Mastermind

Name: Maya Golan

CID: 0129163

Word Count: 1860

Module Title: EE1-07 Software Engineering 1: Introduction to Computing

Assignment Title: Assignment 3 Spring 2019

Lecturer: Max Catafi

Date of Submission: 22nd March 2019

Imperial College London

Table of Contents

[2 Introduction 3](#_Toc4166092)

[3 Feedback 3](#_Toc4166093)

[4 Algorithm 3](#_Toc4166094)

[4.1 Five-Guess Algorithm 3](#_Toc4166095)

[4.2 Incremental Algorithm 4](#_Toc4166096)

[4.3 Incremental Algorithm with White Hits 5](#_Toc4166097)

[5 Optimisation 6](#_Toc4166098)

[6 Conclusion 9](#_Toc4166099)

[7 Bibliography 10](#_Toc4166100)

# Introduction

Mastermind is two-player code-breaking game invented in 1970 by Mordecai Meirowitz. The code-maker creates a sequence of *p* pegs with *c* possible repeatable colours, creating a sequence among possibilities. Throughout this paper, the colours and pegs will be noted as (*p, c*). The code-breaker attempts to solve the sequence by submitting their attempt sequence. For each attempt sequences’ peg that matches both the colour and the position of the solution sequence’s peg, a black hit is returned. For each attempt sequence’s peg that matches the colour but not the position of the solution sequence’s peg, a white hit is returned. The black and white hits returned are global, so the code-breaker does not know which black and white pegs correspond to their attempt sequence. The code-breaker wins the game once the returned number of black hits equals the number of pegs *p.* In this game the code-breaker has an upper limit of 5000 guesses to solve the sequence. For the classic game of (4, 6), Knuth [1] devised a Five-Guess Algorithm where the code-breaker can always succeed in five moves or less.

# Feedback

The function give\_feedback takes in an input as an attempt and provides feedback in terms of black and white hits. The give\_feedback function was implemented by creating a vector colour\_freq that acts as a frequency table of how many times a colour appears in the solution’s sequence. The indexes in the vector represent the colours available, while the elements represent the frequency of each colour. Each time a colour and position of the attempt sequence equates the solution sequence, the quantity of black hits is incremented and the corresponding index in the colour\_freq vector has an incremental deduction. For the remainder of the attempt sequence’s pegs with the correct colour in the wrong position, the remainder is equivalent to the quantity of white hits returned. This means that as the colour\_freq vector has a positive quantity of elements in any index, there exists correctly coloured pegs in the attempt sequence that are in the wrong position that should be returned as white hits. The incremental deduction in an index for both a white hit and a black hit ensure that when an index reaches 0, the quantity of white hits will not exceed the quantity of pegs available.

# Algorithm

## Five-Guess Algorithm

For each sequence attempted, the code-breaker attempts the guess which minimises the maximum number of eligible sequences (partitions) remaining [2]. The algorithm works as follows:

1. Create a pool of all the possible permutations.
2. The initial attempt consists of the first half of the pegs containing 0s, and the second half of the pegs containing 1s. If the number of pegs is odd, then the middle peg will consist of the colour 1. This step will minimise the partitions possible.
3. Obtain the feedback.
4. Using this feedback, the pool of possible permutations is compared to the initial attempt. All possible permutations which do not produce the same feedback with the initial attempt are filtered out the pool.
5. Compare each sequence in the newly filtered pool to all the sequences in the original pool. Create a frequency table for the feedbacks produced by each sequence. The feedback with the highest frequency is assigned as a score back to the possible sequence within the total pool. The sequence with the lowest score is selected as the next attempt. Preference is given to the sequence that is within the filtered pool. At the end of this loop, the smart\_guesses is the next guess.
6. Repeat step 3. until only one sequence remains.

This algorithm yielded good results in terms of a low quantity of attempts at an average of 4.92.

