Master Mind solver

Keran Zheng
Electrical and Electronic Engineering Department
Imperial College London

Functions

This part lists some of the functions used in the implementation of the solver.

```
1.    bool check_contain(int n, const std::vector<int>&
v){//checks if an int is contained in a vector, returns true if
the vector contains n
```

```
1.    void pool_generator(int length, int num, int count,
std::vector<int>& v){//translates a decimal number into a
vector of int of length which contains the numbers of the n-
numeral system number converted from the decimal integer
```

```
1.    void give_feedback_trial(const std::vector<int>&
attempt, const std::vector<int>& trial, int num, int&
black_hits, int& white_hits){//takes input of two vector
sequence and compare them to return black_hits and white_hits
```

```
1.    int min(int n1, int n2)//out put the smaller num of
the two
```

```
1.    int number_of_num(const std::vector<int>& v, int
num){//returns the number of the input number "num" is present
in the input vector
```

Algorithm

The solver applies two different algorithms depending on the length and the number of symbols of the code. This part of the report explains the algorithms applied in the final submission.

- give_feedback

The algorithm applied for the member function $give\_feedback$ referenced the method explained in *The Computer As Master Mind* (Knuth, 1976-77). “Let n_i be the

number of times symbol *i* occurs in the codeword, and n_i' the number of times it occurs in the test pattern, for $1 \leq i \leq 6$; then the number of total number of hits, both white and black is

$$\min(n_1, n_1') + min(n_2, n_2') + \cdots + min(n_6, n_6').”$$

Based on the above formula, the algorithm first calculates the number of black hits and then obtain the number of white hits by subtracting the number of black hits from the total number of hits.

```
1.     void give_feedback(const std::vector<int>& attempt,
int& black_hits, int& white_hits){
2.
3.        int total_hits = 0;
4.        black_hits = 0;
5.
6.        std::vector<int> n_attempt;
7.        std::vector<int> n_sequence;
8.        for(int i = 0; i < num; i++){
9.          n_attempt.push_back(number_of_num(attempt, i));
10.           n_sequence.push_back(number_of_num(sequence, i));
11.         }
12.
13.         for(int i = 0; i < num; i++){
14.           total_hits = total_hits + min(n_sequence[i],
   n_attempt[i]);
15.         }
16.
17.         for(int i = 0; i < length; i++){
18.           if(attempt[i] == sequence[i]){
19.             black_hits++;
20.           }
21.         }
22.
23.         white_hits = total_hits - black_hits;
24.  }
```

Above is the implementation of the $give\_feedback$ algorithm. The function $number\_of\_num(const\ std::vector < int > \&\ v, int\ n)$ takes in a vector and an integer and returns the number of the input integer contained in the vector. As a result, the vectors $n\_attempt$ and $n\_sequence$ stores the number of the symbol, referred as its index in the $n\_attempt$ and $n\_sequence$ vectors, present in the attempt and sequence vectors. And the next for loop sums the value returned by the min function for the minimum number of each symbol present in both attempt and sequence vectors. The last for loop checks every index in the attempt and the sequence vectors

for a matching symbol and counts the number of black hits. Finally, the number of white hits is obtained by subtracting the number of black hits from the total number of hits.

- ## Solver algorithm: Knuth's method

Knuth's method for solving the master mind game with length 4 and 6 different symbols is the most popular and efficient way, in terms of the number of attempts taken, for solving code of length and number of symbols such that the total number of possible permutations is within a certain limit.

According to the paper *The Computer As Master Mind* (Knuth, 1976-77), Knuth has broken down the approach to solving the Master Mind game into the following steps. The first being to create a pool of all possible solutions. The second step is to try a first guess and obtain feedback. In Knuth's example the first attempt is always 1122. The next step is then to eliminate all code arrangements that do not give the same number of black hits and white hits as the feed back when compared with the attempt code arrangement, so that all remained arrangement must contain the correct solution. Knuth's next step is to go through another set of steps which finds out a code from the remaining pool of possible solutions that would eliminate the greatest number of incorrect solutions from the pool. However, this will not be implemented in my solver algorithm. The solver algorithm skips this step and runs the above 3 steps repeatedly until a final solution is obtained. The next part explains the code in the solver for the implementation of the 3 steps.

