A Comparative Study of Brute Force and Decrease and Conquer Algorithm for Mastermind Puzzle Codebreaking

Felicia Sutandijo - 13520050
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: FeliciaSutandijo@gmail.com

***Abstract*—Mastermind is a codebreaking board game featuring codes made of six different colours. To win, the codebreaker needs to guess the exact code that has been determined by the codemaker with the help of black and white pegs indicating correct and close guesses. With 1,296 possible solutions and only eight to twelve chances to guess, it is almost impossible to win the game by luck. Therefore, this paper aims to compare the brute force and decrease and conquer algorithms designed to solve the Mastermind puzzle. The two algorithms will be evaluated by execution time, the average number of guesses, and the maximum number of guesses against all possible cases.**

Keywords—Mastermind; brute force; decrease and conquer

I. INTRODUCTION

Mastermind was invented by Mordecai Meirowitz in the year of 1970. The game entails logical thinking as one needs to guess a certain arrangement of coloured pegs through a series of deductions. A Mastermind game consists of a decoding board, code pegs of six different colours, and key pegs of two different colours (usually black and white) [1].

Fig. 1. A Mastermind Game

With 1,296 possible solutions in a classic Mastermind game, it is therefore impossible to guarantee a win by luck. Fortunately, several strategies could be implemented to break the code, two of which will be explored in this paper. Examples of these strategies include guessing systematically, guessing randomly, aiming to determine the set of colours used in the code first, or focusing on narrowing down the options of possible solutions [2], [3].

In this paper, two algorithms will be compared. The first is a brute force algorithm with initial steps dedicated to determining the set of colours used in the solution code, and the second is a decrease and conquer algorithm focusing on narrowing down the options of possible solutions. Both algorithms will be measured by execution time, average guesses, and maximum (worst case scenario) guesses.

II. THEORETICAL FRAMEWORK

A. Rules of the Mastermind Game

In this paper, the six different colours of code pegs will be represented by “1”, “2”, “3”, “4”, “5”, and “6”. The solution code has a length of four and is randomly generated. Blanks are not allowed, but repetitions of colour are allowed. For each round, the game will give feedback on the number of “black pegs” for exact matches (pegs that are correct in both colour and position) and the number of “white pegs” for close matches (pegs that are correct in colour but is wrongly positioned).

The rules of a classic Mastermind game are as follows.

To play, the codemaker first decides on a sequence of four code pegs, called the solution code. These code pegs come in six different colours and the codemaker is allowed to make any combinations of the six colours. Since the focus of this paper is codebreaking, the role of the codemaker will be carried out by a computer through random generation.

After the code is set, the codebreaker then attempts to replicate the code in eight to twelve rounds depending on the version of the game. With each guess, the codebreaker is given a hint using the key pegs; a black peg indicating a correct guess, and a white peg indicating a correct coloured peg, but in the wrong position.

Armed with the feedback key pegs, the codebreaker can then make a more informed guess in the next round. The codebreaker wins if he/she successfully uncovers the solution code (receiving four black pegs as feedback), and the codemaker wins if the codebreaker is out of turns [1].

For the purpose of measuring the efficiency of the algorithms applied, there will be no limit to how many guesses can be made in the following experiments.

B. Brute Force Algorithm

In computer science, the brute force algorithm is a general algorithm which can be utilized to solve almost all computational problems. This algorithm consists of systematically enumerating all solution candidates and checking whether each candidate satisfies the problem statement. Brute force algorithms are usually based on the problem statement itself or definitions and concepts regarding the problem [4].

While brute force algorithms are straightforward and easy to implement, the costs are proportional to the number of candidates raised. Therefore, problems which have a large set of possible solutions may require an equally large cost when brute force is attempted.

C. Heuristic

Brute force algorithms can be sped up by using heuristics. The word "heuristic" comes from the Greek word "eureka", which means to find or to discover [5]. A heuristic is any approach to problem-solving that employs a practical method neither guaranteed to be optimal nor proven mathematically but is sufficient for reaching short-term goals or approximations. Examples of heuristics are trial and error, rule of thumbs, or educated guesses [6].

Heuristics are not algorithms. Heuristics apply intuition or common sense to provide guidelines for an algorithm, which will limit the number of solution candidates raised. Among other things, heuristics may be used to prevent algorithms from exploring solution candidates that have been known to be impossible as a solution, or it may serve to provide initial information for the algorithm to work with [5].