## Incremental Algorithm

It was quickly recognised that the Five-Guess Algorithm was effective at solving the sequence within low attempts, albeit outside the computational time limit. The issue regarding the computational time arose due to the computational time required to create the pool of possible sequences. Thus, a new algorithm had to be implemented that was able to approximate a sub-optimal solution within polynomial time [3]. This trade-off between achieving a better a better worse-case scenario and a lower polynomial time had to be taken into consideration. Therefore, a variant of the brute-force algorithm was implemented and modified to fit the requirements of Mastermind and was deemed the ‘Incremental Algorithm’. The Incremental Algorithm operates accordingly to a change in the black hits returned, provided by the function give\_feedback. Each index is incremented by 1 and the algorithm shifts onto the next index only when there is a change in a black hit. The algorithm works as follows:

1. The initial attempt returns a sequence of all 0s as long as the length *p*.
2. The first unconfirmed index is incremented until there is a change in the returned black hits. The following step depends on the change in black hits returned:
   1. If there is a positive change in the number of black hits returned, the index is proven to be the current element. The next index is incremented.
   2. If there is a negative change in the number of black hits returned, the index is proven to be the previous element. The index returns to the previous element and the next index is incremented.
   3. If there is no change in the number of black hits returned, the current index is incremented [4].
3. Repeat step 2. until the number of black hits returned equates the length of the sequence, *p*.

Table 1: Worked example for Incremental Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| **Attempt** | **Sequence: 2413** | **BW** | **Case** |
| 1 | 0000 | 00 | Initial |
| 2 | 1000 | 01 | C |
| 3 | 2000 | 10 | A |
| 4 | 2100 | 11 | C |
| 5 | 2200 | 10 | A |
| 6 | 2300 | 11 | C |
| 7 | 2400 | 20 | A |
| 8 | 2410 | 30 | A |
| 9 | 2411 | 30 | C |
| 10 | 2412 | 30 | C |
| 11 | 2413 | 40 | A |

## Incremental Algorithm with White Hits

It was further investigated that it was possible to observe a change in white hits additionally to black hits and to minimise the attempts required to solve the sequence in the Incremental Algorithm even further. Through using the white hits, it was possible to eliminate a colour from appearing incrementally in another peg. This was achieved as follows:

1. The initial attempt returns a sequence of all 0s as long as the length *p*. The number of black and white hits returned is considered the starting combination of black and white hits; the following combination of black and white hits will be compared to this step.
2. The following cases depend on the change in black and white hits returned:
   1. No change in black hits and white hits: increment the current peg to the next available colour.
   2. No change in black hits and an increase in white hits: increment the current peg to the next available colour.
   3. No change in black hits and a decrease in white hits: delete the current colour from the pool of possible colours. Increment the current peg to the next available colour.
   4. An increase in black hits and no change in white hits: this peg contains the correct colour. Increment the next peg to the next available colour.
   5. An increase in black hits and a decrease in white hits: this peg contains the correct colour. Increment the next peg to the next available colour. Delete the current colour from the pool of possible colours. However, if the working peg changed in the previous iteration, increment the current working peg.
   6. A decrease in black hits and an increase in white hits: revert the current peg to the previous colour. Increment the next peg to the next available colour. Move onto the next peg.
3. Repeat step 2. until the number of black hits returned equates the length of the sequence, *p*.

The following table, table 2, shows a worked example for the Incremental Algorithm with White Hits. The deleted possible colours within the pool are struck through and highlighted in red. As observed, in comparison to table 1, this algorithm yields a lower number of attempts for the same given solution.

