CS209 - Artificial Intelligence: Course Project

Siddharth Singh
Department of Computer Science
220010055@iitdh.ac.in

Krutay Upadhyay
Department of Computer Science
220010029@iitdh.ac.in

Subhash Chandra Ponnam
Department of Computer Science
220120024@iitdh.ac.in

Rahul Anand
Department of Computer Science
220120016@iitdh.ac.in

Amogh R
Department of Computer Science
220010005@iitdh.ac.in

R Amogh
Department of Computer Science
220010046@iitdh.ac.in

Abstract

Mastermind's code-breaking challenge presents a compelling opportunity for AI and machine learning algorithms to excel. While initially appearing straightforward, its computational complexity offers opportunities for developing algorithms that might have far reaching consequences. Since the general game of Mastermind is essentially a search problem, a variety of other problems can be modeled on similar terms and solved using the algorithms we develop. Beyond the game realm, strategies devised for Mastermind hold potential for real-world applications like cryptography, logistics optimization, and uncertain decision-making. In this report, we explore a class of algorithms called genetic algorithms, and how they compare against deterministic algorithms when applied on the problem of Mastermind.

1 Introduction

Mastermind is a strategic two-player game where one player creates a secret code of colored pegs, and the other player attempts to guess the code within a limited number of tries. Using feedback from each guess, consisting of the number of correct colors and their positions, the codebreaker refines subsequent guesses until uncovering the exact sequence. The classic Mastermind consisting of six colours and four slots has been studied extensively, and in this report we try to solve a more generalized version of it with a higher number of colours and slots.

Input: A hidden combination of four slots, each chosen from a pool of six colors.
Output: Number of guesses made to find the secret combination

Genetic algorithms are optimization algorithms inspired by natural selection and evolution. They start with a population of potential solutions represented as individuals, undergo selection based on fitness, crossover to combine genetic information, and mutation to introduce diversity. Through iterative generations, individuals evolve towards better solutions, exploring the solution space efficiently. Genetic algorithms excel in complex optimization problems with large search spaces as in the case with Mastermind.

2 Known Methods

2.1 Donald Knuth's Mastermind

Donald Knuth's algorithm for solving the classic (6,4) Mastermind, published in 1977, showed that the codemaker can always win in five moves or fewer. The algorithm works on a mini-max approach.

The "best" initial guess has been shown to be (1, 1, 2, 2), and it was shown that other initial guesses might not provide the guarantee of 5 or fewer guesses for finishing the game. Each step in the algorithm involves eliminating the codes that are not consistent with the ones that have been guessed, from the set of possible 1296 codes (say S), thereby reducing the search space. The next step involves providing a "score" to *all of the possible codes*. The score given to a code the minimum number of possibilities it might eliminate from S. This subroutine requires $1296 \times |S_t|$ number of iterations, where t is the current number of guesses, and $|S_t|$ denotes the reduced set after t guesses. The mini-max approach comes into play when the code with the maximum score is used as our next guess.

An interesting aspect of Knuth's algorithm is that it does not require the guesses to be consistent with the previous guesses. The algorithm works on the basis of maximizing reduction in the set size, and can guess inconsistent codes to cut down on the search space. However, due to the large number of possible codes and the notion of multiple passes through the set, Knuth's algorithm performs well in theory to minimize number of guesses, but does not perform well in terms of time. Hence, we look for a better approach that can balance optimality of guesses along with time.

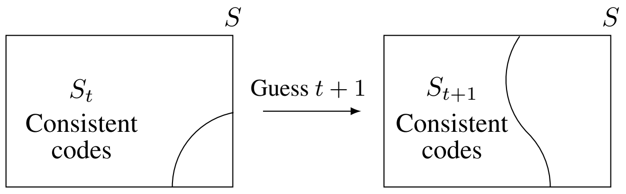


Figure 1: Reducing consistent combination search space

2.2 The MaxMin Information Strategy[1]

Formulation:
The analyst, starting with no information about the code, must consider all the possible R^N patterns. After the i^{th} round, $i = 1, 2, \ldots$, the solution pool reduces to $P_i \subset P_{i-1}$. Let $\{G_i^1, G_i^2, \ldots, G_i^k, \ldots\} = P_{i-1}$ denote the set of possible i^{th} guesses. If guess G_i^k is proposed at the i^{th} round, the new pool P_i consists of all those members of pool P_{i-1} which have not been eliminated by the i^{th} response from the setter. That is, pool P_i is the set of all patterns that might still be the code. Therefore, the goal of the analyst is to reduce some pool, say P_m, to a single element and the game ends when such a pool is found. In this case, we say that the game has ended after m rounds.

