An efficient algorithm to solve Mastermind

Andreas Floros

CID: 01539636

Abstract

The aim of this report is to present an attempt efficient algorithm for solving a game of mastermind with an arbitrary length and an arbitrary number of symbols used. The algorithm presented is based on Donald Knuth's approach to the problem which he showed for the case of length = 4 and number of symbols = 6 in his paper.

Table of Contents

Introduction

Rules of the game

Mastermind is a two-player game where one player (the code maker) forms a sequence of length l consisting of a maximum n symbols[1], and the second player (the code breaker) tries to guess the sequence. After each guess the code breaker receives information from the code maker in the form of black pegs (black hits) and white pegs (white hits). The black pegs that the code breaker receives indicate how many symbols they placed in the correct position while the white pegs indicate how many symbols they placed in the wrong position that are included in the sequence. Given this information, the code breaker's goal is to find the solution in a minimum number of attempts.

Calculating black hits and white hits

When designing an algorithm for Mastermind we need to calculate the black hits and the white hits. For the black hits this is simple, we are simply required to check the sequence with the code breaker's guess and find all the places where they match. For the white hits there are two different approaches that can be taken. Knuth's paper[2] gives a formula for evaluating the total hits (i.e. the sum of black hits and white hits):

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

Where n_i denotes the number of times the symbol i appears in the sequence and n_i' the number of times it appears in the code breaker's attempt. (The formula shown above assumes 6 symbols.)

This result can be used in combination with the black hits calculated to yield the white hits. The second approach makes use of the definition that Knuth gives in his paper for white hits. He defines the white hits

[1] In this report, the symbols are numbers ranging from 0 up to and including $n - 1$. We denote the symbols used in a game as n and the length as l.
[2] http://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf

as "the number of positions j such that $x_j \neq y_j$ but $x_j = y_k$ for some k and y_k has not been used in another hit." Both approaches have advantages and disadvantages which will be discussed further in section Knuth's algorithm and variations.

Solving Mastermind within a timeout

Algorithm complexity

Solving a game of Mastermind in the most attempt efficient way requires a lot of calculations and considerations. As an example, consider the base case with 6 symbols and a length of 4. The code breaker must find one sequence out of the 6^4 possible sequences. The size of the set containing all possible sequences in this case is relatively small compared to an average computer's limitations, and so, calculations in this case do not pose a problem. However, note that the size of this set increases exponentially with the length and in polynomial fashion with the number of symbols (n^l). This suggests a rapid increase in calculations as we move to larger instances. Clearly, when designing a Mastermind solver, the complexity of the functions used needs to be considered to get results in a reasonable amount of time.

If a general filtering strategy is used, where the entire set of all sequences is considered and impossible choices for the solution code are filtered out as more information is obtained from the code maker, then we observe the following:

In the context of C++ (assuming vector usage for our algorithm), it is important to note that member function vector.erase() has complexity[3] $O(n)$, which can hinder the solver's time performance. Of course, one way to get around this is by observing that the complexity of erase is due to reallocations happening. Therefore, before erasing an element from a vector it would be advantageous to swap[4] it with the last element of the vector. Applications of this method include erasing elements from the set of all potential solutions once it has been determined that they can't be the solution. Other ways to bypass erase's linear complexity is by recreating the set of potential solutions every time and assigning it to the

[3] http://www.cplusplus.com/reference/vector/vector/erase/
[4] Swap member function has constant complexity http://www.cplusplus.com/reference/vector/vector/swap/

previous set[5]. While these approaches make the solver slightly more time efficient, they still fail at larger instances since, the rapid growth of the aforementioned set ultimately decides the algorithm's time performance. The problem with this approach lies in the fact that all possibilities are considered simultaneously.

