Mastermind Game with Genetic Algorithm

65070501039 B. Pongpon, 65070501055 U. Sorrawit, 65070501069 D. Kanitsorn,
65070501083 S. Panurut, 65070501092 B. Saranunt

Department of Computer Engineering, Faculty of Engineering
King Mongkut's University of Technology Thonburi, Bangkok, Thailand

ABSTRACT

The study explores effective strategies and conducts a detailed examination of applying a Genetic Algorithm to the Mastermind Game, building upon the framework outlined in [2]. Through hyperparameter fine-tuning and process refinement, the algorithm demonstrated the ability to solve the game in an average of 4.542 rounds for P(6,4) and 5.811 rounds for P(8,5).

A key focus was placed on creating a population with higher variance, allowing for a reduction in both the overall population size and eligibility while maintaining consistent performance. This was achieved by introducing a random population module, thereby reducing excessive memory consumption and runtime.

The hyperparameters, including the crossover rate, mutation rate, and others, were selected through a sweeping test. However, no clear trend or significant correlation with average rounds emerged from the analysis. Furthermore, the study provides an in-depth exploration of the Fitness Function mechanism, helping to theoretically identify bottlenecks affecting its performance. Despite this, the study does not propose a solution to address the bottleneck effect.

Keywords: Mastermind Game, Genetic Algorithm, crossover, mutation

1. INTRODUCTION

Mastermind is a popular 2-player code-breaking board game. Mastermind was invented by Mordecai Meirowitz in 1970s [5], from the behavior of the game, it also makes this game became one of the most famous challenges in the computer engineering field. One of the considerable approach is by genetic algorithm.

In this study, we aim to explores effective strategies and conducts a detailed study that is used to implement the **"Genetic Algorithm"** based on **Efficient Solutions for Mastermind Using Genetic Algorithms** [2] to solve the puzzle of the Mastermind game in the limited amount of guesses and the amount of time that the algorithm needs to generate the eligible guesses.

2. LITERATURE REVIEW

2.1 How to play Mastermind game [6]

In order to play the original "Mastermind Game" you need to use these 3 things which are

- Decoding Board to play the game
- Code pegs for the "code-maker" to set the secret code and
- Key pegs for the "code-breaker" to make their guess and for the "code-maker" to respond to the guess.

Before starting the game two players decide on the number of games to play which must be an **even number**. One player takes the role of the code-maker, while the other becomes the code-breaker. The code-maker selects the pattern of four code pegs, according to the prior agreement that duplicating code or blanking is permissible. The selected pattern will be hidden behind the shield, visible only to the code-maker.

Then the code-breaker tries to make the correct guesses within 12 rounds by placing the color pegs in the decoding board. The code-maker replies to each guess by placing 0-4 feedback pegs, the color key peg indicates the correct answer in the correct position and the white key peg indicates the correct answer in the wrong position.

In the case of duplicating colors in the guess, key pegs are awarded only if they correspond to the same number of the duplicate colors in the hidden code. For example, if the hidden code is red-red-blue-blue and the guess is red-red-red-blue the code maker awards the black pegs to the first two red and award nothing to the third red and black pegs for the blue.

The process of guessing and feedback is continued until the code-breaker guesses the correct answer or reaches the round limit of guess. Points

are traditionally earned when playing as a code-maker. The code-maker receives each point for every wrong guess made by the code-breaker and if the code-breaker is unable to guess the correct pattern in 12 rounds the code-maker will receive the extra point. The winner is determined by the player that has the most points after completing the agreed-upon number of games.

2.2 What is Genetic Algorithm [4]

Genetic Algorithm is a meta-heuristic inspired by the process of natural selection that belongs to the larger class of the Evolutionary Algorithm (EA).

Genetic algorithm use the "iterative process" and call each iteration as **"generation"**. Every generation contains the candidate solutions that are called **"individuals"** each individual has its own property that can be mutated to make them evolve toward the better solution.

A typical genetic algorithm requires 2 parts

1) Genetic Representation
2) Fitness Function

A genetic representation usually is an array of bits (bit string) due to their fixed size, which makes crossover operation easy to implement. Other representations can also be used, but crossover implementation would be more complex.