Defining member data:

```
1.    std::vector<std::vector<int>> pool; //this is the
pool vector which contains all possibilities of code
arrangement in vectors<int>
```

Algorithm steps:

1. Pool generation:

   The pool is a member data defined in the solver struct as, $std::vector < std::vector < int >> pool$, a vector that contains vector of integers. The pool is generated as a very first step. Therefore, its implementation is inside the $init()$ function as it is an initializing phase in solver. Below is the code for pool generation.

```
1.    for(int i = 0; i < pow(num, length); i++){
2.          std::vector<int> tmp;
3.          pool_generator(length, num, i, tmp);
```

```
4.            pool.push_back(tmp);
5.          }
```

Within the for loop, the loop generates an increasing number of i which covers all the number of possible code arrangement. For example, if the code has length 4 and 6 different symbols, the total number of possible code arrangement is $6^4 = 1296$. The loop generates the number i such that it increments from 0 to 1295. Then, within the loop the $pool\_generator(int\ length, int\ num, int\ I, std::vector < int > \&\ v)$ function takes in the integer values of length and the number of symbols of the code and the number loops the loop has gone through i, which is an integer in decimal, and converts it in to a vector of length $length$ in which each index contains a single digit of the decimal number i converted into num-numeral sysyem number. For example, again in the length = 4 num = 6 case, in the case that the loop has gone through 6 times and $i = 6$, the function converts the integer 6 in to a vector that represents its 6-numeral system equivalent number {0, 0, 1, 0}. Therefore, this loop generates vectors from {0, 0, 0, 0} all the way up to {5, 5, 5, 5}, which represents the total 1296 possibilities of code arrangement for a length = 4, num = 6 situation.

2. Create attempt:

   The solver generates attempt by randomly choosing a vector from the pool. This is achieved with the below line.

```
1.    attempt = pool[randn(pool.size())];
```

3. learn:

   In the learning phase, the solver takes in the feedback of black hits and white hits and compares all the pool content with the attempt vector to eliminate all pool content that does not have matching black hits and white hits as provided by the feedback.

```
1.    std::vector<std::vector<int>> newpool;
```

```
2.
3.          int black_new, white_new;
4.          for(int i = 0; i < pool.size(); i++){
5.            give_feedback_trial(attempt, pool[i], num, black_new,
   white_new);
6.            if((black_new == black_hits) && (white_new ==
   white_hits)){
7.              newpool.push_back(pool[i]);
```

```
8.            }
9.
10.           }
11.           pool.clear();
12.           pool = newpool;
13.         }
```

In the above implementation, within the for loop, the $give\_feedback\_trial()$ compares to vectors of code and feed back new black hits and white hits, which are stored in $black\_new$ and $white\_new$ separately. The for loop is defined such that it checks all the vectors in the pool, and the if statement checks for vectors in the pool that satisfy the conditions and push them into a new vector – $newpool$. The newpool then takes over the original pool.

- Solver algorithm: Higher Order Pool Size Index Exchange System (HOPSIES)

The HOPSIES algorithm is devised specially for situations that may take too long, or simply be impossible to solve with Knuth's algorithm. The situations may arise due to there being too many possibilities in the pool and the pool generation and comparison taking seemingly forever, or that the pool vector explodes under the pressure of too many possibilities.

The system divides the process of obtaining the correct code into 3 phases. The first phase being the testing phase, in which the solver creates attempt vector filled with the same symbol from 0 to the $number\ of\ symbol - 1$. For example, in the $length = 4$, $num = 6$ case, the solver creates attempt {0, 0, 0, 0} to {5, 5, 5, 5} and note the number of black hits returned in each attempt. Through the testing phase, the solver obtains information of all the symbols that are present in the correct code and pushes the symbols into a vector which is defined as a member data $std::vector < int >$ $number$ in the solver struct. The next phase looks for a vector that contains 0 black hits and store the vector as the starting attempt. The final phase of the solver algorithm exchanges the symbol of each index in the starting attempt with the symbols stored in the number vector until the index returns a black hit, at which point the solver stores the new code as the starting attempt and start exchanging symbols in its next index. The final phase is repeated until every index returns a black hit.

Defining member data:

```
1.    int num_of_attempt = 0;//declare an integer which
counts the number of times an attempt has been made, increases
```

```
everytime the learn function is called, important for
indicating which phase the solver is at
```

```
1.          std::vector<int> number;//contains all the
symbols that have been proven to be present in the correct
code, symbols arranged from the least number to the greatest
number
```

```
1.      std::vector<int> test_attempt;//contains a code that
is defined as a starting vector to have its index checked and
exchanged for black hit
```

```
1.      std::vector<int> tested_num; //contains number
already checked for blackhit within the exchanging process of a
specific index, must be cleared when the exchanging process
moves forward to the next index
```

```
1.      bool found_test_attempt = false;//this boolean
variable stores the information of whether a starting vector is
found
```

```
1.      int check_num = 0;//this integer indicates the index
of the symbol contained in the numbervector to be used to
exchange with the index in the test_attempt to check for balck
hit
```

```
1.      int check_index = 0; //contains the next index to be
checked for blackhit
```

```
1.      bool high_order = false;//this boolean variable
stores the information of whether the HOPSIES algorithm should
be used
```

```
1.      int black_tmp = 0;//this stores the black hits
obtained by the last attempt sequence, it is initialized to 0
because the test_attempt starts with a vector sequence of 0
black hits
```

Algorithm steps:

1. Testing phase: $num\_of\_attempt < num$

- Create_attempt:

```
1.     if(num_of_attempt < num){
2.           for(int i = 0; i < length; i++){
3.             attempt.push_back(num_of_attempt);
4.           }
5.         }
```

The $create\_attempt$ part of this phase generates an attempt filled with the same symbol. The symbol is represented by $num\_of\_attempt$, so that the symbols generated for each attempt does not go beyond the max symbol represented by num.

