MASTERMIND

HADAR TOURGEMAN (212885453) , SAMA MILHEM (214102972) & MAJD GHANAYEM (213641939)

INTRODUCTION TO ARTIFICIAL INTELLEGENCE - 67842

Table of Contents

INTRODUCTION

For our AI project, we have chosen to implement the classic code-breaking game **Mastermind**. In this two-player game, the "codemaker" creates a secret sequence of four colored pegs from six colors. The "codebreaker" must deduce this sequence within 8 to 12 attempts, receiving feedback after each guess in the form of black pegs (correct color and position) and white pegs (correct color, wrong position).

From an AI perspective, Mastermind presents a fascinating challenge. It requires navigating a vast search space, efficiently interpreting feedback, and dynamically updating strategies. These aspects mirror real-world AI problems, including the need to balance exploration and exploitation, handle incomplete information, and adapt based on evolving feedback.

Our project aims to explore and compare several AI algorithms in solving Mastermind. **We have chosen this game** because its deterministic feedback and constrained problem space make it an ideal testing ground for various AI techniques.

We plan to model and solve Mastermind by employing both reinforcement learning and local search methods. Specifically, we will implement a **Q-learning agent** to represent the reinforcement learning approach. Q-learning is a model-free reinforcement learning algorithm that learns the best action to take in each state by iteratively updating its knowledge base through interactions with the environment.

In addition to reinforcement learning, we will explore local search algorithms, which are known for their ability to find solutions in large search spaces. More specifically, we will implement a **Genetic Algorithm**. The Genetic Algorithm, inspired by natural selection, evolves a population of candidate solutions over successive generations by applying operations such as selection, crossover, and mutation.

To establish a meaningful benchmark for our comparative analysis, we will assess the performance of our proposed algorithms against two key baselines. First, we will evaluate them in relation to **a simple strategy of random guessing**, which represents the naivest approach to solving the Mastermind puzzle. This will help us quantify the improvement each algorithm offers over purely chance-based attempts, providing a clear measure of their efficacy. Secondly, we will contrast our algorithms with **previously suggested methods from the literature**, including well-known strategies like Knuth's algorithm and its variants. By doing this, we can not only see how our algorithms improve on basic approaches, but also how they measure up to more sophisticated methods. This comparison will help us understand what each algorithm is good at, where it might fall short, and how it fits into the bigger picture of Mastermind-solving techniques.

We will evaluate our algorithms based on two main criteria: how effectively they minimize the number of guesses required to correctly break the secret code, and their overall success rate across different game variations, such as varying the length of the secret code and the number of possible colors. This comprehensive analysis will help us

identify which algorithm performs best under different conditions, offering valuable insights into the strengths and limitations of each approach in solving the challenges posed by Mastermind.

Usage Of LLMs:

During this project, we have leveraged LLMs, specifically ChatGPT, as a supportive tool to enhance our work efficiency. While not forming a core part of our research, LLMs have been utilized for several tasks: code debugging to identify and resolve issues in our implementation; report editing to improve the grammar and clarity ; graph creation to assist in visualizing our data and results; and interface development to aid in the creation of HTML and JavaScript components, areas in which we had limited prior experience.

It's important to note that while LLMs have been valuable in improving our workflow and output quality, they have served primarily as assistive tools. The core research, algorithm implementation, and analysis remain the product of our team's efforts.

PREVIOUS WORK

Mastermind, while simple on the surface, presents deep combinatorial challenges due to the vast number of code combinations and the indirect nature of the feedback provided after each guess. Historically, the game has attracted considerable attention within the AI and algorithmic research communities, leading to the development of various strategies aimed at efficiently solving the game.

One of the most notable algorithms is **Donald Knuth's Minimax algorithm(1977)**, which guarantees identifying the secret code in five moves or fewer, with an average of 4.478 guesses for four pegs and six colors. The algorithm uses a minimax technique to minimize the worst-case scenario by selecting the guess that leaves the fewest remaining possibilities. Despite its optimality in terms of guesses, the algorithm's computational intensity, as it evaluates all codes, limits its time efficiency.

Alternative strategies have explored local search and evolutionary algorithms. **Bernier et al. (1996)** applied **simulated annealing**, a probabilistic technique that starts with an initial solution and iteratively explores its neighborhood. This method introduces randomness, allowing the algorithm to escape local optima and find better solutions over time.

Further developments in local search algorithms include the **Random Mutation Hill Climbing** approach, adapted to Mastermind by **Temporel and Kovacs(2003)** based on the work of **Baum et al.(1995)**. This algorithm uses a hill-climbing strategy combined with a heuristic to iteratively improve the current solution by making small random changes. This approach aims to find a satisfactory solution by progressively refining guesses based on feedback.

Genetic Algorithms have also been applied to Mastermind with notable success. **Bento et al.(1999)** introduced one of the earliest GAs for the game, which begins by initializing

a random population of candidate solutions. The fitness of each candidate is evaluated based on how well it matches the feedback from previous guesses. The GA then evolves this population through selection, crossover, and mutation, progressively improving the quality of the guesses. Despite the randomness inherent in GAs, Bento's algorithm requires an average of 7.538 guesses, which is higher than Knuth's Minimax but offers flexibility in exploration.