Fitness Function (known as Evaluation Function) is a value that measures how close a given solution is to solving the problem.

The evolution process starts by randomly generating the first generation to allow the entire range of possible solutions to be the member of the first generation, the number of generated individuals depends on the nature of the problem that algorithm need to solved but typically contains several hundreds or thousands solution, then the algorithm will use the fitness function to evaluate the fitness value of each individual in the generation. The individual that have more satisfied fitness value also have a higher chance to be selected as parent for the next generation.

3. METHODOLOGY

This study aims to develop a genetic algorithm to solve mastermind game. Mastermind game contains 2 modes of secret code which are 1) P(6,4) choose 4 colors from 6 colors 2) P(8,5) choose 5 colors from 8 colors, each secret code can have a duplicate color. We chose proper methods for our genetic algorithm—parent selection method, crossover method, mutation method—along with optimizing GA parameter—crossover rate, mutation rate, population size, etc.—

3.1 GA Methods Choosing & Parameter Tuning

In our genetic algorithm, we chose proper methods suited for our algorithm to find the best parameter tuning, to decrease the number of guesses and time usage.

To optimize our genetic algorithm. We chose the GA method and parameter-tuning guidelines from several recent studies.[2][3] and according to our test result.

3.2 Test & Experimental Design

To select GA methods and hyperparameters. We designed a test and chose the method and hyperparameter values based on a test result. We ran a test of 100 games 15 times to find the average round. In conclusion, we ran a total of 1,500 games, but the reason that we ran 100 games per time was to regenerate a random seed to avoid biases. In this study, we have constructed a test as follows.

3.2.1 Parent Selection Test:

In this test, we started using a parent selection method—random selection, tournament selection, proportional roulette wheel selection, and ranking selection—one by one and tested in the same setting environment. Then, we compared the average rounds and variance of each method and chose the lowest.

After the parent selection test, the "Tournament Selection" gives a satisfactory result. Then, we run a test to find the optimal tournament size that makes the lowest average rounds by using tournament sizes from 2, 5, 10, 15, and 20 respectively.

3.2.2 Crossover Ratio Test:

This test aims to find the optimal crossover ratio to reproduce the new generation. According to **"Efficient Solutions for Mastermind Using Genetic Algorithms"** [2], we decided to use two crossing-over methods—Single-Point Crossover and Two-Point Crossover—to increase the variability of the population. We tested a crossover ratio (Single-Point/Two-Point) from 0/100 to 100/0 and used the ratio that gives the overall lowest average rounds.

3.2.3 Mutation (Random Resetting) Rate Test:

In this test, we aim to find the optimal mutation rate of the Random Resetting method. The Random Resetting method helps to explore new possibilities of guesses and approaches to the correct code, but if

we have too high Mutation Rate, we might diverge from the correct answer. We tested the Mutation Rate from 0, 10, ..., 100% and used the rate that gives the satisfied average rounds.

3.3 Our Pseudo Code

Algorithm Le Adoga Genetic Algorithm

Set $T = 1$;
Play fixed initial guess g_1;
Get response X_1 and Y_1;
while $X_T \neq P$ **do**
  $T = T + 1$;
  Set $\hat{E_T} = \{\}$ and $h = 1$;
  initialize population;
  while $((G \leq maxGen$ And $|\hat{E_T}| \leq maxsize$ Or $|\hat{E_T}| = 0))$ **do**
    if $(G \geq 6$ And $|\hat{E_T}| = 0)$ **do**
      randomly generate half of the population and overwrite;
    Select Parent;
    Generate new population using crossover, mutation, inversion and swap;
    Calculate Fitness;
    Add eligible combinations to $\hat{E_T}$ (if not yet contained);
    $G = G + 1$;
  end while
  Play guess $g_T \in \hat{E_T}[0]$;
  get response X_T and Y_T;
end while

Note that: T is **T**urn that are guessing and **G** is **G**eneration that is being generated.