Optimal Guesses and Lookahead Strategy:
At any step, i, the analyst's goal is to minimize the number of guesses, m, required to break the code. Therefore, an L-step lookahead strategy should select the guess that optimizes the pool sizes L-step further. For example, a two-step lookahead strategy should select the guess G_i^k that optimizes the pool sizes of the $(i+2)^{th}$ step. In choosing the value of L (the lookahead level), the analyst is actually determining the number of levels to be considered from a large tree describing the course of the game for any guess and for any possible response.

Implementation:
The MaxMin strategy selects guesses to maximize the minimum information gained in the worst-case scenario. It aims to minimize the largest number of guesses needed to find the code by optimizing each guess for the smallest possible pool size. While conservative, it provides an upper bound on the required guesses, ensuring the code can be broken within or fewer guesses.

Initial Guess: Based on the lookahead strategy, the initial guess which leads to the maximum information gain on average is (1, 1, 2, 2).

Observations

Table 1: Performance Metrics

Number of guesses	Average time taken (seconds)	Average combinations evaluated
4.41	0.191	All

Conclusion
It is observed that if we select the pool which maximizes the minimum information gain for every subsequent guess, the number of guess taken on average is cut down by a monumental 0.68 while the time remains nearly the same.
This algorithm takes the most amount of time amongst all the other algorithms implemented but it also takes the least number of guesses to guess the secret code (on average). Hence, there is a trade off between the average time and the average number of guesses taken.
The minor downside is that the upper bound is very loose so it could potentially take up to 6 guesses in the worst case but also take 1 or 2 guesses in the best cases. Hence, the variance is somewhat large.
If time is not a problem and the secret code is to be guessed in the minimum number of guesses, then statistically speaking, the Bestavros and Belal algorithm should be used.

2.3 Berghman's Algorithm - Eligible Set[2]

This algorithm is designed for fast run-times and a low expected number of guesses. It performs comparably to other meta-heuristics for standard settings and outperforms existing algorithms for more complex scenarios with more colors and positions. This algorithm uses the concept of an eligible subset of the population. An eligible set consists of the offsprings which have a greater fitness than the threshold.

2.3.1 Implementation of Algorithm

Generation of new population set: After each guess, a new population set is created using codes from the previous population as parents. This process involves crossover, mutation, permutation, and inversion operations to explore the solution space efficiently.

Calculating fitness function: The fitness function evaluates potential solutions based on correct positions and colors compared to the secret code, guiding the selection and evolution process. An additional term is added to the fitness function to improve convergence. A new guess is made after traversing through all the generations randomly from the eligible set.

$$f(c;i) = a\left(\sum_{q=1}^{i} |X_q(c) - X_q|\right) + \sum_{q=1}^{i} |Y_q(c) - Y_q| + bP(i-1) \tag{1}$$

The fitness is added with an extra term : $P(i-1)$ where P is the code length and i is the length of the guess list. the value of b was set to 2 as suggested in the paper.

A random code is selected from the eligible set. If it correctly guesses the code, the game is terminated. Otherwise, the guess is appended to the guess list, and the process continues and terminates after reaching a certain limit of the guess list.

Observation and Results : The algorithm yields similar average guesses but excels in computational time. Convergence issues were encountered due to insufficient generations or poor population generation. However, the algorithm successfully found solutions even for complex scenarios, such as 20 colors and 5 slots, within 12 guesses and less computational time.

