MASTER MIND REPORT

AGRIM MANCHANDA (CID: 01333764)

EE1 INTRODUCTION TO COMPUTING – SOFTWARE ENGINEERING I

MAX CATTAFI

Imperial College London

Table of Contents

1. Introduction:

The aim of this report is to explain the algorithms used to improve the game Master Mind and compare results of evaluative testing of possible algorithms that could have been used. The skeleton code provided to us was a working code maker and a code solver, albeit the latter being highly inefficient. Hence by evaluating and testing different relevant algorithms, it is possible to make the game more efficient.

In Master Mind, the user must provide an input for length (L) and sequence (N) of the code. The game then generates a random sequence of numbers defined by input length with numbers ranging from 0 to N. The code solver tries to find the sequence using minimum number of attempts and receives feedback on its guess. If the guess includes a number at the correct position the solver returns a black hit or if the guess includes a number at the wrong position the solver returns a white hit. By doing so, the game can eliminate large proportions of all possible solutions and make a better guess as its next move based on information received about black/white hits. The game ends when a sequence, of length L, has been found with all L black hits and no white hits.

2. Structs used in code

2.1 Code Maker

In this struct, the *init* and *generate_sequence* were provided in the skeleton code. However, *give_feedback* was required to be completed.

For *give_feedback*, the input to the function is the attempt vector and output will be black hits and white hits (which are inputs passed by reference). The function first copies both sequence and attempt into a temporary vector so that they can be used without needing to change the original vectors. As the black hits are more important, the function first checks for black hits by comparing the position of each element in the two temporary vectors. Once and if two same numbers are in the correct position, the function increments black hits which is initialised to zero at the beginning. The temporary vectors are assigned negative values so that they can't be checked once again for that hit. Once the black hits are processed, the function performs the same steps for white hits. The only difference this time is that it checks elements placed anywhere in the two lists rather than at the exact same position. By checking each attempt, this code returns values for black hits and white hits which are made use by function *learn* (explained later). Once the black hits equal the length and white hits are zero, the *main* stops the game and displays number of attempts required to find the secret code.

2.2 Code Solver

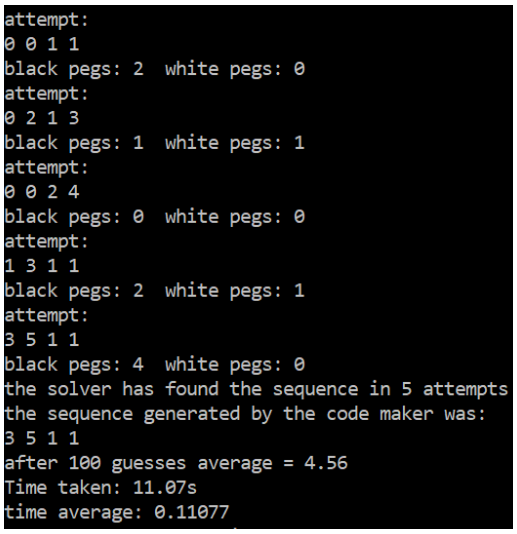
In code solver, there are three members of the struct: *init, create_attempt and learn.*

With *init*, there are extra member data of a vector and *PermutationList.* The purpose of both is to create a vector of vector integers defined as member that store the permutations for each combination of length and sequence, for example with L = 4, N = 6 the vector created will store from 0000 to 5555 (all possible attempts), hence a vector of size N^L. Permutations are created recursively in the function *PermutationList* that we can minimise computational power and effectively reduce processing time.

Note: It is important to notice the way we create the last possible attempt, in this case 5555, for each combination of length and sequence. Taking $6^4 = (1296)_{10}$, we look at the conversion of $(5555)_6$ into Base 10. The following is how we achieve this:

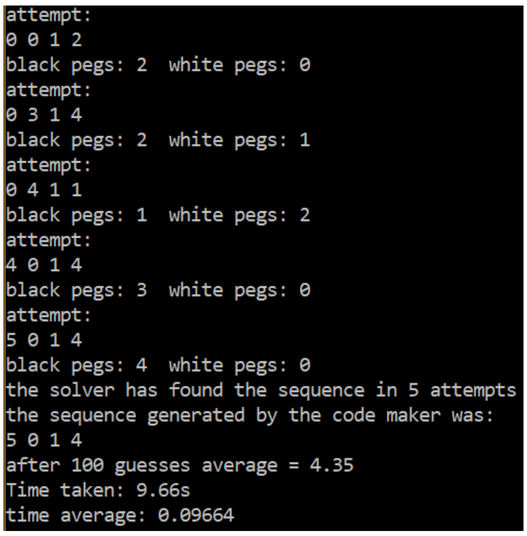
$(5 \times 6^3) + (5 \times 6^2) + (5 \times 6^1) + (5 \times 6^0) = 1295$, which is one lower than 1296. Hence, we can express $(5555)_6 = (1296)_{10} – (1)_{10}$. The same principle would follow for all combinations, in general being $N^L – 1$ for list of permutations. This is because the index starts at 0 instead of 1.