D. Decrease and Conquer Algorithm

Decrease and conquer algorithms are algorithms which attempt to reduce a problem into two smaller sub-problems and only proceed to compute only one sub-problem. This approach can be seen as a modification to the more popular divide and conquer algorithms, which divide a problem into two sub-problems, processing both, and combining the solutions of each sub-problem [7].

Decrease and conquer algorithms only have two steps, which, as the name implies, are decrease and conquer. The first step, *decrease*, is when the algorithm reduces a problem into smaller sub-problems, usually into two sub-problems. The *conquer* step is where the algorithm processes only one sub-problem. There is no "combine" step in the decrease and conquer algorithm, as there is only one processed sub-problem [8].

There are three variants of decrease and conquer:

1. *Decrease by a constant*: a problem is reduced by a constant in each iteration. Examples include insertion sort and selection sort.
2. *Decrease by a constant factor*: a problem is reduced by a constant factor in each iteration. Examples include binary search and fake-coin problems.
3. *Decrease by a variable size*: a problem is reduced by different amounts in each iteration. Examples include Euclid's algorithm and selection by partition.

The algorithm designed in this paper will take the third variant, decrease by a variable size.

III. CODEBREAKING USING BRUTE FORCE ALGORITHM

A. Algorithm

Allowing repetition of colours but no blanks, a classic Mastermind game allows a total of $6^4 = 1{,}296$ combinations of solution codes to be played. An obvious strategy for discovering the randomly generated solution code is to simply try each one of the 1,296 possible combinations. This is called the brute force strategy.

With a classic brute force strategy, the best-case scenario is uncovering the solution code in one guess, while the worst-case scenario is uncovering the solution code in the $1{,}296^{th}$ guess. Obviously, trying out 1,296 different combinations is quite tedious if it were to be done by a human instead of a machine, not to mention the limit imposed by the game rules if it were to be played on a real Mastermind board game.

Taking the algorithm one step further, it is possible to use a heuristic to reduce the number of guesses needed to guarantee a win. To do so, the algorithm is divided into two steps.

The first six guesses of the algorithm are dedicated to discovering the colours and the frequency of each colour making up the solution code. For the first guess, the algorithm guesses a "1111" and gets feedback on the number of black pegs. If there are 0 black pegs, then there is no "1" in the solution code. If there is 1 black peg, then there is one "1" in the solution code, and so on. For the second guess, the algorithm guesses a "2222" to determine the number of "2"s in the solution code. The algorithm continues in this manner until all components of the solution code have been discovered (the algorithm has gotten a total of four black pegs in its feedback), or all six colours have been attempted. The number of white pegs during this first half of the algorithm will always be zero, as there will be no "wrongly-positioned" pegs since all pegs are identical.

The second half of the algorithm involves enumerating all possible permutations of the four components of the solution code discovered earlier. The enumeration yields $4! = 24$ total possible combinations to be tested. This is a reasonable number compared to the 1,296 in a classic brute force attempt. For combinations with repetition of colours, the list of solution candidates could be further reduced to exclude duplicates.

In summary, the brute force algorithm used to break the solution code is as follows:

1. Attempt “1111”, “2222”, and so on until “6666” or until all four components of the solution code have been discovered.
2. Enumerate all 24 possible permutations of the components of the solution code discovered by the previous step and test them one by one systematically until the algorithm receives four black pegs.

The resulting algorithm is intuitive and easy-to-understand, and has a worst-case scenario of 30 guesses, 6 from the first half and 24 from the second half. It is also worth noting that this algorithm performs better with codes that have more repetition of colours since the generated possible permutations contain more duplicates which could be eliminated.

B. A Documented Example

The following output of a Python program implementation illustrates the mechanisms of the brute force algorithm.

```
└─$ py3 main.py
Solution code: 1262
-----------------------------------
ATTEMPT #1: 1111
Black pegs (exact matches): 1
White pegs (close matches): 0
-----------------------------------
ATTEMPT #2: 2222
Black pegs (exact matches): 2
White pegs (close matches): 0
-----------------------------------
ATTEMPT #3: 3333
Black pegs (exact matches): 0
White pegs (close matches): 0
-----------------------------------
ATTEMPT #4: 4444
Black pegs (exact matches): 0
White pegs (close matches): 0
-----------------------------------
ATTEMPT #5: 5555
Black pegs (exact matches): 0
White pegs (close matches): 0
-----------------------------------
ATTEMPT #6: 6666
Black pegs (exact matches): 1
White pegs (close matches): 0
-----------------------------------
ATTEMPT #7: 2162
Black pegs (exact matches): 2
White pegs (close matches): 2
-----------------------------------
ATTEMPT #8: 2621
Black pegs (exact matches): 0
White pegs (close matches): 4
-----------------------------------
ATTEMPT #9: 2612
Black pegs (exact matches): 1
White pegs (close matches): 3
-----------------------------------
ATTEMPT #10: 1622
Black pegs (exact matches): 2
White pegs (close matches): 2
-----------------------------------
ATTEMPT #11: 6221
Black pegs (exact matches): 1
White pegs (close matches): 3
-----------------------------------
ATTEMPT #12: 1262
Black pegs (exact matches): 4
White pegs (close matches): 0
-----------------------------------
Code broken. Total number of guesses: 12
```