2.4 Genetic Mastermind[3]

The Genetic algorithm for Mastermind is a *stepwise optimal* algorithm. This means that the algorithm works to minimize the *distance to a consistent combination* and prompts the agent to play as soon as a consistent combination has been found. Arbitrary combinations are chosen from a population comprising of a specific number of random combinations. If no combination in that population is consistent (i.e. has fitness equal to 0), it is required to regenerate the population by evolving the combinations directly and this process is repeated across multiple generations until a consistent combination is obtained. If this combination is the secret code the algorithm terminates and the agent is successful. Otherwise, the agent plays that combination and the algorithm continues to find the next consistent combination. Genetic operators including *transposition*, *circular mutation*, and *crossover* were used to evolve combinations and add diversity to the population. Further, a *rank and elite based selection scheme* is performed on the population to remove 50% of the worst combinations. The general problem this algorithm faces is to maintain diversity which was (partially) resolved by *hypermutation*, in which the entire population was killed and new individuals were generated if 15 successive generations passed without finding a consistent combination.

2.4.1 Observations and Results

The average number of guesses and evaluated combinations were obtained in the cases of normal and Super Mastermind as shown below.

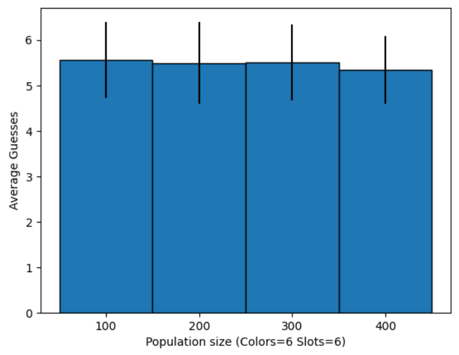
Algorithm	Average Guesses	Average Combinations Evaluated
GenMM (1)	4.67	316
GenMM(2)	4.60	307
Knuth	4.76	All

Table 2: Performance Metrics: GenMM(1) - Random first guess, GenMM(2) - First guess (1,1,2,2), Population Size - 100

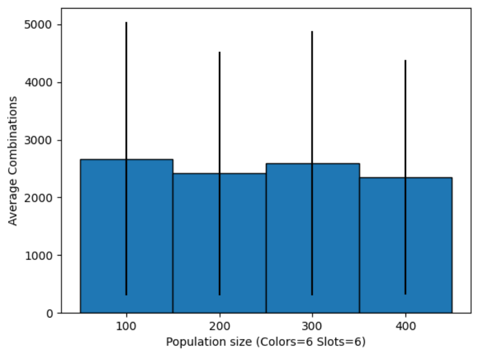
Algorithm	Average Guesses	Average Combinations Evaluated
GenMM	5.89	2052

Table 3: Performance Metrics: Super Mastermind, Population Size - 400

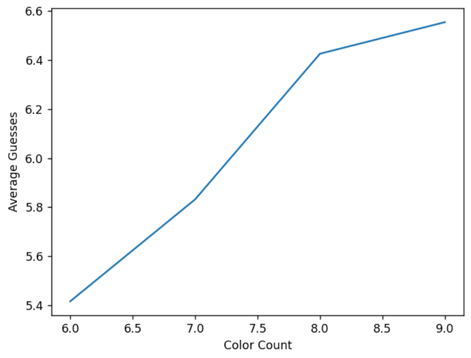
Further, it was observed that while the average guesses and evaluated combinations remained more or less the same on increasing population size, their standard deviations tended to decrease as shown in figures (a) and (b). Interesting observations were also noted on increasing problem/search space size (which is achieved by increasing color count) as shown in figures (c) and (d). The average number of guesses showed no particular trend but increased with problem size. However, the average evaluated combinations increased linearly as the search space size increased exponentially.



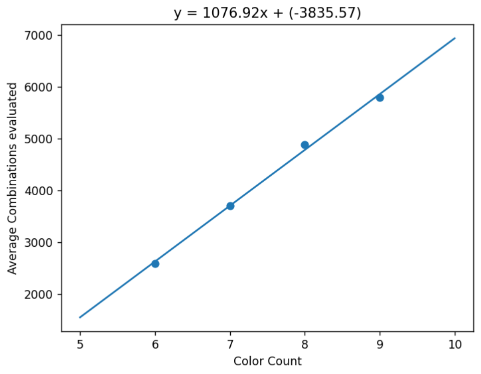
(a) Average Guess vs Population Size



(b) Average Combinations Evaluated vs Population Size



(a) Average Guess vs Color Count



(b) Average Combinations Evaluated vs Color Count

2.5 Demo

Here is the link to a web-based version of the game, built using the Genetic Mastermind algorithm. You can try it out yourself to assess the efficiency of this algorithm, as well as the time it takes and the number of guesses it makes to decode the secret combination.