From the above, it seems that generating all sequences doesn't generalize to large instances. However, if instead we interpret these sequences as numbers in base n (since the symbols used are non-negative integers up to n) we can sort them in increasing order.

$$0 \dots 000, \qquad 0 \dots 001, \qquad 0 \dots 002, \qquad \dots, \qquad (n-1) \dots (n-1)(n-1)(n-1)$$

And we can check them up to the first potential solution that we find. Once the first sequence that fails the filtering criterion is reached it should be chosen as the code breaker's next guess. The reason for this is because choosing the first potential solution as our next guess is as attempt efficient as choosing a random potential solution. Furthermore, this method requires the first potential solution to be chosen as the next guess because it goes through all sequences in increasing order, therefore choosing a different potential solution would complicate our search algorithm. After guessing, the code breaker would then resume searching for potential solutions from the point where they left off. Note that the set of potential solutions is never stored when this method is used which is what makes it extremely time efficient with respect to the previous methods.

Knuth's algorithm and variations

Knuth's/Swaszek's algorithm

Donald Knuth presents the following algorithm[6] for solving Mastermind:

1. Create the set of all sequences corresponding to the number of symbols and the length of the game.

[5] Assuming our filtering strategy is optimal, the new set will be significantly smaller than the previous set. This would make assigning the new set to the previous one much more time efficient compared to erasing elements.
[6] https://en.wikipedia.org/wiki/Mastermind_(board_game)

2. Guess[7].
3. Receive feedback from the code maker. // If black hits = length the game is over.
4. Filter out of the set all sequences that wouldn't have produced the same number of black hits and white hits as the ones just received had the previous attempt been the solution to obtain a reduced set.
5. Apply a minimax technique to find the next guess: For every unused sequence calculate how many sequences would be excluded from the reduced set assuming the worst-case scenario. (i.e. the combination of black hits and white hits given by the code maker that would eliminate the minimum number of sequences from the reduced set for that unused sequence.) The sequence which excludes the maximum number of sequences from the reduced set in the worst-case scenario should be the code breaker's next guess.
6. Repeat from 2.

Note that the 4th step is a simple yet effective filtering strategy: if a potential solution is the solution that means that when it is compared to the previous guess it should produce the same results as the solution. Also note, step 5 implies that the next guess may be outside of the reduced set. Despite this, it still is valuable to the code breaker since it manages to minimize the set of potential solutions. Knuth's method is extremely effective in terms of attempts. However, it also extremely time inefficient. As mentioned before, generating the original set of all sequences is impractical, let alone applying the minimax technique. A modified version[8] of Knuth's algorithm which is more time efficient skips the 5th step and picks a sequence from the reduced set (Swaszek's algorithm).

Optimizing Swaszek's algorithm

Making use of Swaszek's algorithm for larger instances proves to be time inefficient past 15^5. However, if instead of generating all sequences we use the approach described in the previous section to obtain the first potential solution, Swaszek's algorithm can be used with better time efficiency. It is evident, that even this method has its limitations: If χ is a potential solution that was previously picked, and the next potential solution is orders of magnitude away from χ, all sequences in between would have

[7] In his paper, Knuth suggests that picking 1122 as an initial guess for a game with 6 symbols (1 to 6) and a length of 4 is the optimal initial guess.
[8] Swaszek's algorithm http://mathworld.wolfram.com/Mastermind.html

to be checked. A key observation that allows us to bypass this issue is that a sequence can be excluded from the potential solutions even before its last digit is reached. For example, assume a game of mastermind where the code breaker guessed 0234 and received 0 black hits. Any sequence starting with 0 is then immediately excluded from the potential solutions because when compared to 0234 it would produce at least 1 black hit[9]. In this example, the code breaker immediately skips[10] to 1000 to check for potential solutions. Of course, this generalizes to larger instances. By this method several orders of magnitude can be skipped, saving a tremendous amount of time. It is now obvious what the advantage of the second method for calculating white hits (described in the 1st section of the report) is: it allows the code breaker to check if a sequence isn't a potential solution before it reaches its last digit, and so, it allows for magnitude jumps utilizing the white hits. Therefore, when evaluating white hits, the 2nd method should be used as it provides the code breaker with more information than the 1st one which, while simple, doesn't offer any information regarding the digits of the code breaker's attempt that make it invalid.