- Learn:

```
1.     if(num_of_attempt < num){
2.           for(int i = 0; i < black_hits; i++){
3.             number.push_back(num_of_attempt);
4.           }
5.           if((black_hits == 0) && (!found_test_attempt)){
6.             found_test_attempt = true;
7.             test_attempt = attempt;
8.           }
9.         }
```

The learning part of the testing phase takes in input of each testing attempt and push back the exact number of the symbols present in the actual code sequence, according to the number of black hits returned for each testing symbol. For example, if the actual code sequence is {4, 5, 1, 0, 1, 0}, the learn function in testing phase creates a $number$ vector such that it contains {0, 0, 1, 1, 4, 5}, which are the exact symbols present in the actual code sequence in an ascending order.

The second responsibility taken by the learn function in testing phase is to look for a vector sequence that contains 0 black hits and store it as the $test\_attempt$, if such vector sequence appears in the testing phase. This is implemented through the if condition. If such conditions are fulfilled, the $found\_test\_attempt$ confirms the state of having found the $test\_attempt$.

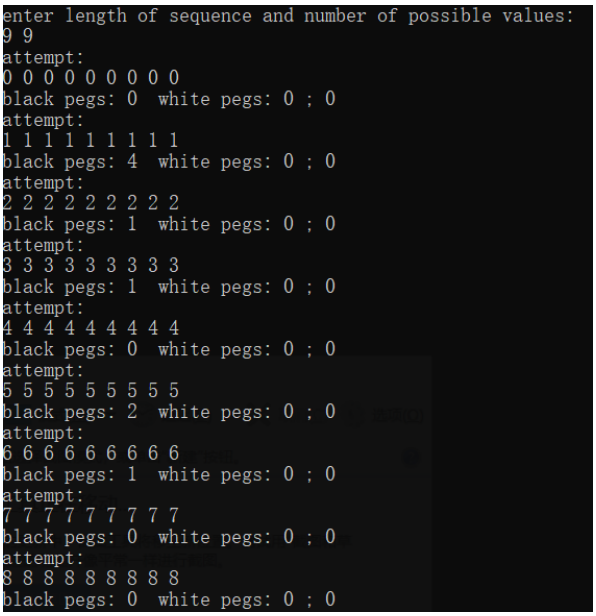

Fig. An example of the testing phase

2. Finding $test\_attempt$: ($num\_of\_attempt \geq num$) && (! $found\_test\_attempt$)

   The HOPSIES algorithm enters a phase of finding a $test\_attempt$ with 0 black hits as a starting vector for the final exchanging phase if no such vector sequence is found within the testing phase. Although necessary, this is undesirable as it costs extra attempts to locate such a vector sequence.

   - Create_attempt:

```
1.     else{//after testing phase
```

```
2.           if(!found_test_attempt){
3.             for(int i = 0; i < length; i++){
4.             attempt.push_back(randn(num));
5.           }
6.         }
```

   The $create\_attempt$ function enters a phase of generating $test\_attempt$ if it has not yet been found, as indicated by $found\_test\_attempt$. Attempts are created as random integers within the range of symbols being pushed into attempt vectors. This is essentially generating random sequence until a suitable $test\_attempt$ with 0 black hits is found.

   - Learn:

```
1.     else{//after testing phase
```

```
2.            if((black_hits == 0) && (!found_test_attempt)){
3.              test_attempt = attempt;
4.              found_test_attempt = true;
5.            }
```

The learn function at this phase checks whether a suitable test_attempt is found. Similarly, the learn function only enters this phase if $found\_test\_attempt$ is false. As soon as the attempt with 0 black hits is obtained, the attempt sequence is stored as $test\_attempt$.