3 Practical Applications: Fuzzy Hash Reversal

A closer look into the algorithms implemented, and Mastermind itself, show that this problem is essentially an informed search problem where the search space can be imagined to be a set of strings. Looking into practical aspects of these solutions, we come to the realm of cryptography, where reversing a hash to its parent string has some similarities to the problem at hand. The classical hash algorithms (such as MD5, SHA, etc.) follow the avalanche effect where a small change in the input parent string causes a large change in the hash. Moreover, there is no notion of 'similarity' in the hashes provided by such algorithms.

In this report, we restrict ourselves to a special set of cryptographic hash algorithms called fuzzy hash algorithms, which provide similar hashes for similar inputs, and included is a notion of a 'similarity score' or a 'difference score'. Formalizing the problem, we can state it as follows:

Q. Given a fuzzy hashing function F, a difference function D_F, and a hash of a string h, can we find a string s such that $F(s) = h$?

3.1 Proof of Concept

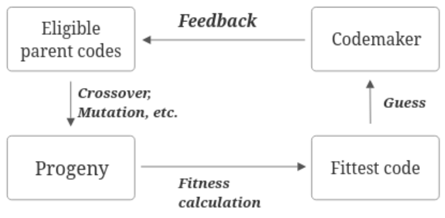
In our approach we model this problem using the TLSH hash algorithm and its difference function, and a string length fixed at 50. **Closeness** c_{ij} of two *known* strings w_i, w_j is measured by the number of character matches when compared index wise, and if a string is unknown, but the hashes are known then the difference function $D_F(h_i, h_j)$ is used to get a **difference score** to compare them.

To pick the inital parent strings, we use a premade random string-hash database. Given a hash h of a secret string s, we pick a set S_0 of strings from the database which have a difference score with s less than a certain threshold τ. Ranking the elements of $S_0 \times S_0$, highest closeness first, we cross the parents in 'chunks' where they have varying characters. The resulting progeny are then ranked based on the *decrease* in the difference score with s (i.e. higest similarity first). Choosing the fittest N progeny as set S_1, the process is then repeated. This continues iteratively until convergence of difference scores.

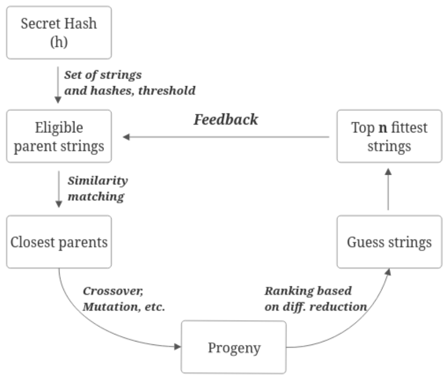
3.2 Observation

From the large random string-hash dataset that we generate, we observe that the members of the initial parent string set rarely have a difference score less than 60, with the average being approximately 75.

However, after a single iteration of this algorithm with threshold $\tau = 70$, we observe that there are a significant number (≈ 100) of child strings that have an improvement of at least a difference score of 10, with the top ranked strings reaching up to an improvement of almost 20. Iteration is expected to further reduce the difference scores with increasing number of generations.



(a) Strategy for Genetic Mastermind



(b) Strategy for fuzzy hash reversal

4 Conclusion/Future Work

We observe that the MaxMin strategy works best in terms of number of guesses. However, comparing time, we notice that the genetic algorithm approach outperforms all other algorithms in the order of hundreds. In conclusion, we can say that genetic algorithms, though not considered the best of methods, are promising for optimization problems involving huge search spaces. The simplicity of genetic algorithms make them good candidates for further research and implementation into real world problems.

One observation is that current implementations of our algorithms faced a few issues, for example in the Eligible Set approach. Another observation from the practical application of fuzzy hashing is that the children strings do not provide any real increase in the characterwise match with the secret string, whereas due to increasing *pattern similarity*, the difference score reduces.