In *create_attempt,* there are three *if or else* statements that define which route is taken for next attempt. Each path is important and is dependent on many factors. The *if* statement is taken if the vector can be made in the beginning, which must be of size N^L. For almost all the cases that are tested in this report, this approach is taken. It is very important to have a strong first turn, with maximum black and white hits, so that the game can be solved in fewer attempts.



```
attempt:
0 0 1 1
black pegs: 2  white pegs: 0
attempt:
0 2 1 3
black pegs: 1  white pegs: 1
attempt:
0 0 2 4
black pegs: 0  white pegs: 0
attempt:
1 3 1 1
black pegs: 2  white pegs: 1
attempt:
3 5 1 1
black pegs: 4  white pegs: 0
the solver has found the sequence in 5 attempts
the sequence generated by the code maker was:
3 5 1 1
after 100 guesses average = 4.56
Time taken: 11.07s
time average: 0.11077
```

First move: AABB (0011)



```
attempt:
0 0 1 2
black pegs: 2  white pegs: 0
attempt:
0 3 1 4
black pegs: 2  white pegs: 1
attempt:
0 4 1 1
black pegs: 1  white pegs: 2
attempt:
4 0 1 4
black pegs: 3  white pegs: 0
attempt:
5 0 1 4
black pegs: 4  white pegs: 0
the solver has found the sequence in 5 attempts
the sequence generated by the code maker was:
5 0 1 4
after 100 guesses average = 4.35
Time taken: 9.66s
time average: 0.09664
```

First move: AABC (0012)

Figure 1(a): comparison of average attempts (L = 4, N = 6) for first move AABB and AABC.

After conducting extensive online research and assuming the case (L = 4, N = 6), the two possible methods for having a strong first move are: AABB or AABC (Arxiv.org, 2018). The research paper looks more closely at the latter as being the optimal first move. This is because the probability of getting a black or white hit is ½ and hence has a higher chance than AABB which only has 1/3 of a chance.

Using the written code, some basic evaluative testing was performed which also supports the results found in the research paper. Figure 1(a) shows that the second attempt with AABC produces a lower attempt average of 4.35 compared to 4.56 for AABB. The first move AABC also complements the algorithm chosen for this game. In the next paragraph, when the algorithm *Max Parts* is explained, it will become much more obvious that the latter choice is much better for fewer attempts overall.

After the first attempt, it is important that the algorithm *Max Parts* is initiated. Therefore, the algorithm has only been set for use if the number of possibilities has been reduced down to 2025 possibilities. This means for large inputs for example (6, 9), the algorithm would first use *learn* to reduce the number of possibilities significantly. Note: the number 2025 has been chosen because it is the first square number after 2000, which theoretically is when *Max Parts* works at its optimum level (Kooi B, 2005). *Max Parts* works by separating the possible solutions, after they have been eliminated by *learn,* into combinations of number of black/white hits that are provided by *give_feedback* function. These are arranged into “parts” which are combinations that match the feedback provided for the previous attempt. By using *Max Parts,* the game maximises the number of parts into which all the set of combinations are partitioned into. Figure 1(b) below shows the partitions that would be created depending on the first move played by the player.

First move	Partition elements
AAAA	4
AAAB	11
AABB	12
AABC	14

Figure 1(b): table showing partitions made with different first move.

Figure 1(b) shows that the best move for the player to win the next round is to choose the maximum partition possible. This allows them to maximise their chances of hitting a black hit so that the game is completed in minimum number of attempts. First move AABC has the most parts hence is chosen for the game algorithm.

In the code, we determine the number of partitions required by each possible solution by calling *MaxParts* function. This function returns the maximum partitions created for each of attempts as if it were the correct solution and the rest as not correct guesses. The partitions are incremented counting the number of elements that return the correct value for possible combinations, as shown in Figure 1(c):

Permutation List	0 black hit, 0 white hit	0 black hit, 1 white hit
00	5	3
01	0	3
02	3	1
03	2	2
Total Parts	10	9

Figure 1(c): table showing the black/white hits for each element in permutation list.