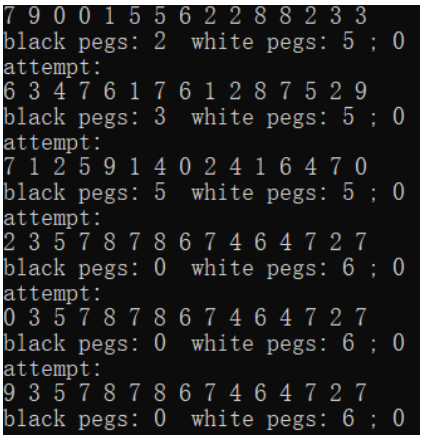

Fig. An example of random generation of test_attempt

3. Index symbol exchange:

This is the phase in which the algorithm starts looking for the correct sequence by exchanging the symbol in each index of the $test\_attempt$ with symbols form the $number$ vector.

- Create_attempt:

```
1.    else{ //found_test_attempt = true
2.            attempt = test_attempt;//test_attempt kept unchanged
3.            attempt[check_index] = number[check_num];//check each
   index of the attempt with numbers from numbervector, starting
   with check_index = 0, check_num = 0
4.          }
```

Since $check\_index$ and $check\_num$ are initialized to 0, the index exchange process is made sure to start at the first index of the $test\_attempt$ vector.

- Learn:

```
1.    else if(found_test_attempt){
2.              if(black_hits > black_tmp){
```

```
3.                test_attempt = attempt;
4.                number.erase(number.begin() + check_num);
5.                check_index++;
6.                check_num = 0;
7.                black_tmp = black_hits;
8.                tested_num.clear();
9.              }
10.               else{
11.                 tested_num.push_back(number[check_num]);
12.                 for(int i = 0; i < number.size(); i++){
13.                   if(!check_contain(number[i], tested_num)){
14.                     check_num = i;
15.                   }
16.                 }
17.               }
18.             }
```

The learn function gives different outcome depending on two different situations.

1) If the black hits returned by current attempt is greater than that was returned in the last attempt, this means that the index has obtained a black hit. Hence, the $learn()$ function tells the $create\_attempt()$ to look for black hit at the next index by incrementing $check\_index$ and reset $check\_num$ to 0, so that all possible symbols in the number vector are checked. The efficiency of the index exchange process can be increased by eliminating symbols in the number vector that are already confirmed to be black hit by calling the erase function to erase the $check\_num$ index which has just been checked.

   Furthermore, the $tested\_num$ vector must be cleared every time a black hit is found.

2) If the black hit returned by current attempt is the same as that was returned in the last attempt, this means that the symbol checked does not fit for black hit in this index. Hence the $learn()$ function keeps the index the same and alter the next symbol to be checked by altering the index to be checked in the number vector. Again, the efficiency can be improved if a vector keeps track of the symbol that has been checked so that the algorithm does not check repeated symbol on a single index. This is achieved with the vector $tested\_num$, which contains symbols that have already been checked for black hit, and the function $check\_contain(int\ n, const\ std::vector < int > \&\ v)$, which returns true if the input integer is contained in the input vector and false otherwise. Hence, within the next for loop the $learn()$ function makes $check\_num$ the next index within the number vector that stores the symbol that has not been checked for black hit, preventing the same symbol being checked more than once.

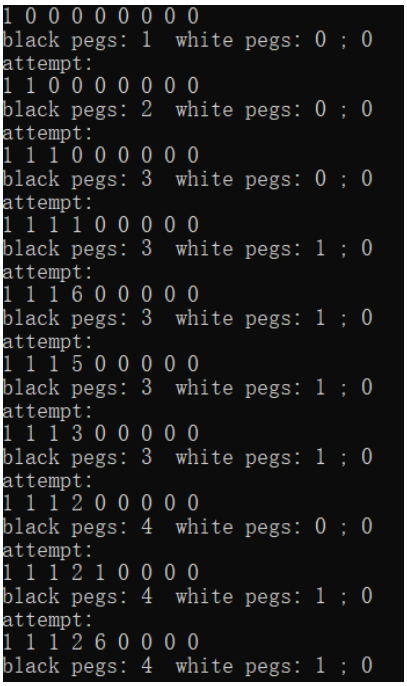

Fig. An example of Index symbol exchange phase

- Solver algorithm: Higher Order Pool Size Low Randomness Index Exchange System (HOPSLRIES)

Although the HOPSIES algorithm is good at solving code with large length and number of symbols, it sacrifices number of attempts to reduce the time taken for solving a code. Clearly, the biggest problem with the HOPSIES algorithm is in its second phase, in which if a starting attempt with 0 black hit is not obtained within the first testing phase the algorithm resorts to checking random sequence to obtain a $test\_attempt$, which costs extra attempts. This is a problem, especially if the length of the code is significantly greater the number of symbols. Because as the number of symbols available for choice at an index becomes less, there's a higher chance to obtain black hit. This is obvious later in this report when the HOPSIES algorithm is put into test. To resolve this issue, a slightly improved algorithm called Higher Order Pool Size Low Randomness Index Exchange System is introduced.