Fig. 2. Brute Force Example

C. Measured Statistics

The following output of a Python program measures the execution time, the average number of guesses, and the maximum (worst-case scenario) number of guesses of the brute force algorithm against all 1,296 possible solution/answer codes.

```
10 guess(es) for ['1', '1', '6', '6']
16 guess(es) for ['1', '1', '6', '5']
12 guess(es) for ['1', '1', '6', '4']
7 guess(es) for ['1', '1', '6', '3']
18 guess(es) for ['1', '1', '6', '2']
8 guess(es) for ['1', '1', '6', '1']
18 guess(es) for ['1', '1', '5', '6']
10 guess(es) for ['1', '1', '5', '5']
15 guess(es) for ['1', '1', '5', '4']
6 guess(es) for ['1', '1', '5', '3']
7 guess(es) for ['1', '1', '5', '2']
9 guess(es) for ['1', '1', '5', '1']
8 guess(es) for ['1', '1', '4', '6']
8 guess(es) for ['1', '1', '4', '5']
10 guess(es) for ['1', '1', '4', '4']
6 guess(es) for ['1', '1', '4', '3']
8 guess(es) for ['1', '1', '4', '2']
7 guess(es) for ['1', '1', '4', '1']
18 guess(es) for ['1', '1', '3', '6']
16 guess(es) for ['1', '1', '3', '5']
15 guess(es) for ['1', '1', '3', '4']
5 guess(es) for ['1', '1', '3', '3']
4 guess(es) for ['1', '1', '3', '2']
7 guess(es) for ['1', '1', '3', '1']
12 guess(es) for ['1', '1', '2', '6']
14 guess(es) for ['1', '1', '2', '5']
10 guess(es) for ['1', '1', '2', '4']
13 guess(es) for ['1', '1', '2', '3']
8 guess(es) for ['1', '1', '2', '2']
6 guess(es) for ['1', '1', '2', '1']
10 guess(es) for ['1', '1', '1', '6']
7 guess(es) for ['1', '1', '1', '5']
5 guess(es) for ['1', '1', '1', '4']
5 guess(es) for ['1', '1', '1', '3']
3 guess(es) for ['1', '1', '1', '2']
1 guess(es) for ['1', '1', '1', '1']

Total time to solve all possible solutions: 0.2082 seconds
Average guesses: 12.802
Worst case scenario: 30
```

Fig. 3. Brute Force Statistics

IV. Codebreaking Using Decrease and Conquer Algorithm

A. Algorithm

The decrease and conquer algorithm starts with a set *S* of all 1,296 possible combinations. This list will be the focus of the algorithm, as each guess will provide information to reduce the list of possible combinations.

The algorithm starts by picking a random guess from *S* to obtain initial information about the solution code. After feedback is received, the algorithm eliminates all possible combinations from *S* which are not consistent with the feedback received. The elimination process requires an evaluation of all the remaining possible combinations against the newly guessed combination and its received feedback.

This process is repeated until *S* only contains one possible combination (which is the solution), or when the algorithm receives feedback of four black pegs, which signifies a win.

In summary, the decrease and conquer algorithm used to break the solution code is as follows:

1. Create a set *S* enumerating all 1,296 possible combinations of all solution codes.
2. Make a random guess from *S*.
3. Use the feedback received to eliminate from *S* all combinations which are not consistent with the feedback.
4. Repeat from step 2 until there is only one possible combination of the solution code, or when feedback of four black pegs is received.