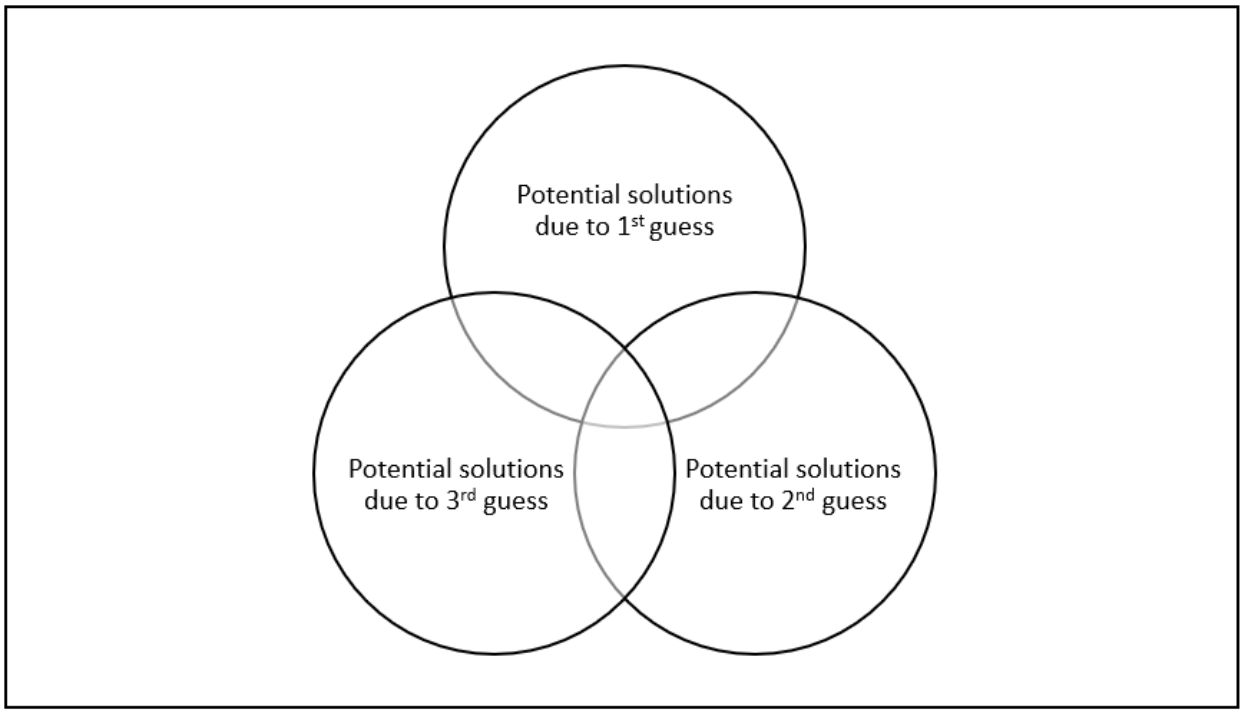
In particular, if the black hits or the white hits at any point during their calculation exceed the black hits or the white hits respectively given by the code maker, we can immediately deduce that the guess that is being checked can't be the solution since it has a higher number of black hits/white hits than what was provided. Also, if at any point the black hits + the length from the end is less than the black hits provided we can also conclude that the guess being checked isn't the solution. For example, assuming 0234 gives 3 black hits, the code breaker concludes that codes starting with 23 (length from the end = 2) can't be the solution since there is no way for them to yield more than 2 black hits when compared to 0234. For white hits a similar argument can be used. If the white hits + 2*the length from the end is less than the white hits given, then the guess being checked is invalid. Note that the reason for multiplying the length from the end by a factor of 2 is because as the two sequences are being compared, a maximum of 2 white hits can arise from one digit. If, for example, 0234 is compared with 0243 the code breaker's calculations would yield: 1 black hit and 0 white hits for the 1st sub length (comparing 0 with 0), 2 black hits and 0 white hits for the 2nd sub length (comparing 02 with 02), 2 black hits and 0 white hits for the 3rd sub length (comparing 023 with 024), 2 black hits and 2 white hits for the 4th sub length (comparing 0234 with 0243).