Figure 1(c) shows how the maximum parts determined, in this case for (0, 0) will be 10 parts that will be chosen.

After the partitions are received, the code updates the maximum partition, which is initialised to 0, to the next highest partition. After every attempt the number of partitions and the value of partitions decreases, but the code keeps picking maximum part to make the next attempt with more black/white hits. Eventually no partitions are left because the attempt made has all the correct elements in the right position hence the game ends.

The last *else* statement is one that will be taken in extremely rare cases – if the code is not able to create a vector large enough to support all the permutations. The code simply makes a random attempt which are then used by *give_feedback* to pass back results, it is a version of *Simple* algorithm.

In function *learn,* it first calls *give_learn_feedback*, which functions exactly in the same way as *give_feedback.* The purpose of this function is to pass the last attempt and all possible sequences to *give_feedback*, obtain the black and white hits and then eliminate the sequences that do not adhere to the feedback provided. For example, if a potential sequence returned zero black hits and zero white hits, then all possible combinations with those numbers are ignored and the remaining are stored into a new temporary vector. By following this process, we can quickly eliminate sequences that are not present, so the time required to find the correct sequence should shrink exponentially.

Overall all functions have been optimised to work with others to provide the best attempt information so that the game can be solved in minimum number of attempts. The main advantage of this approach is that it can execute quickly for small sequences, the first move helps to eliminate many combinations and the *Max Parts* algorithm provides a small average for number of attempts. The main disadvantage is that for extremely large vectors such as 9 x 9, the memory requirement to create the vector will be too large and hence it will not be able to execute the solver in such cases. Given this, it is still important to perform thorough experimental evaluative testing to judge the performance of used algorithm in comparison to other possible ones.

3. Evaluative Testing

It is important to perform relevant evaluative testing to compare the algorithm used against all other possible outcomes. With all algorithms having advantages and disadvantages, by performing evaluative testing, it is possible to narrow down possible options that could be used to solve the code. Whilst this report has already covered the testing for first attempt, this section also covers an additional test to determine its effect on average game length. This section will cover comparisons between *Max Parts and Simple.* The first test will also cover *MiniMax (using Knuth's 5 guess theory).*

3.1 Brief Explanation

MiniMax: The game starts by creating a vector with all possibilities and the first move being AABB (L=4, N = 6). The algorithm first removes all possibilities that do not match with the number of black hits/white hits in the correct sequence. For each attempt, the algorithm calculates the number of possible sequences that would be eliminated for each case of black/white hit. The player then plays the highest scored attempt from minimum scored attempts, where score is minimum number of possibilities eliminated from all possibilities vector. This process repeats until all possibilities are eliminated for black hits at which point the game has finished (Mastermind?, 2018).

Simple: all possible combinations are ordered, and the first combination is played as attempt (usually AAAA). Based on feedback on black/white hits, the next attempt is one that matches the combination of the previous attempt with the feedback provided. The game repeats these steps until the correct combination has been found. A version of this has been used as *else* statement under *create_attempt* (Kooi B, 2005).

3.2 Testing

In these series of tests, we will consider a range of possible combinations for length and sequence. Each test will be performed for 100 games hence an average will be taken. For the *MiniMax*, most of the data has been obtained from online sources (Mercury Webster, 2018).

First test: L = 4, N = 6:

Algorithm	**Average attempts**	**Expected attempts***	**Time per game (seconds)**	**Worst attempt**
Max Parts	4.39	4.37	0.09	6
MiniMax	-	4.46	-	5
Simple	5.45	5.77	0.04	9

Figure 2(a): results of average attempts for most common combination of 4, 6.

*Expected attempts taken from: Mercury Webster, 2018.

Second test: L = 5, N = 8:

Algorithm	Average attempts	Time per game (seconds)	Worst attempt
Max Parts	5.55	1.72	7
Simple	6.27	1.10	8

Figure 2(b): results of average attempts for combination of 5, 8.

Third test: L = 6, N = 9:

Algorithm	Average attempts	Time per game (seconds)	Worst attempt
Max Parts	6.39	3.49	7
Simple	6.72	0.87	8

Figure 2(c): results of average attempts for most common combination of 6, 9.