The HOPSLRIES uses the same member data and has a similar testing phase as with HOPSIES. The main difference between HOPSLRIES and HOPSIES is in their index symbol exchange phase. HOPSLRIES is better than HOPSIES in solving code for which length is greater than num in that it abandons the approach of having to find a starting vector with 0 black hits. Instead, any vector sequence from the testing phase could be used as its $test\_attempt$.

Algorithm steps:

1. Testing phase: $num\_of\_attempt < num$
    - Create_attempt:

```
1.     if(num_of_attempt < num){
2.           for(int i = 0; i < length; i++){
3.             attempt.push_back(num_of_attempt);
4.           }
5.           if(num_of_attempt == 0){
6.             test_attempt = attempt;//define test_attempt
7.           }
8.         }
```

$Create\_attempt$ in testing phase is implemented with the same testing attempt generating ability to generate vector sequence filled with same symbol. However, it also makes the first testing attempt, which is a vector filled with 0, the $test\_attempt$ in the index symbol exchange phase.

- Learn:

```
1.     if(num_of_attempt < num){
2.           for(int i = 0; i < black_hits; i++){
3.             if(num_of_attempt != 0){
4.               number.push_back(num_of_attempt);
5.             }
6.             else{
7.               black_tmp = black_hits;//stores black_hits for
   first test_attempt
8.             }
9.           }
10.        }
```

The $learn()$ function in testing phase features some slight altercations. Firstly, it no longer pushes the symbol 0 into the number vector, as in the index symbol exchange phase the index is no longer exchanged with the symbol 0 to avoid duplication. Secondly, the testing phase of the $learn()$ function sets $black\_tmp$ as the number of black hits returned for the first testing vector sequence, which is the vector filled with symbol 0, because the $test\_attempt$ is set as the vector filled with 0. Therefore, $black\_tmp$, representing the number of black hits from the last attempt to be compared with the number of black hits from the present attempt, must be initialized to the black hits obtained by the $test\_attempt$.

2. Index symbol exchange:

- Create_attempt:

```
    else{//after testing phase
        attempt = test_attempt;
        attempt[check_index] = number[check_num];
      }
```

Same implementation as in HOPSIES.

- Learn:

```
    else{//after testing phase
      if(black_hits < black_tmp){
        check_index++;
        check_num = 0;
      }
      else if(black_hits == black_tmp){
        tested_num.push_back(number[check_num]);
        for(int i = 0; i < number.size(); i++){
          if(!check_contain(number[i], tested_num)){
              check_num = i;
            }
          }
        }
        else if(black_hits > black_tmp){
          test_attempt = attempt;
          number.erase(number.begin() + check_num);
          check_index++;
          check_num = 0;
          black_tmp = black_hits;
          tested_num.clear();
        }

      }
```

Again, the learn function gives different outcome depending on different conditions, only this time there are three different situations.

1) The first situation where current $black\_hits$ is smaller than previous $black\_tmp$ suggests that an index with black hit is replaced with other symbol. Hence no change is made, and the $learn()$ function tells the $create\_attempt()$ function to check for the next index.
2) The second situation where current $black\_hits$ is equal to previous $black\_tmp$ means the index being checked is not a black hit by itself, and the previous attempt to change the index with different symbol did not obtain a black hit. Therefore, again, the $learn()$ function pushes the symbol that has

been checked into the $tested\_num$ vector and sets $check\_num$ to the next index in the number vector that corresponds to a different symbol.

3) The third situation where current $black\_hits$ is greater than previous $black\_tmp$ means the index being checked is not a black hit by itself, and through exchanging index symbol, the index has been made into a black hit. Hence the $learn()$ function tells the $create\_attempt()$ function to check for the next index, resets the $check\_num$ parameter to 0, update the $black\_tmp$ parameter with the new $black\_hits$, and clears the $tested\_num$ vector.

```
enter length of sequence and number of possible values:
15 3
attempt:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
black pegs: 6  white pegs: 0 ; 0
attempt:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
black pegs: 5  white pegs: 0 ; 0
attempt:
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
black pegs: 4  white pegs: 0 ; 0
attempt:
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
black pegs: 5  white pegs: 2 ; 0
attempt:
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
black pegs: 6  white pegs: 1 ; 0
attempt:
0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
black pegs: 7  white pegs: 0 ; 0
attempt:
0 2 1 0 0 0 0 0 0 0 0 0 0 0 0
black pegs: 8  white pegs: 0 ; 0
attempt:
0 2 1 1 0 0 0 0 0 0 0 0 0 0 0
black pegs: 9  white pegs: 0 ; 0
attempt:
0 2 1 1 1 0 0 0 0 0 0 0 0 0 0
black pegs: 10  white pegs: 0 ; 0
attempt:
0 2 1 1 1 1 0 0 0 0 0 0 0 0 0
black pegs: 11  white pegs: 0 ; 0
attempt:
0 2 1 1 1 1 1 0 0 0 0 0 0 0 0
black pegs: 12  white pegs: 0 ; 0
attempt:
0 2 1 1 1 1 1 2 0 0 0 0 0 0 0
black pegs: 11  white pegs: 2 ; 0
attempt:
0 2 1 1 1 1 1 0 2 0 0 0 0 0 0
black pegs: 13  white pegs: 0 ; 0
attempt:
0 2 1 1 1 1 1 0 2 2 0 0 0 0 0
black pegs: 14  white pegs: 0 ; 0
attempt:
0 2 1 1 1 1 1 0 2 2 2 0 0 0 0
black pegs: 13  white pegs: 2 ; 0
attempt:
0 2 1 1 1 1 1 0 2 2 0 2 0 0 0
black pegs: 15  white pegs: 0 ; 0
the solver has found the sequence in 16 attempts
the sequence generated by the code maker was:
0 2 1 1 1 1 1 0 2 2 0 2 0 0 0
```