To further optimize Swaszek's algorithm, the code breaker should try to maximize the magnitude of the jumps. To do this, the maximum possible jump should be extracted from a guess before the code

[9] Therefore, it should be filtered out by the criterion described in step 4 of Knuth's algorithm.
[10] Skips like the one explained will be referred to as jumps and are performed by the jump function in the code provided.

breaker moves on to the next one. For example, assume 0234 produced 0 black hits and 0100 produced 0 white hits. When they are compared with 1230 the 1st sequence suggests that codes starting with 123 are not potential solutions while the 2nd suggests that codes starting with 12 are not potential solutions. In this case the code breaker should jump to 1300, not 1240.

Finally, to push the boundaries of Swaszek's algorithm even further, a depth[11] variable is introduced which controls how much of the previous information obtained from the code maker is considered during the filtering process. The diagram below illustrates this idea.



The set defined by the intersection of the potential solutions due to the 1st guess and the potential solutions due to the 2nd guess is the reduced set obtained after 2 guesses while the intersection of all 3 sets shown is the reduced set obtained after 3 guesses.

After 3 guesses the code breaker knows that the code must lie in the intersection of the 3 sets shown above. Obviously, the more guesses the code breaker makes the smaller the intersection becomes until it converges to the solution code. This implies that as the game develops potential solutions become harder to find which could cause the code breaker to waste a lot of time. Therefore, once a limit[12] is reached, the

[11] In the code provided, the depth variable represents how far back the code breaker looks when filtering codes. The maximum depth is when depth = 0, where the code breaker looks at the information provided by the code maker from the very first attempt.

[12] This limit is in the form of the number of jumps the code breaker makes and it is defined such that all instances up to 15^{15} are solvable within 10 seconds.

code breaker should consider a larger set (i.e. the previous set) of potential solutions to solve the game within a reasonable amount of time. In doing so, the code breaker has a relatively high chance of finding the solution within the next guess (compared to picking a random guess) and can further minimize the set of potential solutions because, as mentioned above, sequences that aren't potential solutions are still valuable to the code breaker for minimizing the potential solutions set.

The tables below show the average attempts of a non-optimized Swaszek algorithm implementation and the average attempts of the optimized version for various instances.

num/length	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1.39	1.95	2.37	2.71	3.06	3.64	4	4.49	4.77	5.15	5.7	5.91	6.46	6.71	7.09
3	1.91	2.3	2.88	3.07	3.51	3.77	4.28	4.54	4.88	5.43	5.79	6.25			
4	2.53	2.8	3.33	3.74	4.15	4.41	4.87	5.37	5.86	6.42					
5	2.85	3.37	3.68	4.2	4.59	4.94	5.4	6.02							
6	3.27	3.81	4.06	4.44	5.04	5.58	6.16								
7	3.48	4.2	4.58	4.89	5.38	5.97	6.45								
8	4.51	4.47	4.87	5.3	5.89	6.32									
9	4.65	5.09	5.56	5.78	6.25	6.64									
10	5.69	5.53	5.78	6.14	6.51	7.09									
11	6.01	5.82	6.13	6.56	6.94										
12	6.38	6.5	6.54	6.71	7.12										
13	6.57	7.14	6.82	7.22	7.46										
14	6.71	7.4	7.31	7.58	7.62										
15	7.43	7.69	7.82	7.91	8.27										

Non-optimized Swaszek algorithm[13]