Subsequent research by **Bernier et al. (1996) and Berghman et al. (2009)** introduced variations of GAs aimed at improving performance. Bernier's GA incorporated mechanisms to maintain population diversity, crucial for efficiently exploring the changing fitness landscape. This approach reduced the average number of guesses to 5.62. Berghman et al. (2009) further refined the GA by utilizing multiple generations to create a large set of eligible guesses, selecting the most promising one as the next move. Their algorithm achieves an average of 4.47 guesses and is noted for its computational efficiency, requiring only 0.762 seconds on average per guess.

In summary, while Knuth's Minimax remains a benchmark for its optimal move count, other methods like simulated annealing, hill climbing, and GAs provide alternative solutions, balancing computational efficiency and adaptability.

METHODOLOGY

Local Search

The Fitness Function

The fitness function we implemented plays a pivotal role in evaluating and guiding the search process in the Genetic Algorithm approach for solving Mastermind.

This function assesses the quality of a trial code by comparing it to all previous guesses and the feedback they received. Specifically, it calculates the number of black pegs (indicating correct color and position) and white pegs (indicating correct color but incorrect position) that the trial code would score if the previous guesses were the secret code. The differences between these values and the actual feedback measure the trial code's accuracy. A lower fitness score means the trial code better fits previous feedback, guiding the algorithm toward more accurate solutions. This method, adapted from Berghman et al. (2009) paper "Efficient Solutions for Mastermind Using Genetic Algorithms", helps the algorithm progressively identify the secret code.

Formally, we get that to compute the fitness value of a guess c, we compare it with every previous guess g_q, where $X'_q(c)$ is the number of black pins and $Y'_q(c)$ is the number of white pins, such that:

$$fitness\_Value(c) = \sum_{q=1}^{i} \left| X'_q(c) - X_q \right| + \sum_{q=1}^{i} \left| Y'_q(c) - Y_q \right|$$

Where X_q is the number of black pins in the feedback of the g_q guess and Y_q is the number of white pins.

An illustration

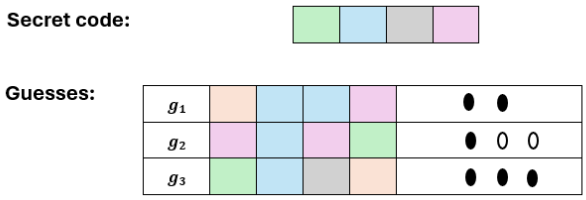

Figure 1- Fitness value calculation

$$\left|X'_{g_2}(g_3) - X_{g_2}\right| + \left|X'_{g_1}(g_3) - X_{g_1}\right| = |1-1| + |1-2| = 1$$

$$\left|Y'_{g_2}(g_3) - Y_{g_2}\right| + \left|Y'_{g_1}(g_3) - Y_{g_1}\right| = |1-2| + |1-0| = 2$$

$$\boldsymbol{Fitness\_value(g_3) = 2 + 1 = 3}$$

In the results section, we will discuss potential variations of the fitness function like weighting black and white pegs differently and adding a penalty based on the number of turns. However, these changes were less effective, and resulted in an increasing of the average number of turns. Therefore, we kept the original function, which consistently performed better.

The New Genetic Algorithm

Genetic Algorithms in General:

The genetic algorithm solves optimization problems based on **natural selection**, repeatedly modifying a **population** of individual solutions. At each step, it **selects** individuals from the current population to be parents and uses them to **produce children** for the next generation. Over successive generations, the population "evolves" toward an optimal solution.

The New Genetic algorithm:

The “new genetic algorithm” is a variation of GAs that aims to enhance the standard genetic algorithm by introducing additional mechanisms to improve efficiency and accuracy.

Steps:

1. **Initialization**
   The algorithm begins by generating an **initial random population** of 150 possible solutions, each referred to as a “chromosome”, represented as a sequence of colors. The diverse initial population ensures broad exploration of the solution space.

2. **Selection**
   We then select **40% of the individuals** from the current population, to serve as parents for the next generation, with selection probability proportional to fitness

value, such that individuals with **better fitness scores** (lower scores), have a higher likelihood of being chosen. This ensures the next generation is created using the most promising solutions found so far.

3. **Crossover**
   Crossover combines parts of two parent codes to create offspring, mimicking natural reproduction. The algorithm employs either **1-point or 2-point crossover with equal probability,** ensuring diverse offspring for broad solution exploration.

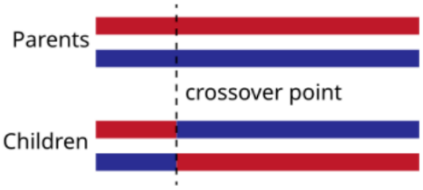

Figure 2- one point crossover

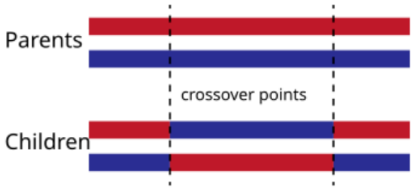

Figure 3 - 2 points crossover

4. **Mutation, Permutation, and Inversion**
   We then apply mutation, permutation, and inversion to maintain genetic diversity and avoid premature convergence to suboptimal solutions.
   - **Mutation -** After crossover, a 3% mutation probability changes one randomly selected peg to a different color.
   - **Permutation -** After crossover, a 3% permutation probability swaps the colors of two random positions in the code.
   - **Inversion -** After crossover, a 2% permutation probability reverses the sequence of colors between two randomly selected positions in the code.