As an example, suppose the algorithm attempts to guess the solution code "3536". The following are the steps taken to uncover the solution code:

1. Set *S* enumerating all 1,296 possible solutions is created.
2. A random guess is picked from *S*, say "4353".
3. Feedback of 0 black pegs and 3 white pegs is received.
4. All possibilities that are inconsistent with the feedback is eliminated. For example, "1212" is eliminated, since if the solution was "1212", the guess "4353" would have gotten feedback of 0 black pegs and 0 white pegs, which is inconsistent with the feedback received earlier. The remaining candidates in *S* are now reduced to only 44 possibilities.
5. Another random guess from the remaining members of *S* is chosen, say "3546".
6. Feedback of 3 black pegs and 0 white pegs is received.
7. *S* is again reduced to only the possible candidates of the solution code, evaluated against the last guess "3536" and its feedback. *S* now only contains 5 possible candidates, "3544", "3545", "3541", "3542", and "3536".
8. From the five remaining candidates, another random guess, "3544", is picked.
9. Feedback of 2 black pegs and 0 white pegs is received.
10. This move leads to the elimination of all but one candidate in *S*, leaving only the solution "3536".
11. The solution is found in a total of 4 attempts, including the last guess "3536".

Considering the large number of initial possible combinations needed to be generated and kept track of (which is terribly hard for a human player to replicate), an effort to modify the algorithm to exclude its first step could be considered. Without initially enumerating all possibilities, a player could start by making a random guess. For the next guesses, the player should then evaluate another random guess against all the known previous guesses and see if it is consistent. If it is consistent, the player should make the guess. If it is not, the player should evaluate another random guess.

B. A Documented Example

The following output of a Python program implementation illustrates the mechanisms of the decrease and conquer algorithm.

```
└─$ py3 main.py
Solution code: 4165
-------------------------------------
ATTEMPT #1: 4645
Black pegs (exact matches): 2
White pegs (close matches): 1
-------------------------------------
ATTEMPT #2: 4653
Black pegs (exact matches): 1
White pegs (close matches): 2
-------------------------------------
ATTEMPT #3: 5646
Black pegs (exact matches): 0
White pegs (close matches): 3
-------------------------------------
ATTEMPT #4: 4265
Black pegs (exact matches): 3
White pegs (close matches): 0
-------------------------------------
ATTEMPT #5: 4565
Black pegs (exact matches): 3
White pegs (close matches): 0
-------------------------------------
ATTEMPT #6: 4165
Black pegs (exact matches): 4
White pegs (close matches): 0
-------------------------------------
Code broken. Total number of guesses: 6
```

Fig. 4. Decrease and Conquer Example

C. Measured Statistics

The following screenshot of an output of a Python program measures the execution time, the average number of guesses,

and the maximum (worst-case scenario) number of guesses of the decrease and conquer algorithm against all 1,296 possible solution/answer codes.

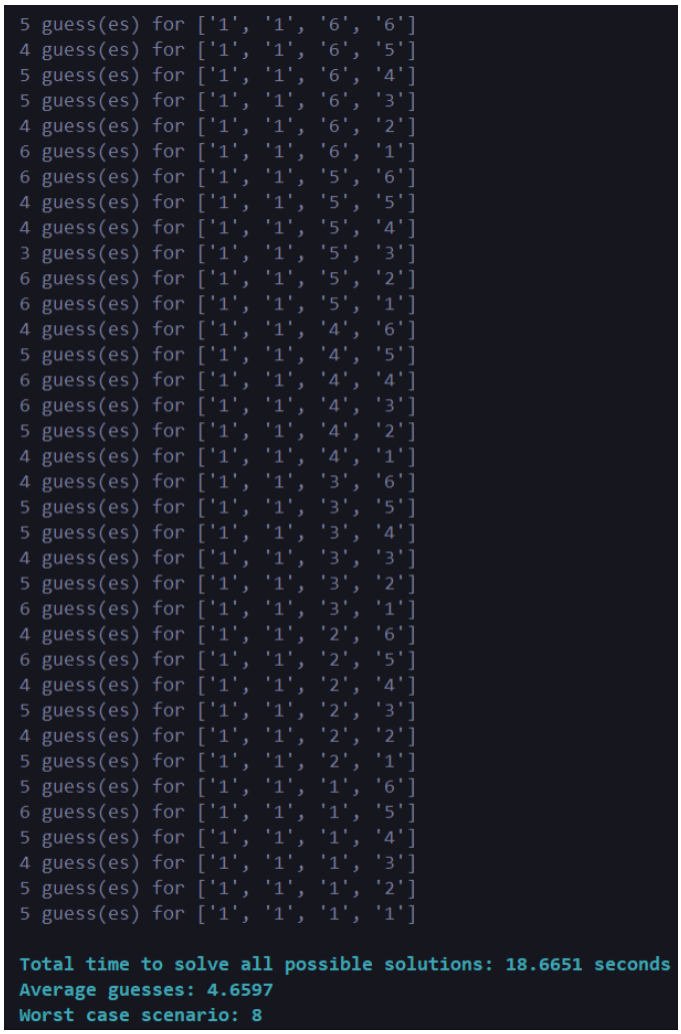

Fig. 5. Decrease and Conquer Statistics

V. Comparisons of Brute Force and Decrease and Conquer Algorithms

A. Overview

In terms of performance, the brute force and decrease and conquer algorithms are both measured by the total time needed to solve all possible 1,296 combinations, average guesses, and maximum (worst-case scenario) guesses needed to uncover the solution code. The comparisons are stated in Table I.