num/length	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1.43 | 2	2.15 | 3	2.33 | 4	3.07 | 5	3.06 | 6	3.74 | 6	4.24 | 7	4.65 | 8	5.21 | 8	5.62 | 9	5.95 | 11	6.01 | 9	6.68 | 10	7.13 | 10	7.68 | 10
3	2.03 | 3	2.38 | 4	2.89 | 5	3.21 | 5	3.92 | 6	4.01 | 7	4.41 | 6	4.65 | 7	5.28 | 8	5.64 | 8	6.77 | 10	7.56 | 12	8.92 | 16	10.9 | 18	12.8 | 23
4	2.43 | 4	3.07 | 5	3.56 | 5	3.84 | 5	4.13 | 6	4.63 | 6	5.31 | 7	5.73 | 8	6.7 | 10	8.3 | 14	9.85 | 17	12.17 | 21	14.81 | 29	18.56 | 33	23.84 | 44
5	2.88 | 5	3.52 | 6	3.96 | 7	4.54 | 7	4.89 | 7	5.35 | 7	5.95 | 9	7.2 | 11	9.14 | 15	11.04 | 17	13.28 | 26	17.5 | 37	21.44 | 46	25.57 | 57	29.54 | 75
6	3.68 | 6	4.27 | 7	4.53 | 7	4.85 | 8	5.53 | 9	5.96 | 8	6.97 | 12	8.92 | 15	10.76 | 17	13.55 | 23	16.12 | 31	20.73 | 45	25.07 | 61	28.88 | 73	
7	4.11 | 7	4.66 | 7	5.14 | 9	5.45 | 8	5.95 | 8	6.6 | 9	8.37 | 14	10.43 | 18	13.13 | 24	15.23 | 26	18.74 | 35	23.03 | 55	27.78 | 76		
8	4.67 | 8	5.41 | 9	5.76 | 8	6.1 | 8	6.56 | 9	7.56 | 12	9.1 | 16	12.28 | 20	14.66 | 25	17.68 | 34	22.16 | 43	26.56 | 51			
9	4.87 | 9	5.88 | 10	6.46 | 9	6.66 | 10	7.4 | 10	8.51 | 13	10.89 | 19	13.58 | 27	16.42 | 30	19.62 | 40	24.28 | 55	27.01 | 81			
10	5.93 | 10	6.27 | 10	6.81 | 10	7.09 | 10	7.78 | 11	9.71 | 16	12.3 | 22	14.73 | 24	17.46 | 30	21.49 | 44	26.84 | 53				
11	6.06 | 11	7.05 | 11	7.44 | 12	7.52 | 11	8.62 | 11	10.45 | 16	12.98 | 23	16.48 | 34	19.08 | 37	23.3 | 42	28.01 | 60				
12	5.9 | 12	7.83 | 12	8.33 | 12	8.04 | 11	9.18 | 13	11.15 | 19	13.71 | 24	17.21 | 32	19.45 | 37	24.38 | 48	29.64 | 60				
13	6.93 | 13	8.18 | 13	9.01 | 13	9.11 | 12	10.1 | 14	12.34 | 20	15 | 26	17.63 | 32	21.34 | 43	28.29 | 50	30.33 | 64				
14	8.01 | 14	8.62 | 14	9.75 | 14	9.47 | 13	10.91 | 16	12.94 | 20	16.07 | 31	19.12 | 38	23.46 | 48	27.63 | 50	32.25 | 64				
15	7.88 | 15	9.53 | 15	10.52 | 15	10.3 | 14	11.37 | 16	14.39 | 25	16.76 | 29	20.42 | 39	25.34 | 48	28.77 | 62	35.63 | 84				

Optimized version (average attempts | maximum attempts)[14]

The optimized version can reach large instances that the non-optimized version could never hope to reach. However, it obvious from looking at the maximum attempts in the 2nd table that the optimized version is unstable. Specifically, observe that the difference between the maximum attempts and the average attempts varies vastly. For example, the case of 9 symbols and length = 12 has an average of 27

[13] This was with the swap – erase trick that was mentioned in the previous section.
[14] Table taken with the limit variable set to 850 (see code).

attempts and a maximum of 81 attempts. This happens because of the depth limit that was introduced. Clearly, in this instance, the value chosen for the limit variable (see code) was too small and therefore severely restricted the code breaker's attempt efficiency. Observe also, that the average attempts for small instances are not the same for both algorithms. This suggests that even for small instances the limit variable plays a significant role. The less we restrict the code breaker through the limit variable the more stable the attempt averages/attempt maximums become but the time also increases. Choosing the optimal limiting values (i.e. values that would give the best attempt efficiency within a 10 second timeout) was done through experiments.

Generalization for large instances

Split algorithm

Despite our efforts to push the boundaries of Swaszek's algorithm further, larger instances (15^{15}) remain out of reach. Clearly, a different approach is required. If larger instances are interpreted as smaller instances being summed together[15] we can easily solve them. Of course, splitting a large instance into smaller instances comes with a cost, the filtering criterion that was previously used cannot utilize the white hits obtained from the code maker. This is because the white hits that are provided refer to the entire code, not sections of it[16]. Another drawback is that to be able to perform the filtering criterion on specific intervals of the code, a particular arrangement needs to be formed initially. This arrangement should give 0 black hits. The reason for this is that if the filtering criterion is attempted on a segment of the code, the black hits within that segment should be clearly defined. For example, if 02345 gives 1 black hit the code breaker can't split the code word and perform filtering because it is not known where this black hit is. This leads to uncertainty when it comes to the black hits of the segment being solved.