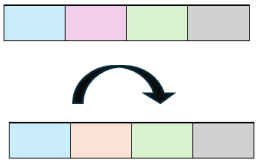
Figure 4 - Mutation

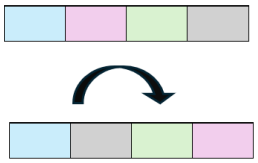
Figure 5 - Permutation

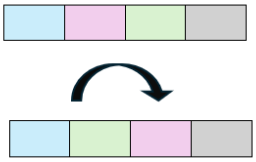
Figure 6 - Inversion

5. **Fitness Evaluation**
   After generating the new population, we evaluate each individual's fitness by comparing it to all previous guesses, as explained earlier.
6. **Eligibility Set**
   The eligibility set includes codes with a fitness score of zero, meaning they perfectly match the feedback from all previous guesses and are the most likely candidates for the correct solution.
7. **Choosing the Next Guess**
   After MAX_GENERATIONS (set to 100 in our version based on experiments from the referenced paper), the next guess is randomly selected from the Eligibility Set. This set contains the most promising candidates—codes with a fitness score of

zero that perfectly match previous feedback—focusing the search on the most accurate solutions and increasing the chances of finding the correct code quickly.

8. **Termination**
   The process continues until the correct code is guessed.

Reinforcement Learning

Q-Learning Agent

Q-learning is designed to enable an agent to learn a policy that maximizes cumulative rewards by interacting with its environment over time. The environment is typically modeled as a Markov Decision Process (MDP), where the agent observes its current state, takes an action, receives a reward, and transitions to a new state. This process can be represented as a sequence of state-action-reward interactions, with the goal of learning an optimal policy that maximizes the total expected reward (Q-value) for each state-action pair.

We chose Q-learning for solving Mastermind because it effectively uses feedback to improve over time. Mastermind provides clear guidance after each guess, which aligns well with Q-learning's ability to learn through trial and error. This method allows the agent to explore possible guesses while refining its strategy, making it ideal for navigating the game's large search space. The balance between exploration and exploitation ensures efficient learning, thus we believe that Q-learning could be a strong tool for solving the Mastermind problem.

Abstracting the State Space

n our implementation, we simplify the problem by abstracting the state space. Instead of dealing with color sequences directly, we convert each guess into a numerical index and vice versa. This abstraction allows the Q-learning agent to handle the large number of possible guesses more efficiently, speeding up the search process. Each action corresponds to selecting a new guess from the set of possible codes. The agent uses an epsilon-greedy strategy to balance exploration (i.e. choosing random actions) and exploitation (i.e. selecting the best-known action based on current Q-values). This ensures the agent learns while continuing to explore potential new solutions.

Learning Process

The algorithm learns through trial and error without prior knowledge of the environment, updating its Q-value table at each time step using the following rule:

$$Q(s,a) = Q(s,a) + \alpha \cdot \left[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)\right]$$

Where:

- $Q(s, a)$ represents the Q-value of taking an action a in state s.
- α is the learning rate, which controls how much new information overrides old information.
- r is the immediate reward received after taking action a.
- γ is the discount factor, which represents the importance of future rewards.
- $\max_{a'} Q(s', a')$ represents the maximum Q-value of the next state s' over all possible actions a'.

Parameter Selection and Performance

We tested multiple **hyperparameter combinations** and selected the following values for the final implementation: learning rate $\alpha = 0.3$, discount factor $\gamma = 0.8$, and exploration rate $\varepsilon = 0.4$.

For more complex configurations with a larger state space, computational challenges arose. To address this, we implemented **multithreading** to try and accelerate the simulations and evaluate the agent's performance across different setups.

Reward Structure

Since **the reward function** assigns a numerical value to each action, we reward the agent with (+1) for correctly guessing the secret code and penalize it with (-1) for incorrect guesses. This discourages repeating ineffective actions and encourages exploring other possibilities that could lead to the correct guess.

RESULTS

The New Genetic Algorithm: Evaluating the Impact of Weights and Penalties on Performance

In our implementation of the Genetic Algorithm for solving Mastermind, we tested 20 different combinations by varying the weights assigned to black pegs, white pegs, and the penalty using 100 games at each simulation. The fitness function was defined as:

$$fitness\_Value(c) = a \cdot \sum_{q=1}^{i} \left| X'_q(c) - X_q \right| + \mathrm{b} \cdot \sum_{q=1}^{i} \left| Y'_q(c) - Y_q \right| + penalty$$

Where :

- a and b are the weights for black pegs (X) and white pegs (Y) respectively, with $\boldsymbol{a, b} \in [\mathbf{1}, \mathbf{4}]$**.**
- The penalty term is calculated as:

$$penalty = penalty\_\text{weight} \cdot (num\_iterations - 1) \cdot num\_colors$$

with $\boldsymbol{penalty}$**_weight** $\in [\mathbf{0}, \mathbf{4}]$**.**

Our primary focus was to explore how different ratios between a and b affected the algorithm's performance on the default game configuration (6 colors, 4 positions). We also examined how adding a penalty influenced the search, as the number of iterations increased, leading to larger penalties that discouraged longer solution times. This exploration of the different combinations aimed to identify the configuration that minimized the number of guesses required to crack the code.