Table 2: Worked example for Incremental Algorithm with White Hits

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attempt** | **Sequence: 2413** | **BW** | **Case** | **Pool of possible colours** |
| 1 | 0000 | 00 | Initial | 01234 |
| 2 | 1000 | 01 | B | 01234 |
| 3 | 2000 | 10 | E | 01~~2~~34 |
| 4 | 2100 | 11 | B | 01~~2~~34 |
| 5 | 2300 | 11 | A | 01~~2~~34 |
| 6 | 2400 | 20 | E | 01~~2~~3~~4~~ |
| 7 | 2410 | 30 | D | 01~~2~~3~~4~~ |
| 8 | 2411 | 30 | A | 01~~2~~3~~4~~ |
| 9 | 2413 | 40 | D | 01~~2~~3~~4~~ |

This method will give a maximum total number of attempts of 225 for (15, 15), as it is the length *p* multiplied by the number of colours *c*.

# Optimisation

Initially, the Five-Guess Algorithm was implemented for all combinations of colours and pegs. However, it was observed that the computation speed at which the Five-Guess Algorithm operated was not sufficient to meet the 10s-time limit imposed on the programme. This is observed in table 3. The cells highlighted in yellow represent the first instance at which the number of colours for that quantity of pegs exceeds the 10s-time limit. The average computational time was calculated by finding the average time taken to compute the Five-Guess Algorithm over 100 iterations.

Table 3: Average Computation Time (s) of Five-Guess Algorithm



Table 4: Number of Permutations



Thus, to find the limit at which the Five-Guess Algorithm operated within, the computation speed limit of the Five-Guess Algorithm was mapped onto all the possible permutations of the sequences, seen in table 4. It was thus observed that smallest permutation at which the computation speed limit failed was 50625 permutations. Therefore, it was decided that the Incremental Algorithm with White Hits will be utilised at the point where the possible permutations of the sequences surpass 50000. This was desired as the Five-Guess Algorithm was more efficient at achieving a lower average of attempts required to solve the sequence. Therefore, there was an incentive to maximise the range of the parameters, that being the length *p* and the number of possible colours *c*. The switching point between the algorithms is determined during the initialiser of the mm\_solver function.

Table 5: Average Attempts for Combined Algorithms

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | **Number of Colours** | | | | |
| **3** | **6** | **9** | **12** | **15** |
| **Number of Pegs** | **3** | 2.87 | 4.77 | 6.73 | 8.90 | 11.20 |
| **6** | 4.11 | 5.90 | 23.31 | 32.80 | 42.60 |
| **9** | 5.55 | 26.30 | 30.70 | 43.40 | 46.70 |
| **12** | 18.50 | 29.20 | 41.40 | 55.20 | 63.90 |
| **15** | 26.80 | 37.30 | 56.30 | 64.10 | 79.10 |

As observed in table 5, despite the significantly higher average attempts required to solve sequences with higher (*c, p*), the combined Five-Guess Algorithm and the Incremental Algorithm with White Hits is successful at code-breaking the sequence within the limited computational time.

# Conclusion

The most efficient algorithm at producing the lowest average attempts for any (*c, p*) is the Five-Guess Algorithm. However, there exists such a trade-off between the efficiency regarding the quantity of attempts and the polynomial time. Thus, in order to maximise the efficiency of each algorithm, the cross-over point between the two algorithms was investigated. This cross-over point was determined at the point at which the Five-Guess Algorithm was unable to compute the solution within the computational time limit.

# Bibliography

|  |  |
| --- | --- |
| [1] | D. E. Knuth, “The Computer as MasterMind,” *Journal of Recreational Mathematics,* vol. 9, no. 1, pp. 1-6, 1976. |
| [2] | J. Stevenson, “Mastermind With Many Colours,” University of Manchester School of Computer Science, Manchester, 2016. |
| [3] | J. K. a. E. Tardos, Algorithm Design, Boston: Pearson Education, Inc., 2006. |
| [4] | J. Dowell, “Defeating Mastermind,” 2014. [Online]. Available: http://mercury.webster.edu/aleshunas/Support%20Materials/Analysis/Dowelll%20-%20Mastermind%20v2-0.pdf. [Accessed 22 March 2019]. |