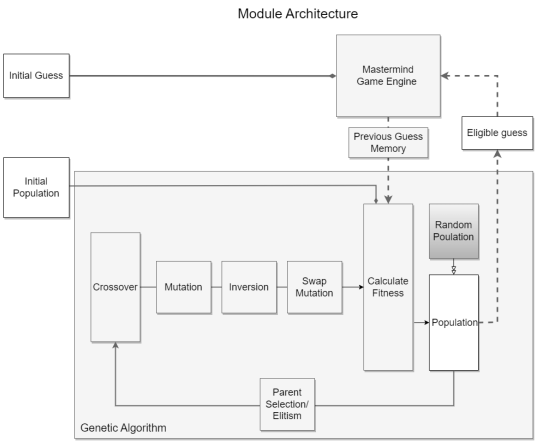

Fig. 1: Module architecture

3.4 Fitness Value

To compute the fitness value, we have incorporated the equation from the study of Lotte Berghman [2] into our work. The fundamental concept involves determining fitness by assessing the relative variances between the current guess and previous guesses. Employing the fitness function in this way serves to reduce the need for feedback calls, thereby minimizing the number of play rounds.

$$f(\mathbf{c}; T) = a\sum_{q=1}^{T} |X'_q(\mathbf{c}) - Xq| + b\sum_{q=1}^{T} |Y'_q(\mathbf{c}) - Yq|$$

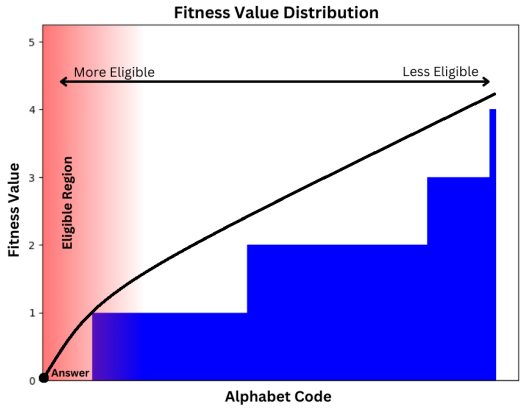

Fig. 2: Relation of Fitness Value and Alphabet Code

The fitness values for all elements in the case of using 4 colors from a set of 6 colors P(6,4) are illustrated in Fig. [2]. The leftmost element on the x-axis corresponds to the correct answer set (AAAA), while the elements to the right represent all possible permutations of the actual answer. In each iteration, the primary objective is to generate a population within the eligible region (fitness = 0). The guess with the highest fitness is then utilized as a baseline for subsequent calculations.

4. Result & Discussion

4.1 Fitness Function Behavioral Study

Upon observing the dynamics of the fitness function after each baseline update in every round, it becomes evident that the eligible region rapidly narrows, as shown in Fig. [3], while the guesses approach the real answer.

By observing the behavior of the Fitness function, we can identify a bottleneck effect in the algorithm. As illustrated in Fig. [3], the primary mechanism of this Fitness Function involves narrowing down the eligible region based on previous guesses. The blue bar represents fitness with less information—having larger eligible region. The orange area represents more informative guesses with smaller eligible region.

In some instances, the correct answer is added to the Eligible set ($\hat{E}$) but is not selected as one of the guesses, leading to an increase in the number of rounds in certain cases. From this observed behavior, the hypothesis emerges that the algorithm could converge more rapidly by generating as many possible codes as feasible based on previous feedback. We recommend further investigation following the approach outlined by Bestavros and Belal[1]. However, the study has not yet discovered an optimized solution for the Fitness function.

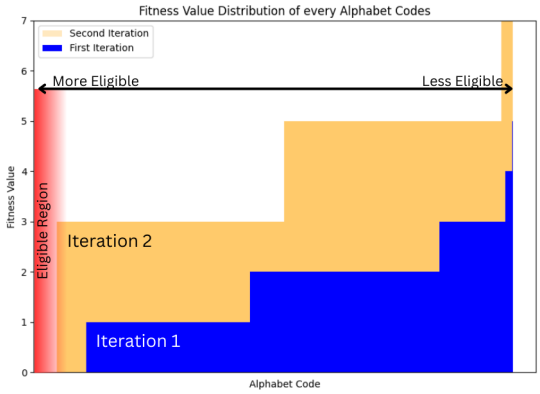

Fig. 3: Fitness value Converge to Real Answer

4.2 Parameter Tuning

In this section, we fine-tune each parameter based on careful observations of the genetic algorithm's behavior. This tuning process is integral to our subsequent performance test, where we assess the algorithm's efficiency.