Results – Top 11 combinations:

We will highlight the top 11 combinations that yielded the best performance in our tests[1]:

Black pegs weight	White pegs weight	Penalty weight	Average guesses
1	**1**	**0**	**4.66**
1	3	0	4.66
1	2	0	4.68
2	3	0	4.68
2	3	3	4.77
1	1	4	4.86
1	1	1	4.88
2	3	2	4.9
2	3	1	4.9
1	3	2	4.9
1	1	2	4.91

Table 1 - Impact of Weights and Penalties on Performance

In the provided table, we can see that the best-performing combination is when the black pegs weight is 1, the white pegs weight is 1, and the penalty weight is 0, yielding an average of 4.66 guesses. This means that giving equal importance to both black and white pegs, without applying a penalty for the number of iterations, resulted in the most efficient solution.

Interestingly, the study we **referenced (Efficient Solutions for Mastermind Using Genetic Algorithms)** also found that using equal weights for black and white pegs was optimal. However, their results indicated better performance when applying a penalty weight of 2, which differs from our findings (where this combination resulted in higher average number guesses). This discrepancy could be due to the slight differences in the implementation, such as how certain operations are handled or how randomness is introduced, which could influence the outcome and affect the overall efficiency of the algorithm.

[1] For the remaining 9 combinations, you can refer to the fitness_weights_results.csv file, which is included in the output directory attached to our code

One possible intuition behind using the same weight for black and white pegs is that both pieces of feedback are equally important in narrowing down the correct code:

1. **Black pegs** (correct color and position) give very direct feedback about how accurate a guess is.
2. **White pegs** (correct color, wrong position) still provide valuable information, as they indicate that a color is correct but mispositioned. Ignoring or underweighting this information could slow down the refinement process.

We believe that by giving equal weight to both, the algorithm efficiently balances between refining the correct colors and adjusting their positions and overweighting one type of peg might lead to an inefficient search, focusing too much on either positions or color correctness, instead of both.

The Q - Learning Algorithm: Hyperparameters Tuning

In our Q-learning implementation, we needed to determine the optimal set of hyperparameter. To achieve this, we tested 64 different combinations using 1000 games at each simulation - α , γ and ε , each of which plays a crucial role in the learning process.

The α parameter, controls how quickly the agent adapts to new information. A higher alpha allows the agent to quickly learn from new experiences by overriding past knowledge, while a lower alpha makes it rely more on prior learning. This balance is important because too high of an alpha can cause instability, as the agent may adapt too quickly and lose track of valuable past insights, while too low of an alpha may slow down the learning process, thus we've used the following values for α - $\in \{0.1, 0.2, 0.3, 0.4\}$.

The γ dictates how much the agent values future rewards compared to immediate ones. A high discount means the agent prioritizes long-term rewards, encouraging it to plan ahead, while a low discount focuses the agent on short-term gains. Finding the right balance is key since a higher discount encourages foresight, which can improve performance in more complex scenarios, but it might cause the agent to take longer to adapt to immediate solutions. Thus, we've decided to check the following set of values – $\{0.5, 0.7, 0.8, 0.9\}$.

The ε parameter influences the balance between exploration and exploitation. A higher epsilon promotes exploration, where the agent tries random actions to discover new strategies, while a lower epsilon encourages exploitation, focusing on refining and using already learned strategies. High exploration is useful for discovering better solutions, but it may lead to more guesses initially. On the other hand, too low exploration helps the agent to refine its guesses but could cause it to miss better, unexplored options. Thus, we've decided to check the following set of values – $\{0.1, 0.2, 0.3, 0.4\}$.

By tuning these parameters, we aimed to find the optimal balance between exploration, learning speed, and strategic planning, ultimately improving the agent's performance and minimizing the number of guesses required to solve the game.

Results – top 10 combinations:

We will highlight the top 10 combinations that yielded the best performance in our tests[2]:

Alpha	Discount	Epsilon	Avg Guesses
0.3	**0.8**	**0.4**	**4.546**
0.2	0.5	0.4	4.554
0.3	0.9	0.2	4.563
0.1	0.9	0.2	4.58
0.4	0.7	0.1	4.581
0.3	0.7	0.4	4.583
0.3	0.8	0.3	4.584
0.2	0.7	0.2	4.593
0.2	0.8	0.1	4.595
0.2	0.8	0.3	4.596

Table 2 - Hyperparameters Tuning

As we can see, the best-performing combination in our Q-learning hyperparameter tuning is $\alpha = 0.3$, $\gamma = 0.8$ and $\varepsilon = 0.4$.

First, the moderate learning rate $\alpha = 0.3$ allowed the agent to update its knowledge based on new information without completely overriding past learning. This ensures that while the agent adapts to fresh feedback, it also retains valuable insights from previous guesses, which is critical for continuous improvement.

Second, the high discount factor ($\gamma = 0.8$) meant that the agent valued future rewards more heavily. In Mastermind, each guess provides important information that can guide future decisions. This forward-looking approach helps the agent think strategically over multiple guesses, improving its problem-solving efficiency.

Lastly, the exploration rate ($\varepsilon = 0.4$) ensured the agent maintained a good balance between exploring new strategies and exploiting known ones. Given the vast search space of possible code combinations in the game, this balance is key to avoiding getting stuck in suboptimal guesses while still refining its approach over time.

Algorithms Comparison: 6 positions, 4 colors

In this section, we compare our algorithms for the default Mastermind configuration (6 positions, 4 colors) based on the average number of guesses needed to solve 1,000 games. We benchmark our results against four well-known methods: a random agent,

[2] For the remaining 54 combinations, you can refer to the hyperparameter_results.csv file, which is included in the output directory attached to our code

Knuth's Minimax algorithm, and two versions of the Genetic Algorithm from previous research.