Fig. An example of HOPSLRIES solving code with length much bigger than number of symbols

Evaluation

The evaluation of efficiency of algorithm is based on the number of attempts taken to break the code. The comparison will be carried out for different length and number of symbols of code. The Knuth's method will be tested for points at which the 10 second timeout limit is

reached and point at which the method breaks down due to having too many possibilities which exist the capacity of the pool

- Knuth's method

Knuth algorithm is the most efficient in terms of the number of attempts taken to solve a code. The problem with this method is that it considers all the possible arrangement, which grows exponentially with increasing length of the code. For higher order code, not only does it take more attempts to solve, but it also takes significantly longer time to return the correct sequence.

Through testing, a surprising correlation between num and length and average attempts to solving the code has been found. Another relation found is the time taken to solve the code and the total number of possible arrangements of code.

num\length	4	5	6	7	8	9	10	11	12	13
4	3.799	4.16	4.48	4.85	5.5	5.7	6.4	6.9	7.5	N/A
5	4.221	4.62	5.08	5.61	5.8	6.33	7	N/A	N/A	
6	4.615	5.08	5.42	6	6.5	8	N/A			
7	5.014	5.45	5.95	6.6	7	N/A				
8	5.434	5.73	6.4	6.7	7.5					
9	5.853	6.28	6.7	7.25	N/A					
10	5.95	6.36	7.1	7.4						
11	6.6	6.98	7.2	N/A						
12	6.89	7.2	7.4							
13	7.12	7.5	8.4							
14	7.37	7.5	8.5							
15	8.09	8.7	8.7							

Fig. A table showing the average attempts taken to solve a code, where a red box indicates the time taken to solve a code is between 9 and 10 seconds

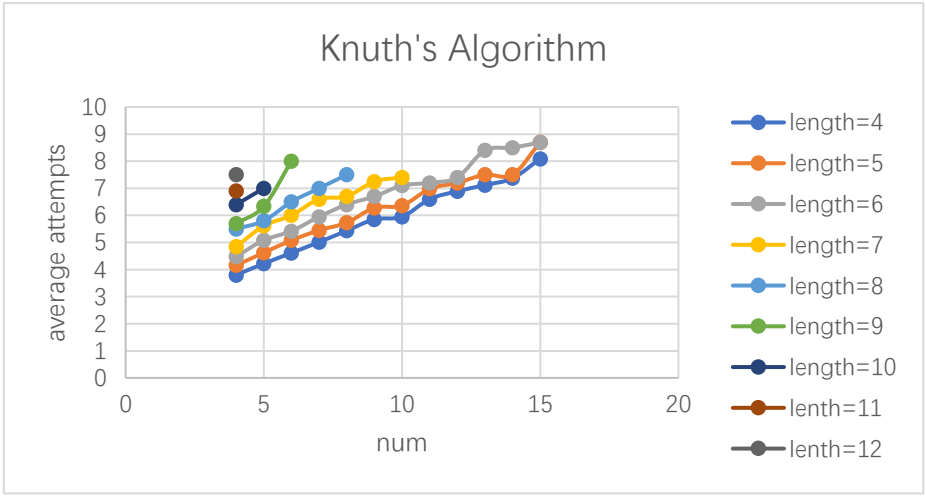

Fig. A graph showing the average attempts to solve a code against the num of the code for each length of symbols

A relationship is revealed as increasing length and number of symbols results in increasing average attempts to solve the code, which is as expected. Interestingly, this also illustrates that there is no direct connection between the maximum number of possibilities of a code and the average attempts taken to solve it. For example, in the length=4, num=15 case there

are $15^4 = 50625$ possibilities, but its average attempt is higher than cases that have less number of symbols but higher possibilities, such as the length=9, num=4 case, where there are $4^{10} = 1048576$ possibilities.