4.2.1 Parent Selection Method:

According to the observation of parent selection methods—Random Selection, Tournament Selection, Proportional Roulette Wheel Selection and Ranking Selection. Tournament Selection given slightly higher performance of 4.60 average rounds on P(6,4) and 5.76 average rounds on P(8,5). Therefore, Tournament Selection is being used in our study.

Parent Selection	P(6,4)	P(8,5)
Random	4.76	5.80
Tournament	4.60	5.76
Roulette Wheel	4.70	5.87
Ranking	4.62	5.84

TABLE I: Average numbers of guesses for different parent selection

4.2.2 Tournament Size Method:

We selected the parent selection method as the Tournament selection for choosing individuals from a population to participate in the crossover and mutation processes. The critical advantage of tournament selection is its simplicity and versatility. It does not require sorting the population based on fitness, making it computationally efficient.

The experiment on the Tournament size to find the lowest average numbers of guesses with different values by using population size = 30, mutation rate = 10%, inversion rate = 3%, and permutation (swap mutation) rate = 2%

Tournament size	Average Guesses	Variances
0	4.59	0.01
2	4.58	0.01
5	4.61	0.00
10	4.62	0.00
15	4.59	0.01
20	4.64	0.01

TABLE II: Average numbers of guesses for different values of Tournament size on the P = 4 and N = 6

Tournament size	Average Guesses	Variances
0	5.86	0.01
2	5.83	0.01
5	5.85	0.00
10	5.80	0.01
15	5.86	0.01
20	5.85	0.01

TABLE III: Average numbers of guesses for different values of Tournament size on the P = 5 and N = 8

Referring to the Table [II], [III] , the averages and variances alone do not effectively anticipate data trends that might reduce the number of guesses. Therefore, our strategy prioritizes the lowest average number of guesses while allowing for some variances to encourage a diverse population.

In our experimental approach, we employed a tournament size of 2 to tackle our Mastermind game, resulting in averages of 4.58 and 5.83 with variances of 0.01 for scenarios P(6,4) and P(8,5), respectively. This choice aims to balance achieving a low average number of guesses and introducing enough diversity to enhance the genetic algorithm's performance.

4.2.3 Crossover Method:

In our study of genetic algorithms for the Mastermind game in C programming, we have observed that relying solely on either a one-point or

a two-point crossover tends to increase the number of guess rounds, introducing higher variability. To address this issue, we have experimented with a combination of both crossover methods, assigning specific probabilities to each.

Our hybrid approach involves using one-point and two-point crossovers, with a probability distribution that we fine-tuned for optimal performance. To better understand the impact of these probabilities, we generated a histogram to visualize the results.

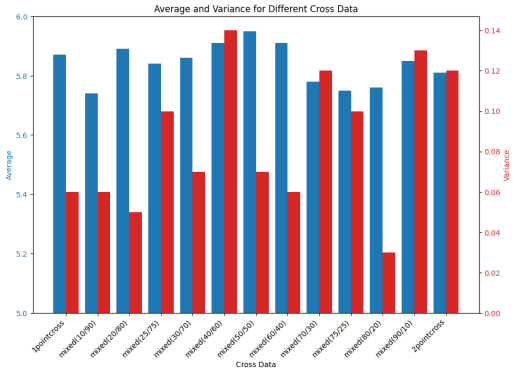

Fig. 4: Histogram of a combination of probability between one-point crossover and two-point crossover

The histogram revealed a probability distribution of 80 % for one-point crossover and 20 % for two-point crossover, resulting in a stable round with the lowest variance. This balanced combination shows promising results regarding efficiency and convergence toward the solution.