The first benchmark is **Berghman et al. (2009) - The New Genetic Algorithm**, which is described in the paper "Efficient Solutions for Mastermind Using Genetic Algorithms." And represents the **original version** of the Genetic Algorithm for solving the Mastermind problem.

The second benchmark is the **Vivian van Oijen (2018)** - Bachelor Thesis that explored the performance of the New Genetic Algorithm through practical experiments which makes her work a valuable point of comparison to assess both the speed and scalability of our Genetic Algorithm.

Finally, the results for **Knuth's Minimax algorithm** are based on theoretical outcomes published in the literature.

Algorithm	Avg. num. Of guesses (P=4, N=6)
The Q-Learning Algorithm	4.546
The New Genetic Algorithm	4.66
Random Algorithm	1153.95
Knuth's Minimax	4.478
Berghman et al. (2009)- The New Genetic Algorithm	4.47
OIJEN, V. van (2018)- The New Genetic Algorithm	4.56

Table 3- algorithms comparison

The **Random Algorithm** predictably performs the worst, requiring over 1,000 guesses on average. Lacking any strategy or feedback-driven adjustments, it relies entirely on random selection, which results in significantly more guesses compared to more strategic methods. In stark contrast, **Knuth's Minimax Algorithm**, which is designed specifically for Mastermind, delivers near-optimal performance, averaging 4.478 guesses. This algorithm, based on theoretical principles, sets the benchmark for efficiency in solving the problem.

Our **Q-learning algorithm** performed exceptionally well, averaging 4.546 guesses. This result highlights the power of reinforcement learning, which effectively balances exploration and exploitation to solve problems. Through trial and error, Q-learning progressively refines its strategy using feedback, making it a competitive approach.

Although it lags slightly behind Knuth's Minimax algorithm, its adaptability and learning capabilities make it a robust tool for tackling the problem.

The **New Genetic Algorithm** we implemented also showed strong performance, averaging 4.66 guesses. Though it didn't match Q-learning or Knuth's Minimax in efficiency, it still outperformed random guessing by a wide margin. Using evolutionary techniques like selection, mutation, and crossover, the genetic algorithm effectively narrows the solution space, although its inherent randomness can lead to some variability in results.

When compared to other benchmarks, **Berghman et al. (2009) - The New Genetic Algorithm** averaged 4.47 guesses, slightly outperforming our implementation. This suggests that our version is close to Berghman's original algorithm, with minor differences likely due to variations in our implementation. Similarly, **Vivian van Oijen's (2018) Genetic Algorithm**, which averaged 4.56 guesses, closely mirrors our results. Oijen's work, based on practical experiments in her Bachelor Thesis, aligns with our findings and highlights similar performance outcomes.

One key insight from Oijen's research is that Berghman's original algorithm does not scale as well in practical implementations, a finding that resonates with our results. Like Oijen, we found that while our genetic algorithm performed well, it didn't quite reach the efficiency of the original, emphasizing the challenge of replicating the exact effectiveness of the initial algorithm in real-world applications.

Overall, both the Genetic Algorithm and Q-learning demonstrate the strength of feedback-driven strategies, achieving results that come close to the theoretical optimum provided by Knuth's Minimax algorithm. While Q-learning slightly outperforms the genetic algorithm in this particular scenario, we will explore the advantages of using the genetic algorithm in the future sections.

Impact of Number of Colors and Code Length on Average Time and Number of Guesses

In this experiment, we aimed to analyze the impact of varying the number of colors and code positions on the average number of guesses and the time taken to solve the game. We tested colors ranging from 6 to 14 and code lengths (positions) from 4 to 8. For each combination of colors and positions, we generated a random secret code and ran 100 simulations using **The New Genetic Algorithm** with fixed parameters **and got the following results:**