As an example, assume the split algorithm is performed on an instance with 15 symbols and a length of 13. Assume further that the code breaker decides to split this instance into lengths 5 and 8. To solve the game, the code breaker would have to find a sequence of length 13 satisfying the 0 black hits condition.

[15] For example, symbols = 15 and length = 15 can be thought of as one instance with 15 symbols and a length of 7 and another with 15 symbols and a length of 8.

[16] The one exception to this is when the last segment of the code is reached where the code breaker knows that all previous segments contain black hits and therefore all white hit information is contained within the final segment.

Once this is achieved, the code breaker solves the 1st segment of length 5 via the method discussed in the previous section (utilizing black hits only) and then solves the 2nd segment via the same method (in this case white hits can be utilized because the code breaker is certain that all white hit information is contained within that segment). Since the last segment of the code is solved using the white hits as well as the black hits, it is advantageous to make it the segment of the largest size (In the example provided, splitting 5 and 8 is better than 8 and 5).

Getting 0 black hits

An interesting observation is that getting 0 black hits isn't always a simple procedure. We note the following:

If $n > l$, then the code breaker can guarantee 0 black hits in a maximum of l attempts. This is because, in this case, a minimum of $n - l$ symbols will not be included in the code. This means that once l attempts are reached, (assuming the attempts consist of the same number) the code breaker will either have an attempt with 0 black hits or will be able to pick an attempt not previously chosen (i.e. an attempt consisting of a number that wasn't previously chosen) which is guaranteed to give 0 black hits. For example, if 5 symbols are used and the length is 3 then the code breaker would guess 000, 111 and 222. If none of these attempts gave 0 black hits that means that 0,1 and 2 are included in the sequence which implies that 3 and 4 are not (i.e. 333 and 444 give 0 black hits). Even with random guessing getting 0 black hits in this case isn't very attempt inefficient. In a probabilistic sense, the set of 0 black hits to the set of all sequences is $\frac{(n-1)^l}{n^l} = \left(1 - \frac{1}{n}\right)^l$. For example, when $n = 10$ and $l = 7$, this suggests that 47.83% of guesses are 0 black hit guesses. If $n = l$, the above formula reduces to $\left(1 - \frac{1}{n}\right)^n \approx \frac{1}{e} \approx 0.367$, or 36.7%. Finally, if $n < l$, less 36.7% of choices will yield 0 black hits. In the extreme case of $n = 2$, $l = 15$ the formula suggests that 0.0031% of choices will give 0 black hits.

In general, the ratio of 0 black hits sequences to all sequences is $\left(1 - \frac{1}{n}\right)^l = \left(\left(1 - \frac{1}{n}\right)^n\right)^{\frac{l}{n}} \approx e^{-\frac{l}{n}}$.

In the latter case it is obvious that random guesses will not suffice to produce 0 black hits.

Fortunately, due to the optimizations mentioned in the previous sections, the split algorithm is only utilized after 5^{15}. This means that at worst, the set of 0 black hits sequences to the total sequences is $\approx$

6.49% (when $n = 6$ and $l = 15$). Because of this, no sophisticated algorithm is required for obtaining 0 black hits.

