	5.28	5.7	6.14	6.7	7.23	7.78	8.47
3	1	2.37	2.84	3.38	3.87	4.22	4.79	5.22	5.63	6.05	6.2	6.86	7	7.79	7.83
4	1	2.78	3.35	3.54	4.12	4.48	4.94	5.34	5.98	6.18	6.8	7.12	7.64	7.74	8.34
5	1	3.36	3.54	4.08	4.74	5.12	5.5	6	6.42	6.64	6.98	7.1	7.32	7.72	8.24
6	1	3.68	3.61	4.66	5.3	5.36	5.94	6.42	6.94	2.4	4.56	7.56			
7	1	4.21	4.19	4.98	5.64	6.08	6.4	6.82	7.2						
8	1	4.4	4.63	5.4	5.8	6.4	7.4								
9	1	5.17	5.14	6.2	6.1	7.1									
10	1	5.3	5.52	5.9	6.7										
11	1	6.01	6.05	7											
12	1	6.37	5.9	6.8											
13	1	6.4	6.8	5											
14	1	7.03	6.7												
15	1	7.42	42.97												

Figure 15: Table to show which numbers can be achieved in less than 10 seconds and their average attempts. Yellow is viable and red is not

length/dig	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2															
3															
4															
5	0	0	0	0.002	0.000638	0.01668	0.0387	0.07386	0.13616	0.23646	0.39384	0.6966	0.92958	1.37744	1.96474
6	0	0	0.00018	0.0087	0.03458	0.10606	0.2712	0.60702	1.30236	2.39592	4.33794	7.53628			
7	0	0.000085	0.00492	0.03666	0.1834	0.67184	1.95634	5.01372	11.9394						
8	0	0.0006	0.0155	0.1503	0.9181	3.9902	14.0925								
9	0	0.0018	0.0536	0.7015	5.2522	28.001									
10	0	0.0027	0.158	2.83561	27.5167										
11	0	0.0065	0.4948	11.7171											
12	0	0.0128	1.552	47.7447											
13	0	0.0249	4.5941	204											
14	0	0.0492	14.2898												
15	0	0.0988	43.1	42.975											

Figure 16: Average time taken for each value. If it states 0 this means the value was too small/negligible and hence could not be measured

length/dig	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2															
3															
4															
5	0	0	0	0.002	0.007	0.017	0.041	0.078	0.144	0.249	0.415	0.645	1.159	1.487	2.101
6	0	0	0.002	0.001	0.036	0.10606	0.279	0.60702	1.347	2.5	4.56				
7	0	0.001	0.006	0.038	0.19	0.691	2.026	5.171	12.275						
8	0	0.003	0.021	0.155	0.931	4.124	14.442								
9	0	0.005	0.061	0.71	5.33	28.627									
10	0	0.007	0.161	2.897	27.997										
11	0	0.011	0.506	11.883											
12	0	0.019	1.552	48.425											
13	0	0.037	4.639	204.8											
14	0	0.064	14.347												
15	0	0.118	43.5												

Figure 17: Maximum time taken for each value. If it states 0 this means the value was too small/negligible and hence could not be measured

Algorithm 2

length/dig	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1.46	2.08	2.38	3.2	3.28	4.14	4.56	4.92	5.9	6.26	6.3	6.94	7.5	8.08
2	1	3.14	4.28	5.66	7.62	8.54	9.26	10.26	11.92	12.28	13.76	14.42	16.24	16.04	17.48
3	1	4.64	6.3	7.66	9.06	10.52	11.36	12.82	13.4	15.24	15.96	17.3	17.98	19.02	20.12
4	1	4.96	7.82	10.22	10.78	12.14	13.16	14.46	15.48	17.3	17.5	19.06	19.92	20.9	22.26
5	1	6.66	9.4	11.4	12.74	14.34	15.12	16.36	18.66	19.02	20.42	20.62	22.74	23.16	25.04
6	1	8.22	10.52	12.64	14.26	15.76	17.78	19.72	19.68	21.56	22.46	23.96	25.56	26.26	27.38
7	1	3.96	11.2	4.98	16.1	18.14	20.2	22.3	22.02	24.22	27.08	26.6	27.02	28.6	30.82
8	1	4.24	12.43	16.16	17.92	20.4	22.5	23.82	26.08	28.04	28.36	30.36	31.66	31.89	33.08
9	1	4.5	13.79	16.76	21.06	22.34	24.64	27.28	28.48	30.04	32.08	33.26	35.5	37.34	37.64
10	1	5.32	15.62	19	23.16	25.16	27.14	30.06	32.26	32.82	36.08	36.84	39.54	39.16	40.48
11	1	5.86	17.02	21.24	24.3	27.32	29.53	33.36	35.26	38.44	37.16	40.64	42.32	44.02	44.42
12	1	6.44	18.84	22.42	27.46	30.04	34.08	34.76	39.78	39.82	42.38	45.1	46.68	48.26	49.18
13	1	6.76	19.62	24.12	29.3	32.62	35.98	39.08	42.04	44.7	43.82	48.7	49.56	52.6	53.98
14	1	6.76	20.44	26.26	30.56	35.2	39.06	42.96	43.68	46.72	51.46	52.46	54.04	57.92	57.98
15	1	7.4	22.22	28.14	32.53	36.82	41.18	46.06	48.82	52.16	51.14	57.94	58.38	60.56	61.7

Figure 18: average attempts for all values and values which knuth algorithm cannot complete

length/dig	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1.55	1.8	2.5	2.87	3.38	3.84	4.29	4.72	5.95	5.86	6.23	6.32	6.91	8.36
2	1	1.92	2.45	3.2	3.32	3.81	4.31	4.71	5.28	5.7	6.14	6.7	7.23	7.78	8.47
3	1	2.37	2.84	3.38	3.87	4.22	4.79	5.22	5.63	6.05	6.2	6.86	7	7.79	7.83
4	1	2.78	3.35	3.54	4.12	4.48	4.94	5.34	5.98	6.18	6.8	7.12	7.64	7.74	8.34
5	1	3.36	3.54	4.08	4.74	5.12	5.5	6	6.42	6.64	6.98	7.1	7.32	7.72	8.24
6	1	3.68	3.61	4.66	5.3	5.36	5.94	6.42	6.94	2.4	4.56	7.56			
7	1	4.21	4.19	4.98	5.64	6.08	6.4	6.82	7.2						
8	1	4.4	4.63	5.4	5.8	6.4	7.4								
9	1	5.17	5.14	6.2	6.1	7.1									
10	1	5.3	5.52	5.9	6.7										
11	1	6.01	6.05	7											
12	1	6.37	5.9	6.8											
13	1	6.4	6.8	5											
14	1	7.03	6.7												
15	1	7.42	42.97												

Figure 19: average attempts for all values and values which knuth algorithm cannot complete

According to these calculations this would lead us to only go up to the values for knuth when the number of possible attempts = 8 to the power of 7.