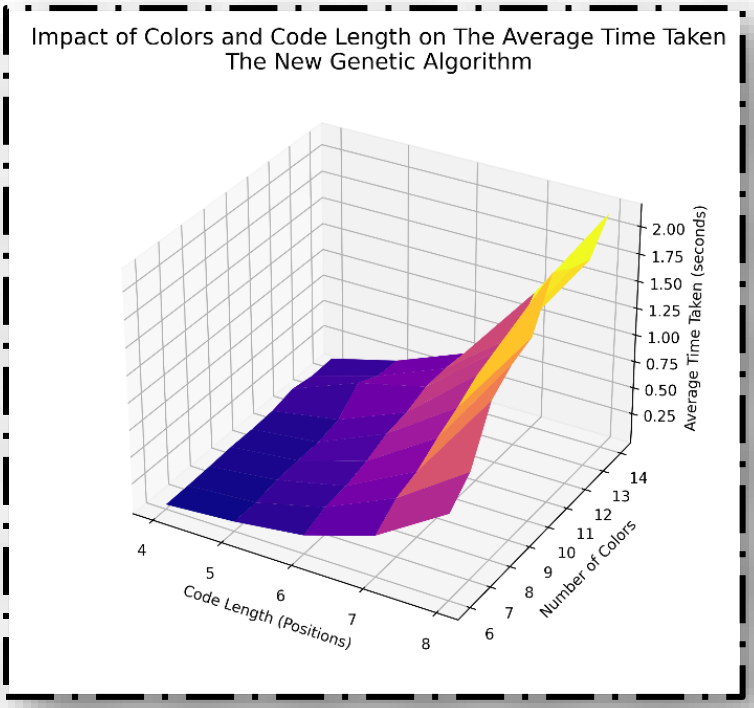

Figure 7- Average time taken

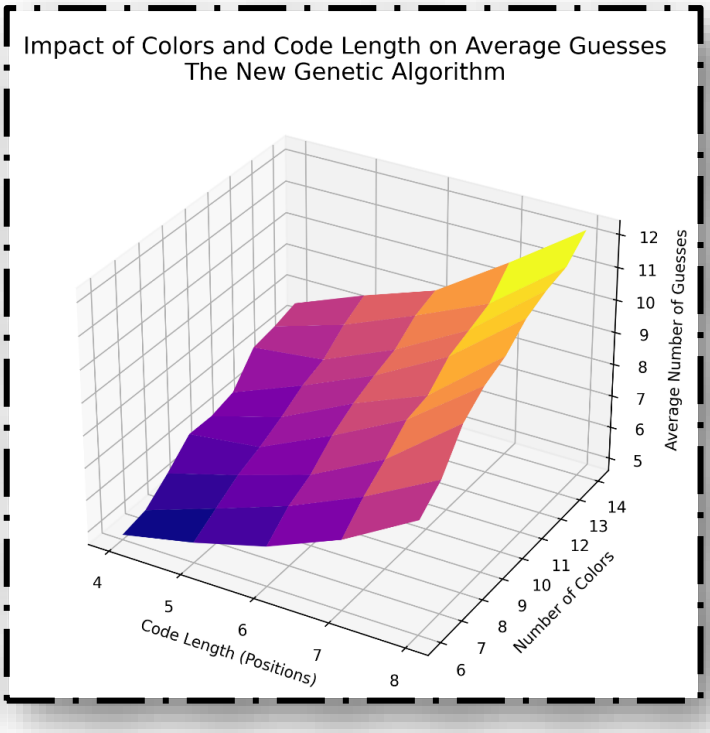

Figure 8- Average number of guesses

The first graph shows that as the number of colors and code length increase, **the average time taken** to solve the game rises significantly. This reflects the growing computational complexity of the problem. However, even though the time increases, it remains reasonable on average, demonstrating the efficiency of the algorithm despite the complexity. The second graph highlights **the average number of guesses** required to solve the game, which also increases with more colors and longer code lengths. For shorter code lengths (i.e., 4 or 5), the number of guesses remains relatively low, but it escalates rapidly as both parameters increase, with the most complex scenarios requiring over 12 guesses. These graphs collectively demonstrate that as the game's complexity increases, both the number of guesses and the time required to find a solution rise.

As for the Q-learning algorithm, when we tested its performance on more complex Mastermind configurations, we encountered a significant challenge: the algorithm's runtime increased substantially as the problem complexity grew. This slowdown can be attributed to both the space complexity of storing the Q-table and the runtime complexity of training the agent. As we increase the code length and the number of colors, the number of possible states and actions expands drastically. Specifically, the state space grows as $\boldsymbol{num\_colors}^{\boldsymbol{code\_length}}$ which is manageable for smaller configurations like 6^4 but becomes far more computationally demanding for larger numbers.

To address this, we introduced **multithreading to the algorithm**, allowing parallel execution of tasks. This adjustment helped mitigate the slowdowns and enabled us to better assess the algorithm's performance in higher-dimensional configurations. To avoid

overwhelming the system, we limited each simulation to 50 games and tested with colors ranging from 6 to 9 and code lengths between 4 and 6. The results of these simulations, are summarized below, using 3 different heatmaps – each represent a single metric - average turns, average time per game, and total training time across different color and position configurations