Fourth Test: L = 8, N = 8:

Algorithm	Average attempts	Time per game (seconds)	Worst attempt
Max Parts	7.27	56.05	9
Simple	8.34	27.25	10

Figure 2(d): results of average attempts for combination of 8, 12.

3.2 Average Attempts

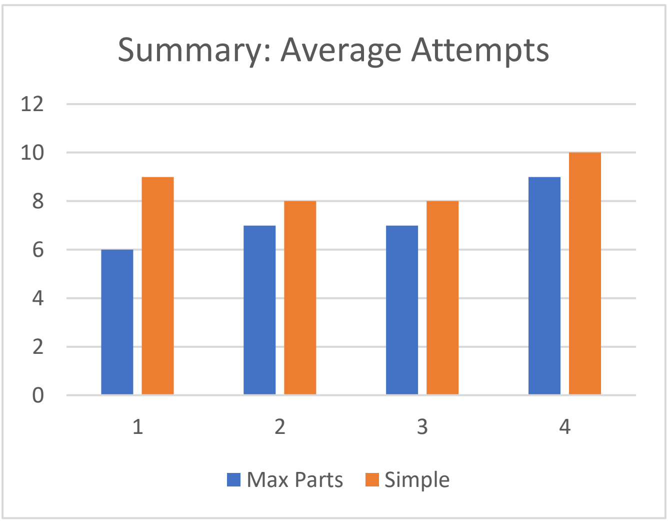


Figure 3(a): bar graph of average testing results.

Figure 3(a) summarises the above information for average attempts in all combinations. In general, as length and sequence increases, the number of attempts for both algorithms also increases which is as expected. Another trend is that *Simple* algorithm always has more number of attempts than *Max Parts* which is because the algorithm does not make partitions in the combinations like the latter. The most noticeable difference is (4,6) and (8,8) as *Simple* has a difference in attempt of approximately one. For first test of *Max Parts,* it is also just 0.02 attempts higher than the theoretical average hence the results are accurate. Using these information and extrapolation, it is possible to say that as the (L, N) increase, *Max Parts* algorithm will be better to solve the game in fewer attempts.

3.3 Time per game

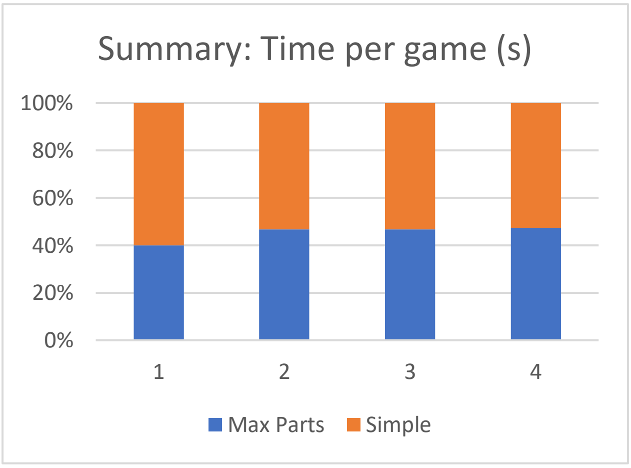


Figure 3(b): stacked column graph of time per game results.

Figure 3(b) shows a comparison of the time taken for each test as a percentage comparison between *Max Parts* and *Simple.* The general trend is that *Max Parts* takes a higher proportion of time compared to *Simple.* The most noticeable difference is with (6, 9) where *Max Parts* takes almost 4 times longer per game compared to *Simple.* The results for this are as expected because *Max Parts* searches through vector of partitions hence has a time average of O(n^2) compared to O(n) as generally observed in *Simple.*

3.4 Worst attempt

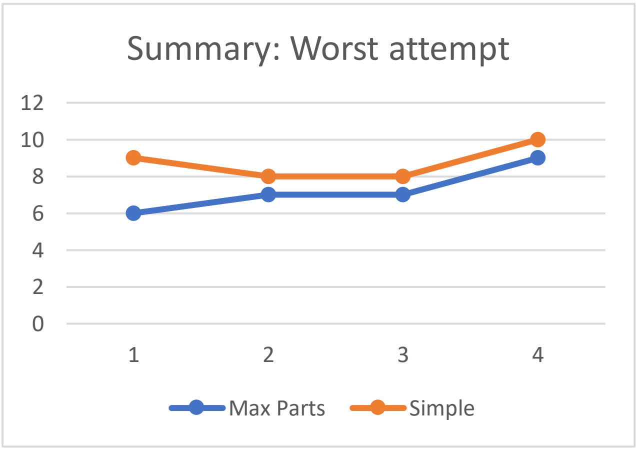


Figure 3(c): line graph of worst attempt results.