The histogram also highlighted another effective configuration: a 10 % probability for a one-point crossover and a 90 % probability for a two-point crossover. This setup yielded the lowest average guess round, indicating a quicker convergence toward the solution with consistent performance.

Based on these findings, we have adopted the 10/90 probability distribution for one/two-point crossover in our genetic algorithm for the Mastermind game. This choice balances stability and efficiency, enhancing the algorithm's overall performance and increasing its success rate in solving the Mastermind game efficiently.

4.2.4 Mutation Method:

Since we are solving a mastermind game, which has more than one different color. We cannot use Bit Flip Mutation.

We use Random Resetting to make the code diverse. It's the only mutation that can change the color of the code. While Inversion and Swap only shuffle the code. As shown in Table [IV].

Method	Possible Color
Inversion & Swap	AAB ABA BAA
Random Resetting	AAA AAB AAC
	AAD AAE AAF
	ABB ACB ADB
	AEB AFB BAB
	CAB DAB EAB
	FAB

TABLE IV: Possible Color of Input AAB with a different Mutation method (N=6)

On the other hand, Random Resetting could not be more efficient for exploring the code with the same color throughout. For example, if we have ABC and the solution is CBA. Random Resetting would only change the code to the other number, while Inversion or Swap would give us the correct answer more quickly.

The experiment on the tournament size = 2, 10/90 probability of one/two crossover to find the lowest average numbers of guesses with different values of mutation rate According from Table [V], we

Mutation rate	Average Guesses	Variances
0%	5.93	0.04
10%	5.77	0.04
20%	5.81	0.17
30%	5.92	0.08
40%	5.78	0.07
50%	5.90	0.09
60%	5.83	0.10
70%	5.86	0.08
80%	5.81	0.11
90%	5.79	0.11
100%	5.88	0.14

TABLE V: Average numbers of guesses for different values of Mutation rate on the P = 5 and N = 8

used Random Resetting Rate of 10 % in our genetic algorithm.

4.2.5 Elitism:

Elitism holds a key position in generic programming, significantly expediting the attainment of local optima by keeping the best fitness values for successive generations.

The experiment on the $|\hat{E}|$ = 20 to find the lowest average numbers of guesses with different values of Elitism rate by using tournament size = 2, population size = 30, mutation rate = 10%, inversion rate = 3%, and permutation rate = 2%

Elitism rate	Average Guesses	Variances
0%	4.65	0.01
10%	4.58	0.01
20%	4.60	0.01
30%	4.57	0.00
40%	4.58	0.01
50%	4.64	0.01

TABLE VI: Average numbers of guesses for different values of Elitism rate on the P = 4 and N = 6

Elitism rate	Average Guesses	Variances
0%	5.89	0.01
10%	5.82	0.01
20%	5.83	0.01
30%	5.82	0.01
40%	5.83	0.01
50%	4.68	0.01

TABLE VII: Average numbers of guesses for different values of Elitism rate on the P = 5 and N = 8

Referencing Table [VI], [VII] averages, and variances may not effectively capture data trends that could reduce the number of guesses in our Mastermind game.

In our experiment, we prioritize the lowest average guesses, specifically choosing $\hat{E}$ = 20 with an elitism rate of 30%. This configuration results in an average number of guesses of 5.82 and a variance of 0.01. These findings underscore the effectiveness of employing elitism to enhance the algorithm's performance in the Mastermind game.

4.2.6 Population & Generation:

Normally, a larger population size is expected to provide greater diversity in the genetic code, leading to a reduction in the average guess round. Delinquent to this expectation, our tests have revealed that the population size only marginalizes the average guess round.