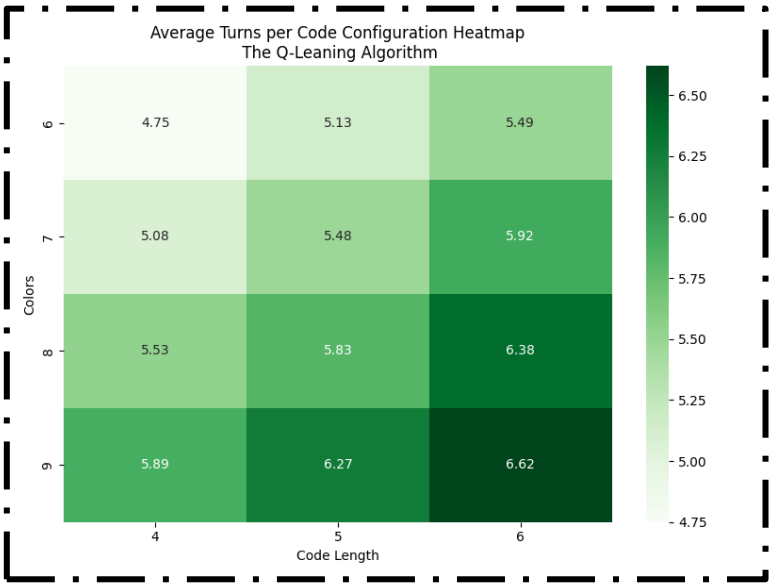

Figure 9 – Average Turns Taken

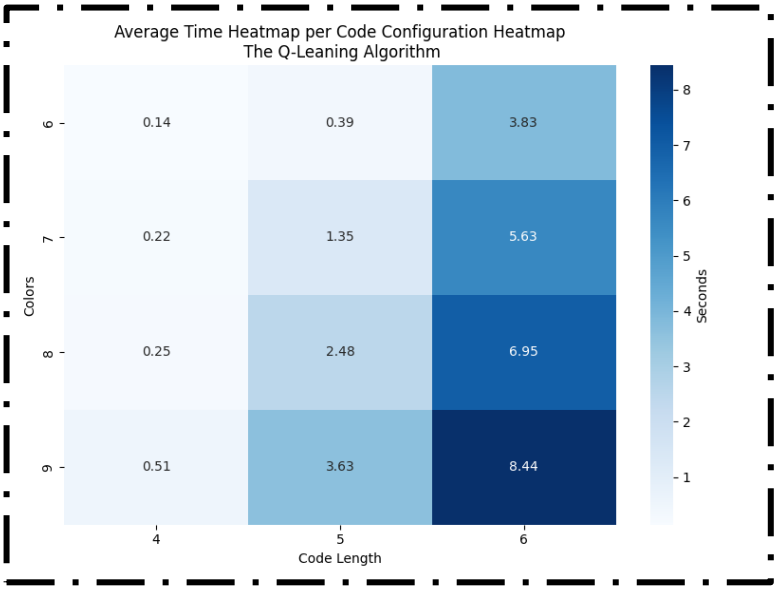

Figure 10 – Average Time Taken

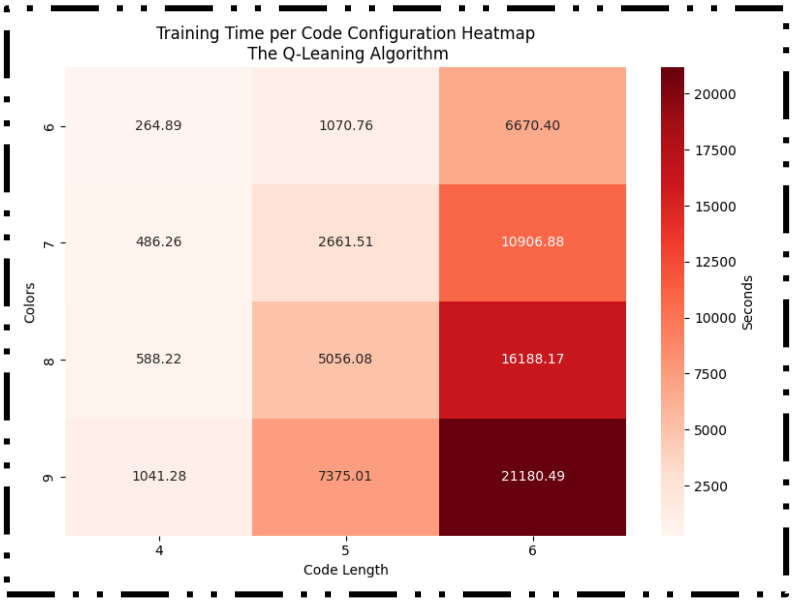

Figure 11 – Training Time

The first heatmap illustrates the average number of guesses the Q-Learning agent requires to crack the code. Even as the number of colors and code length increase, the number of guesses remains relatively low. This highlights the agent's efficiency in finding solutions, even as the problem complexity grows. In contrast, the New Genetic Algorithm

shows a sharper increase in the number of guesses with the same increase in complexity, indicating that **Q-Learning is more effective at scaling to larger configurations.**

The second heatmap, which represents the average time per game, also demonstrates the agent's solid performance. Despite the increase in complexity, the game time remains reasonable, **though there is a notable increase in time compared to the Genetic Algorithm.** This suggests that while Q-Learning performs efficiently in terms of guesses, it may require more time to compute each move in complex scenarios. The increase in time here is more pronounced than in the Genetic Algorithm, though still within a manageable range for gameplay.

The most significant challenge arises during the agent's training phase, as shown in the training time heatmap. For a default game configuration (6 colors and 4 positions), training takes around 264 seconds. However, as the number of colors and code length increase, the training time grows exponentially, extending to hours for more complex configurations. This is due to the dramatic expansion of the state space, where the number of possible color and position combinations balloons, significantly slowing down the training process.

Despite these long training times, the Q-Learning agent demonstrates its real strength after the training phase is complete. Once trained, the agent can quickly and efficiently make guesses in subsequent games, handling even the most complex configurations with ease. **The bottleneck lies in the initial training,** but afterward, the agent maintains low turn counts and reasonable gameplay times. This underscores the adaptability and power of Q-Learning, making it a strong choice for complex challenges—though the lengthy training process is a significant factor to consider in terms of scalability.

Interactive GUI Game:

An Overview of the Game's Graphical Interface and Mechanics

Our Mastermind game GUI provides three engaging gameplay modes designed to offer both classic and competitive experiences.

In the **Classic Game** mode, the computer selects a secret 4-color code randomly. The player has up to 10 attempts to guess the code, with feedback provided after each guess to guide their strategy. This mode maintains the classic challenge of Mastermind, offering a familiar and strategic gameplay experience.

The **Play Against a Genetic Agent** mode adds a competitive edge to the game. In this mode, the secret code is concealed from both the player and the AI, which employs our **Genetic algorithm** to make its guesses. While the player can see the AI's guesses, they do not receive feedback on those guesses. The objective is to discover the code in fewer guesses than the AI. Success in this mode earns the player 5 points for each victory over the AI. These points can be used to obtain hints in subsequent games, providing an opportunity to enhance strategic play.

The **Q-Learning Agent Mode** allows users to run a real-time simulation of a Q-Learning agent as it trains and improves its ability to guess a secret code. In this mode, users can observe the training process and watch the agent's performance on their secret code.