Figure 3(c) shows the general trend observed with worst attempt for both cases. The general trend is that worst attempts increase for both and their behaviour matches each other for all tests. The most noticeable difference is in (4,6) where *Max Parts* has worst attempt of 6, compared to *Simple* with worst attempt 9. This is as expected because *Simple* may have a case where the random attempt counteracts the feedback received (even though it tries to respect the black/white hits in sequence), so it must generate another attempt.

3.5 Game length

It has already been determined by some evaluative testing that the best first move for the game should be AABC. Section 2.2 already discusses the effect of choosing the same on the average number of attempts, but in this section, we investigate the effects on average game length.

First move	**Game length**						
	1	2	3	4	5	6	7
0011	1	10	55	610	614	6	0
0012	1	12	58	669	553	3	0

Figure 4: table comparing game lengths with different first move for (4, 6) on Max Parts.

Figure 4 shows the result of testing. Whilst there are differences for each game length, the most significant difference is for game length 4 and 5. To complete the game with 4 attempts, 0011 had a distribution of 610 games compared to 0012 requiring 669. This shows that the data is skewed, as the largest proportion of games won for 0012 are in 4 attempts, compared to 5 attempts for first move 0012. The skewed distribution goes to show that AABC complements the *Max Parts* algorithm so that most of its games are won in 4 attempts.

3.6 Memory Requirements

Whilst memory requirements are not a considerable issue for this project, it is still an area that should be tested to gauge the performance of the program. This test was performed for (8, 8) for *Max Parts* and *Simple.*

Max Parts: After the first attempt, the memory requirements were at their peak at 890 MB. This is as expected because the code will try to obtain the maximum partitions for the guesses so that the next attempt is much better than the first and hence the computational time is also very large. After the first attempt, the memory reduces significantly to an average of 250 MB until the solution is found.

Simple: After making a first random move, the memory requirements reach their peak at 990 MB. This is 100 MB larger than *Max Parts,* however this is as expected because a larger number of possible solution vector is created despite first move receiving some feedback. Generally, until a solution is not found, the memory requirements remain at an average of 400 MB.

4. Conclusion

This report evaluates the functions that are used to create and solve Master Mind. The functions are all described in detail and shown that they fit together to make the game easier to solve as the number of attempts progresses. The evaluative testing section performs testing on key aspects of the code to judge its performance. The results from each testing element show that *Max Parts* has fewer average attempts, lower worst attempts, better game length distribution and lower memory requirements. The biggest disadvantage is the computational time, which for this project is not as significant, but can have a real impact if this were developed for commercial purposes.

5. Future Improvements

Despite the code working well for lower combination of sequences, the biggest problem is that it is not scalable for high inputs. This is because the code is not able to allocate enough memory to create a permutation list for vectors that are too large and *int* is not able to numbers greater than 37 million itself. One way to make it more scalable is to use other algorithms such as using information from black/white pins to increment the index within each solution that is tried. This will make it work for 9 x 9 and greater. It is difficult to predict the scalability although it will work for very high values. This could also be combined with the current algorithm used as it can be used if *Max Parts* is not being implemented. Another possible method could be to use binary tree structure which will store every possible combination and eliminate trees as attempts are made, but the biggest foreseeable issue is with memory requirements, so again it might not be scalable.

6. References

Arxiv.org. (2018). [online] Available at: https://arxiv.org/pdf/1305.1010.pdf [Accessed 21 Mar. 2018].

Mastermind? (2018). *Clever ways to solve Mastermind?*. [online] Puzzling.stackexchange.com. Available at: https://puzzling.stackexchange.com/questions/546/clever-ways-to-solve-mastermind?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa [Accessed 21 Mar. 2018].

Kooi, B. (2005). *Yet Another Mastermind Strategy*. [ebook] Netherlands: University of Groningen, pp.13-20. Available at: https://core.ac.uk/download/pdf/148138063.pdf [Accessed 21 Mar. 2018].

Mercury.webster.edu. (2018). [online] Available at: http://mercury.webster.edu/aleshunas/Support%20Materials/Analysis/Dowelll%20-%20Mastermind%20v2-0.pdf [Accessed 22 Mar. 2018].