TABLE I. Comparison of Two Algorithms

	Algorithms	
	Brute Force	***Decrease and Conquer***
Total time (s)	0.2082	18.6651
Average guesses	12.8025	4.6597
Worst case	30	8

B. Time

Through this experiment, it can be seen that the decrease and conquer algorithm requires an average of almost 90 times more time to win a game than the brute force algorithm. Although the number of guesses needed for the decrease and conquer algorithm to determine the solution is considerably less than the brute force algorithm, it requires a careful selection of which guesses are still consistent with the previous guesses in each iteration. Much like in a human being, this "thinking" process takes time and thus the algorithm proves to be slower than a more general-approach brute force algorithm.

C. Number of Guesses

Optimized for a lower number of guesses, the decrease and conquer algorithm guarantees a win just within 8 guesses, with an average of 4.6597 guesses. On the other hand, brute force requires 30 guesses to be sure of success, with an average of 12.8025 guesses. Therefore, in a game with a limited number of guesses allowed, the brute force algorithm is not suitable.

D. Other Factors

The computer and the human mind are two different things. For a beginner human player, the simplicity of the brute force algorithm makes it easier to understand and implement in a game of Mastermind. The player simply needs to follow a methodical approach without plenty of thinking to be able to determine the code. On the other hand, the decrease and conquer algorithm requires a deeper understanding of the game and careful deliberation each time the player needs to make a guess. Therefore, a beginner player may find it easier to initially practice codebreaking with the brute force algorithm, before moving on to a more advanced decrease and conquer algorithm.

VI. Conclusion

The brute force and decrease and conquer algorithm are viable strategies to implement in a game of Mastermind, both with their own trade-offs. The brute force algorithm is well-suited for games without a maximum number of rounds and when one has reasons to believe that the solution code contains repetitions of colours. On the other hand, the decrease and conquer algorithm guarantees a win in less than eight rounds, thus is suitable for more complex games with a rule of a certain maximum number of rounds and for more advanced players.

Video Link at Youtube

A video explaining the contents of this paper may be viewed at https://youtu.be/vhfvBfylKWM.

Github Repository

The Python codes used in the experiments in this paper may be accessed from https://github.com/FelineJTD/Mastermind-Solver.

ACKNOWLEDGMENT

The author is incredibly grateful to Dr. Nur Ulfa Maulidevi, S.T., M.Sc. for her guidance and lessons throughout the semester, which insights have been invaluable to the completion of this project. Also, the author would like to thank a friend who has evaluated and reviewed the code for this paper.

REFERENCES

[1] "How to play Mastermind | Official Rules | UltraBoardGames." https://www.ultraboardgames.com/mastermind/game-rules.php (accessed May 23, 2022).

[2] Knuth. Donald, "The Computer as Master Mind" (PDF). "knuth-mastermind.pdf." Accessed: May 23, 2022. [Online]. Available: http://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf

[3] G. Ville, "An Optimal Mastermind (4,7) Strategy and More Results in the Expected Case." arXiv, May 05, 2013. Accessed: May 23, 2022. [Online]. Available: http://arxiv.org/abs/1305.1010

[4] R. Munir, "Algoritma Brute Force (2022) Bag1." Accessed: May 23, 2022. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf

[5] R. Munir, "Algoritma Brute Force (2022) Bag2." Accessed: May 23, 2022. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag2.pdf

[6] "Heuristics - Definition and examples — Conceptually." https://conceptually.org/concepts/heuristics (accessed May 23, 2022).

[7] R. Munir, "Algoritma Decrease and Conquer." Accessed: May 23, 2002. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Decrease-and-Conquer-2021-Bagian1.pdf

[8] "Decrease and Conquer - GeeksforGeeks." https://www.geeksforgeeks.org/decrease-and-conquer/ (accessed May 23, 2022).

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Mei 2022

Felicia Sutandijo
13520050