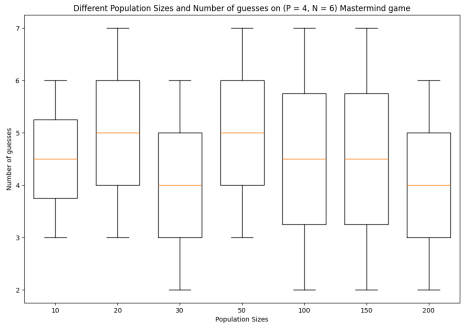

Fig. 5: The average and standard deviation for the number of guesses on different population sizes on P = 4 and N = 6

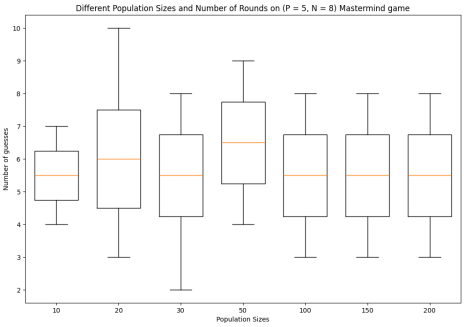

Fig. 6: The average and standard deviation for the number of guesses on different population sizes on P = 5 and N = 8

4.2.7 Improve Execution time:

Our goal extends beyond minimizing the number of guess rounds; we also consider execution time. It was observed that the population size significantly influences the execution time for both scenarios, P(6,4) and P(8,5). However, the impact of population size in P(6,4) appears to be less pronounced compared to its effect in P(5,8). As shown in Fig. [7]

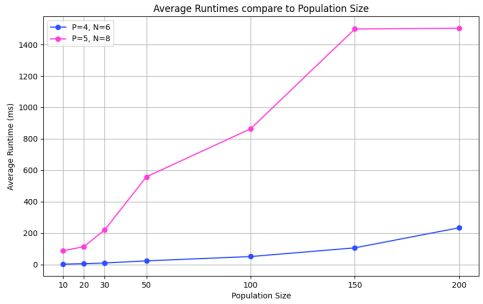

Fig. 7: Average runtime compared to Population size

4.3 Best Tuning and Comparative Result

Parameter	Value	Result
Tournament Size	2	
One-point Crossover Rate	10 %	
Two-point Crossover Rate	90 %	P(6,4)
Mutation Rate	2 %	4.542 avg Round
Permutation Rate	2 %	10.2 ms
Inversion Rate	3 %	
$\hat{E}$ Size	20	
Population Size	30	P(8,5)
Maximum Generation	20	5.811 avg Round
Elitism Population Rate	30 %$\hat{E}$	561 ms
Black Weight	1	
White Weight	1	

TABLE VIII: Best tuning Parameter

4.3.1 P = 4, N = 6:

According to experimental observation in section 4.2, The best tuning for Our algorithm is shown in this table. We obtain 4.542 average rounds, min round at 1, and max round at 7 from this tuning, which does not differ from the previous study—Table [IX][2]. However, Our algorithm tuning consumed significantly less memory usage as we decreased our population size and max generation.

Algorithm	Average Guesses	Max Guesses
Knuth	4.478	6
Norvig	4.47	6
Chen et al. (2007)	4385	6
Kooi	4.373	6
Irving	4.369	6
Neuwirth	4.3642	6
Chen et al.(2007)	4.359	6
Koyama and Lai	4.34	5
Rosu	4.34	-
Bestavos and Belal(MaxMin)	3.86*	-
Bestavos and Belal(MaxEnt)	3.835*	-
Lotte Berghman	4.39	7
Our Tuning	4.542	7

TABLE IX: Results for full enumeration with P = 4 and N = 6

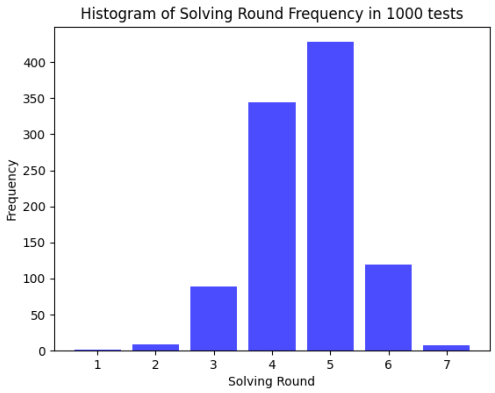

Fig. 8: Histogram of Solving Frequency in 1000 P(4,6) tests

4.3.2 P = 5, N = 8:

In this section, our tuning is no different from P(6,4). We've got an average of 5.811 rounds, min round at 2 and max round at 9. Compared to other meta-heuristic algorithm, the performance is noticeably less than other meta-heuristic algorithm.[2]