If we are to randomly guess to reach 0 black hits a random generator function needs to be used. One approach would be to use the functions rand and srand which are provided in the standard library. Of course, to get random integers in a specific interval $[0,n)$ the modulus would have to be used. For example, we would write std::rand()%n. The problem with this approach is that this wouldn't produce uniformly distributed results and the reason for this is can be easily illustrated by considering the simple example of $n = 4$. Assuming rand gives a maximum value of 5 (this is much larger in reality but the point being made still holds) the possible outputs of the function described above are 0, 1, 2, 3, 0, 1 which clearly aren't uniformly distributed. Because the actual maximum value of rand() is much larger than the numbers we are working with, we could use this implementation and it would hardly make a difference. By experimenting with other randomizers, (such as the Mersenne twister) it was found that the only difference between more sophisticated randomizers and the standard approach is slight deviations in time performance. For this reason, functions rand and srand were used in the final algorithm. The table below shows the average attempts of the complete algorithm for various instances.

num/length	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1.47	1.97	2.35	2.84	3.2	3.75	4.14	4.58	5.13	5.44	5.85	6.33	6.73	7.12	7.58
3	1.99	2.48	2.88	3.32	3.58	3.98	4.36	4.74	5.2	5.55	6.4	7.42	7.61	9	10.53
4	2.44	2.99	3.43	3.8	4.21	4.63	5.06	5.51	6.1	7.11	9.53	11.7	12.52	15.67	18.23
5	2.96	3.47	3.99	4.36	4.72	5.23	5.75	6.21	7.6	9.4	13.16	15.79	17.4	21.64	31.75
6	3.52	4.08	4.54	4.88	5.36	5.81	6.2	7.01	9.56	11.7	16.59	20.2	24.98	27.25	30.08
7	3.9	4.58	5.01	5.46	5.86	6.27	6.83	7.79	11.31	14.11	19.3	23.18	29.456	24.09	26.71
8	4.44	5.14	5.64	6.07	6.4	6.89	7.53	8.93	12.92	15.87	21.65	24.49	22.73	24.82	26.41
9	5.08	6.02	6.22	6.57	6.93	7.36	8.42	10.25	14.37	17.58	23.45	21.8	23.79	25.14	26.8
10	5.56	6.7	7.02	7.11	7.46	7.94	9.24	11.02	16.04	19.33	25.41	22.86	24.62	26	27.86
11	6.09	6.99	7.66	7.78	8.09	8.75	9.98	12.17	17.19	20.88	27.1	24.4	26.05	27.48	29.19
12	6.66	7.81	8.21	8.6	8.57	9.2	10.78	13.12	18.49	22.39	28.47	25.81	27.57	29.06	31.26
13	6.8	8.54	9.02	9.31	9.23	9.81	11.43	14.15	20.22	23.7	29.9	27.53	28.88	30	32.64
14	7.3	9	9.88	9.78	9.68	10.47	12.46	15.07	21.17	24.63	32.02	29.23	30.87	32.57	34.83
15	8.11	9.68	10.55	10.61	10.21	11.03	13.03	16.05	21.61	26.98	28.79	30.87	32.53	34.14	36.38

Average attempts for the complete algorithm

The table shows that some instances of smaller size may have higher average attempt counts than instances of larger sizes. This is an effect of the limit variable that was mentioned in the previous section. Clearly, a higher limit value should've been chosen to give a more stable attempt count. Because of time

restrictions (10s timeout) lower values were used. Despite the unstable nature of this algorithm for $n < l$ (as previously described achieving 0 black hits gets more difficult) a respectable average is maintained.

Conclusion

In summary, finding an attempt efficient algorithm to solve Mastermind within a timeout was a task that required extreme care and caution. There were many ways of implementation, however, wastefulness when programming severely hindered the solver's performance. In the end, a strategic approach was used so that the code breaker could apply Swaszek's algorithm and further optimizations managed to extend the algorithms usage and generalize it for large instances.

References

1. Knuth, D. (1976). *The computer as Mastermind*. [online] Cs.uni.edu. Available at: http://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf [Accessed 20 Mar. 2019].
2. Cplusplus.com. (n.d.). *Reference - C++ Reference*. [online] Available at: http://www.cplusplus.com/reference/ [Accessed 20 Mar. 2019].
3. En.wikipedia.org. (n.d.). *Mastermind (board game)*. [online] Available at: https://en.wikipedia.org/wiki/Mastermind_(board_game) [Accessed 20 Mar. 2019].
4. Weisstein, E. (n.d.). *Mastermind -- from Wolfram MathWorld*. [online] Mathworld.wolfram.com. Available at: http://mathworld.wolfram.com/Mastermind.html [Accessed 20 Mar. 2019].