We intend to work on these in the future, as well as another interesting application brought up during the course of this project, which is protein to RNA Reverse Translation and mutation detection.

5 Contributions

- Subhash Chandra P: Berghman's Algorithm
- Rahul Anand: The GenMM Heuristic and Consistency of a Combination
- Siddharth Singh: The MaxMin Information Strategy, Practical applications
- Krutay Upadhyay: Fuzzy Hashing, Project Report
- Amogh R: Initial Game Files, Fuzzy Hashing, Presentation
- R Amogh: Webapp for GenMM, Donald Knuth's Mastermind

Github: https://github.com/GitGudBruhh/MiniProjectSARKAS

6 Appendix

6.1 The MaxMin Information Strategy

6.1.1 Information Contents

After the i^{th} round of the game, the 3 -tuples $\left(P_{i-1}, G_i^k, R_i^j\right)$ may be viewed as a message m_i from an information source, $\mathcal{S}$. Let R_i^j be the setter's i^{th} response. The amount of information from $m_i = \left(P_{i-1}, G_i^k, R_i^j\right)$ is:

$$I\left(m_i\right) = -\log_2\left(\text{Prob}\left(R_i^j / \left(P_i, G_i^k\right)\right)\right) \text{bit(s)}.$$

To calculate $I\left(m_i\right)$, we must find the value of Prob $\left(R_i / P_i^j, G_i^k\right)$.

$$P_{i-1} = C_i^k\left(R_i^1\right) \cup C_i^k\left(R_i^2\right) \cup \ldots \cup C_i^k\left(R_i^j\right) \cup \ldots$$

where $C_i^k\left(R_i^j\right)$ is simply the set of patterns in P_{i-1} resulting in a response R_i^j when compared to G_i^k.

$$\text{Prob}\left(m_i\right) = \text{Prob}\left(R_i^j / \left(P_{i-1}, G_i^k\right)\right) = \frac{\left|C_i^k\left(R_i^j\right)\right|}{\left|P_{i-1}\right|}, \text{ and}$$

$$I\left(m_i\right) = \log 2\left(\frac{|Pi-1|}{\left|C_i^k\left(R_i^j\right)\right|}\right) \text{bit(s)}.$$

The game will end after the m^{th} step, if and only if:

$$\sum_{i=1}^{m} \log_2\left(I\left(m_i\right)\right) = \log_2\left(R^N\right) = N\log_2(R)$$

6.1.2 Pseudo Code

Algorithm 1 MaxMin Procedure

```
1: procedure MaxMin($P_{i-1}$)
2:     for all possible guesses $G_i^k \in P_{i-1}$ do
3:         for all possible responses $R_i^j$ do
4:             Compute $I\left(m_i^{k,j}\right)$
5:         end for
6:         Sort $I\left(m_i^{k,j}\right)$ in ascending order to a vector $V_i^k()$
7:     end for
8:     Select guess $G_i^o$ for which $V_i^o(x) \geq V_i^k(x)$, and $V_i^o(y) > V_i^k(y)$; where $k \neq o$, and
       $x = 1, 2, \ldots, y-1$.
9: end procedure
```

6.2 Berghman's Algorithm

Algorithm 2 An Alternative Genetic Algorithm

```
1: Set $i = 1$;
2: Play fixed initial guess $g_1$;
3: Get response $X_1$ and $Y_1$;
4: while $X_i \neq P$ do
5:     $i = i + 1$;
6:     Set $E_i = \{\}$ and $h = 1$;
7:     Initialize population;
8:     while ($h \leq$ maxgen) and ($|E_i| <$ maxsize) do
9:         Generate new population using crossover, mutation,
10:        inversion and permutation;
11:        Calculate fitness;
12:        Add eligible combinations to $E_i$ (if not yet contained in $E_i$);
13:        $h = h + 1$;
14:    end while
15:    Play guess $g_i \in E_i$;
16:    Get response $X_i$ and $Y_i$;
17: end while
```

6.3 The GenMM Distance Heuristic and Consistency of a Combination

In accordance with the paper written by J.J. Merelo-Guervo´s, P. Castilloa, and V.M. Rivas, key terminologies and concepts have been described below:

- The term *hint h* shall be used to describe the response to a combination of colors that the agent plays (call that combination c_{played}). In this case, the response refers to the number of black and white pegs obtained as a result of that play. Thus, a hint $h(c_i, c_j)$ is of the form (n_b, n_w) where n_b is the number of black pegs and n_w is the number of white pegs obtained while comparing combination c_i to c_j
- Further let c_{secret} denote the combination of colors given by the secret code.
- A tuple comprising of hint along with a combination (c_{played},h) shall be called a *rule r*.
- Some arbitrary combination of colors *c* is said to meet or be *consistent* with a *rule r* if and only if

$$h(c_{played}, c_{secret}) = h(c, c_{played}) \tag{2}$$

that is, the combination c should have as many black and white pins with respect to the played combination as that of the played combination with respect to the secret code.

- In general, an arbitrary combination *c* is said to be *consistent* if and only if

$$\forall r_i = (c_i, h_i) \quad h_i = h(c, c_i) \quad where \quad i = 1, \dots, n \tag{3}$$

Here n refers to the number of combinations played so far. Thus, the combination c is consistent *iff* it is consistent with all rules obtained up to the latest point of play. (Here, h_i denotes $h(c_i, c_{secret})$ and c_i indicates the played combination in turn i)

- A combination that is consistent has chance of being the secret code to be obtained as it meets all the constraints (which are the hints.)
- The "distance" heuristic for a combination indicates how close the combination is to meeting all hints and it is given by (where $h_i = (n_{bi}, n_{wi})$)

$$d(c, c_i) = d(h(c, c_i), h_i) = |n_b - n_{bi}| + |n_w - n_{wi}| \tag{4}$$

In accordance with the above heuristic, any consistent combination should have a 0 distance to all combinations played.

- Further, the *fitness f* for some combination *c* is given by:

$$f(c) = \sum_{i=1,\dots,n} -d(c, c_i) \tag{5}$$

which posses a negative value for a inconsistent combination and is 0 for a consistent one.

7 References

1. Knuth, Donald (1976–1977). "The Computer as Master Mind" (PDF). J. Recr. Math. (9): 1–6.
2. Koyama, Kenji; Lai, Tony (1993). "An Optimal Mastermind Strategy". Journal of Recreational Mathematics (25): 230–256.
3. Berghman, Lotte (2007–2008). "Efficient solutions for Mastermind using genetic algorithms" (PDF). K.U.Leuven (1): 1–15.
4. Merelo J.J.; Mora A.M.; Cotta C.; Fernández-Leiva A.J. (2013). "Finding an Evolutionary Solution to the Game of Mastermind with Good Scaling Behavior". In Nicosia, G.; Pardalos, P. (eds.). Learning and Intelligent Optimization. Lecture Notes in Computer Science. Vol.7997. Springer. pp. 288–293
5. De Bondt, Michiel C. (November 2004), NP-completeness of Master Mind and Minesweeper, Radboud University Nijmegen
6. Merelo-Guervós, J.J., Runarsson, T.P. (2010). Finding Better Solutions to the Mastermind Puzzle Using Evolutionary Algorithms. In: Di Chio, C., et al. Applications of Evolutionary Computation. EvoApplications 2010. Lecture Notes in Computer Science, vol 6024. Springer, Berlin, Heidelberg.
7. Maestro-Montojo, J., Merelo, J.J., Salcedo-Sanz, S. (2013). Comparing Evolutionary Algorithms to Solve the Game of MasterMind. In: Esparcia-Alcázar, A.I. (eds) Applications of Evolutionary Computation. EvoApplications 2013. Lecture Notes in Computer Science, vol 7835. Springer, Berlin, Heidelberg.
8. Runarsson, T.P., Merelo-Guervós, J.J. (2010). Adapting Heuristic Mastermind Strategies to Evolutionary Algorithms. In: González, J.R., Pelta, D.A., Cruz, C., Terrazas, G., Krasnogor, N. (eds) Nature Inspired Cooperative Strategies for Optimization (NICSO 2010). Studies in Computational Intelligence, vol 284. Springer, Berlin, Heidelberg.