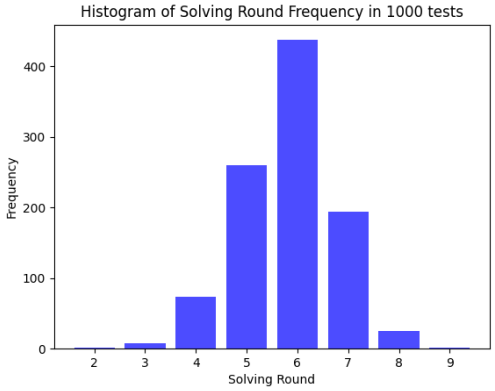

Fig. 9: Histogram of Solving Frequency in 1000 P(5,8) tests

Algorithm	Average Guesses
Rosu	5.88
Kalisker and Camens	6.39
Bento et al	6.866
Our Tuning	5.811

TABLE X: Results for full enumeration with P = 5 and N = 8

5. Conclusion

In our experiment, we explored novel approaches to play Mastermind with a focus on minimizing execution time and achieving a low average number of guesses. The algorithm we employed was inspired by Lotte Berghman's work [2]. Our strategy centers on generating a population with high variability, resulting in a significant reduction in both population (30) and eligible size (20). We achieved an average of under 4.542 guesses for P(6,4) and under 5.811 guesses with a 100% win rate. Although the average guess round is no differ to other meta-heuristic algorithms for P(6,4), our approach outperforms them as the color and position increase in P(8,5).

Hyperparameters were fine-tuned through sweeping experiments, selecting values that yielded the best average round. The mutation, permutation, and inversion rates were set at 2%, 2%, and 3%, respectively. Two crossing-over processes, One-point and Two-point Crossover, were employed in a 10:90 ratio. Tournament selection with a size of 2 was used for parent selection, with a maximum of 20 generations . Our observations revealed no significant trends related to the average round.

The study delves into the behavior of the fitness function, identifying a potential bottleneck. Due to the convergent nature of the fitness function, informative guesses must be input, even when the correct answer is already within the eligible set. This results in one or two additional solving rounds. Although a solution for this case is not implemented in our study, the work of Bestavros and Belal [1] could potentially optimize such situations in the future.

6. References

[1] A. & Belal Bestavros. "MasterMind: A game of diagnosis strategies". In: *Bulletin of the Faculty of Engineering, Alexandria University* (1986).

[2] et al. Lotte Berghman. *Efficient solutions for Mastermind using genetic algorithms*. ORSTAT, 2000.

[3] Juan Merelo Guervós, Pedro Castillo, and Victor Rivas Santos. "Finding a needle in a haystack using hints and evolutionary computation: The case of evolutionary MasterMind". In: *Appl. Soft Comput.* 6 (Jan. 2006), pp. 170–179. DOI: 10.1016/j.asoc.2004.09.003.

[4] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Mit Press, 1996.

[5] Toby Nelson. *A Brief History of the Master Mind Board Game*. Sept. 2015. URL: https://web.archive.org/web/20150906044819/http://www.tnelson.demon.co.uk/mastermind/history.html.

[6] Wikipedia. *Mastermind (board game)*. Feb. 2020. URL: https://en.wikipedia.org/wiki/Mastermind_(board_game).

APPENDIX

Code for Mastermind game with genetic algorithm

The source code for this project is available on GitHub
https://github.com/sokungz01/mastermindGeneticAlgorithm
Last updated on 11:19 PM on 19 Dec 2023
Specified run on gcc -std=c17 compiler

Additionally, a live demonstration of the program is available on Replit. To interact with the program and witness its functionality
https://replit.com/@sokungz01/mastermindshalarderJINGJANGMAK Last updated on 11:19 PM on 19 Dec 2023