Another revelation through the testing is that there seems to be a connection between the time taken to obtain a correct code sequence and the possibilities of code arrangements.

$$11^6 = 1771561$$
$$8^7 = 2097152$$
$$5^9 = 1953125$$

It turns out that, through evaluation of number of possibilities of code sequence that are within the 10-9 seconds (red box) limit, they have similar pool sizes. This reveals the maximum possibilities of code arrangements that can be solved by the Knuth's algorithm within 10 seconds, and therefore, is useful in terms of determining when to switch algorithm and optimizing the solver.

Hence the switching point between Knuth's algorithm and the higher order ones is set at the middle one of the three. The solver switches from Knuth's method to the higher order ones when the code has possible arrangements that are greater than 1953125. This is implemented in the $init()$ function in the solver struct.

```
1.    if(log(pow(num, length)) > log(1953126)){
2.          high_order = true;
3.        }
```

● HOPSIES algorithm

The HOPSIES algorithm is efficient time wise for large code length and number of symbols. However, its drawback is obvious in its phase of randomizing starting attempt, which costs extra attempts. In the above algorithm section, it is predicted that the HOPSIES algorithm is more likely to take extra attempts when the length of the code is significantly greater than the number of symbols. This is proven by experiments, as shown in the below table and graph.

num\leng	4	5	6	7	8	9	10	11	12	13	14	15
4	10.24	13.17	15.29	21.5	24.01	29.01	40.39	42.12	59.08	65.83	85.07	119.4
5	11.14	13.66	15.96	19.24	21.93	26.12	32.3	34.24	42.43	47.76	56.15	66.39
6	12.06	14.59	17.16	20.34	22.35	26.66	29.29	33.94	38.21	43.25	49.95	52.2
7	13.26	15.59	18.15	21.66	24.13	27.16	30.78	34.12	38.5	43.65	47.37	51.73
8	14.43	16.94	19.92	22	25.89	28.99	32.79	35.66	39.11	42.6	50.16	52.81
9	15.3	18.15	20.56	23.74	27.05	30.76	34.12	37.51	41.79	44.76	50.66	53.43
10	16.51	19.29	22	25.47	28.18	31.49	35.51	39.69	43.29	47.8	52.43	57.58
11	17.9	20.17	23.14	26.08	29.6	33.85	37.46	40.77	44.89	49.97	54.99	58.91
12	18.63	21.12	24.5	27.35	31.38	34.48	38.53	42.68	46.58	51.85	57.23	60.96
13	19.62	22.29	25	28.41	32.28	36.33	39.45	44.57	48.93	54.8	59.33	63
14	20.48	23.53	25.4	29.36	33.37	38.19	41.71	46.18	50.2	55.67	59.44	66.03
15	21.64	23.93	27.5	30.36	34.28	38.44	42.44	47.01	51.35	59.07	62.12	66.68

Fig. A table showing the average attempts taken by the HOPSIES to solve the code for each

code length and number of symbols

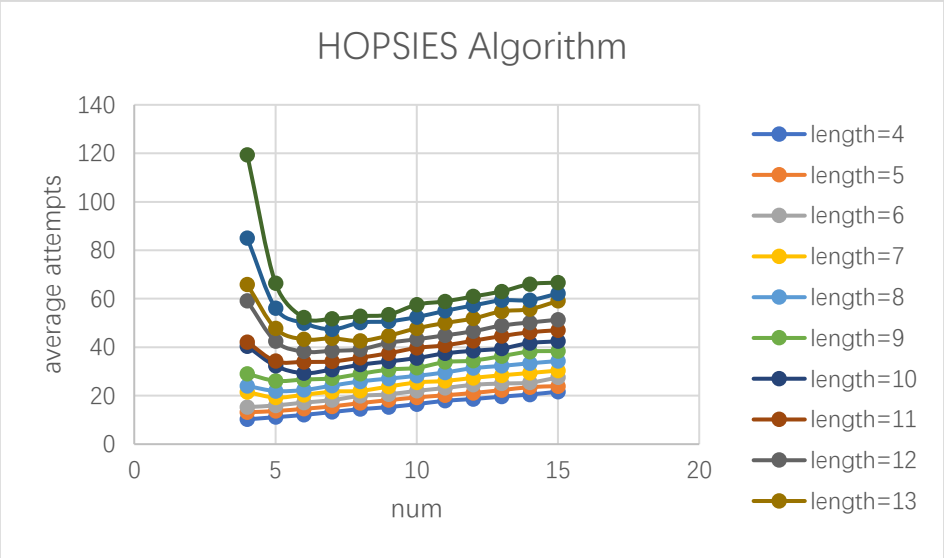

Fig. A graph showing the average attempts to solve a code against the num of the code for each length of symbols

The drawback of the HOPSIES is clearly visible through the non-linear behavior of the plot for higher code length and low number of symbols.

- HOPSLRIES algorithm

The HOPSILRIES is devised to resolve this non-linearity at higher code length. And it is predicted in the algorithm section that its performance will not be affected by mismatching code length and number of symbols, as it does not require a starting vector sequence with 0 black hits. This is shown in the below table and graph.

num\leng	4	5	6	7	8	9	10	11	12	13	14	15
4	8.59	10.14	11.45	13.34	15.21	16.9	18.04	20.39	22.16	23.28	24.76	28.02
5	9.64	11.78	13.45	15.15	17.71	19.48	21.62	23.56	25.15	27.77	29.09	31.92
6	11.37	13.27	15.8	17.35	19.21	21.21	24.25	26.98	29.32	30.99	34.15	36.5
7	12.38	14.65	16.44	19.14	21.18	23.99	26.44	29.47	32.25	34.87	37.52	40.68
8	13.69	15.64	17.8	20.64	23.02	26.2	27.85	32.01	35.5	37.01	40.61	44.52
9	14.67	17.01	18.74	22.22	25.26	27.31	30.61	33.77	37.87	39.75	43.45	47.54
10	16.09	18.26	20.5	23.36	26.21	28.64	32.61	35.47	39.15	42.45	45.8	49.99
11	17.16	19.04	22.28	24.97	27.94	31.71	34.75	36.71	41.48	46.59	48.98	53.65
12	17.94	20.26	22.71	25.85	29.73	32.97	36.05	38.99	42.37	46.96	51.18	56.33
13	18.89	21.39	24.34	26.66	30.27	33.5	37.29	40.99	43.91	48.31	52.96	57.59
14	20.07	22.49	25.46	28.47	30.67	35.41	38.1	42.32	47.36	51.45	54.79	59.19
15	21.19	23.47	26.08	29.57	32.6	36.8	40	44.61	47.74	52.53	57.78	61.34

Fig. A table showing the average attempts taken by the HOPSLRIES to solve the code for each code length and number of symbols

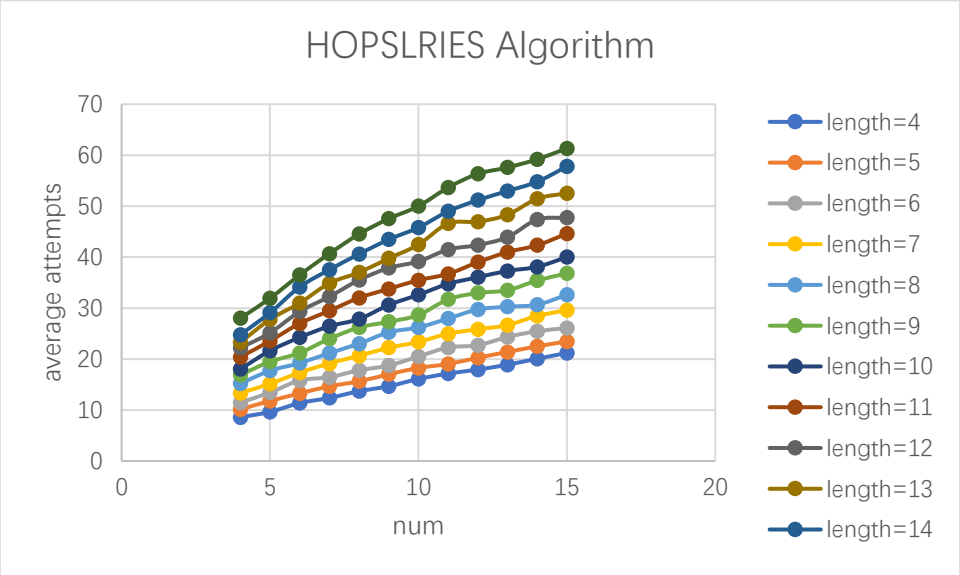

Fig. A graph showing the average attempts to solve a code against the num of the code for each length of symbols

The plot for HOPSLRIES demonstrate a nice upward trend that is similar to that displayed by the Knuth's algorithm. This shows that the HOPSLRIES algorithm is successful in reducing attempts when the code length is greater than the number of symbols.

Interestingly, by comparing the tables for HOPSIES and HOPSLRIES, it is noticed that HOPSLRIES has a lower average attempts value at almost every instance of code length and number of symbols. This shows that the HOPSLRIES algorithm generally of better efficiency than the HOPSIES algorithm. Therefore, the application of solver uses HOPSLRIES entirely.

References:

1. Donald E. Knuth, The Computer As Master Mind, J. Recreational Mathematics, Vol.9, 1976-77