One important note: When we translated our Q-Learning code from Python to JavaScript, the agent's training time significantly improved, even when playing with more than six colors. This performance boost is mainly due to the optimizations in handling loops and state arrays in JavaScript, making the simulation run more efficiently.

The game can be accessed through the **`menu_page.html`** file, which serves as the entry point for choosing between the three gameplay modes and optionally choosing the number of colors. This feature adds an extra layer of difficulty, allowing players to tailor the challenge to their preference.

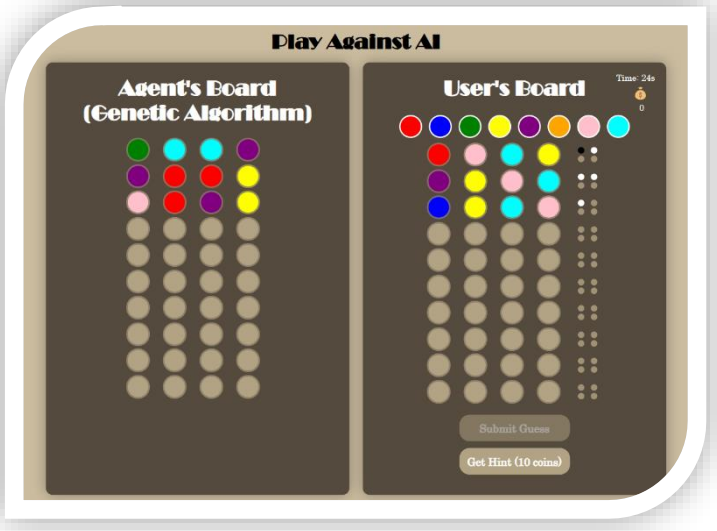

Figure 12 – Genetic_game.html page, allows the user to select from up to 10 optional colors.

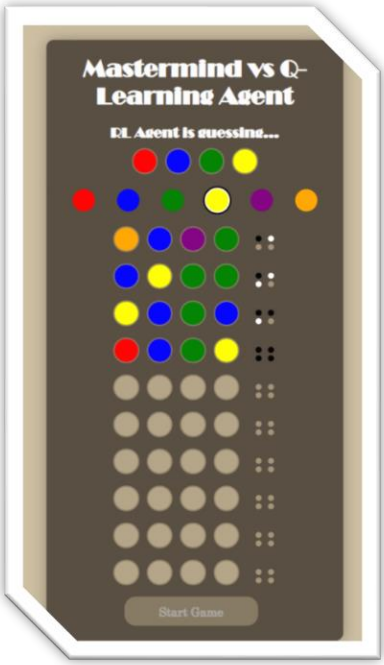

Figure 13 – RLagent.html page, allows the user to set a secret code and watch An RL agent simulation

SUMMARY

IIn this project, we tackled the classic code-breaking game Mastermind using artificial intelligence techniques, specifically focusing on local search methods and reinforcement learning. Our goal was to implement and compare two approaches: a variation of the genetic algorithm, called the New Genetic Algorithm, and a Q-learning agent. These were benchmarked against random guessing and Knuth's Minimax algorithm, a well-known solution for Mastermind.

Mastermind is a challenging AI problem due to its vast search space and the need to interpret indirect feedback from guesses. Our New Genetic Algorithm built on previous research, incorporating techniques like mutation, permutation, and inversion to maintain

diversity in the population. Meanwhile, the Q-learning agent learned optimal strategies through iterative gameplay, using feedback from the environment to refine its guesses.

Our results showed that both AI approaches significantly outperformed random guessing, highlighting the value of intelligent strategies. Compared to The New Genetic algorithm, the Q-learning agent slightly edged it out in terms of average guesses required. However, both methods demonstrated robust performance, closely matching the theoretical optimal solutions.

We conducted experiments to explore how different parameters affected performance. For the genetic algorithm, we focused on identifying the most effective weights and penalties for faster solutions. For Q-learning, we tested different hyperparameter combinations to optimize its guess count.

A key insight from our experiments was the impact of increasing the number of colors and code length. As the game became more complex, both algorithms required more time and guesses, but Q-learning showed more consistent performance across various configurations. The genetic algorithm, while faster overall, showed steeper increases in guesses for larger problem spaces. A major challenge for Q-learning was the training time, particularly in higher dimensions, where multithreading and other optimizations were necessary.

While our results are promising, it's important to acknowledge that the algorithms were tested under controlled conditions. In real-world applications, factors like time constraints and computational resources could influence which approach is more suitable. The genetic algorithm's population-based search may be more adaptable to larger search spaces, while Q-learning might need more refined state representations in such cases.

Both approaches can be adapted to other optimization and search problems. The genetic algorithm’s population-based strategy could be applied to resource allocation or scheduling tasks, while Q-learning’s reinforcement learning framework is well-suited to problems with delayed rewards, such as financial decision-making or game AI.

Future work could explore deeper reinforcement learning techniques, such as deep Q-networks, to capture more complex patterns. Hybrid approaches that combine the strengths of genetic algorithms and reinforcement learning could also be a promising direction, allowing for faster convergence and improved efficiency in solution-finding.

In conclusion, our project successfully demonstrated the potential of AI techniques to solve complex games like Mastermind. Both the New Genetic Algorithm and Q-learning showed strong performance, but there remains significant room for improvement and further exploration, particularly in scaling these solutions to larger, more complex problems. As AI continues to evolve, we anticipate even more sophisticated strategies to emerge, not only for games but for real-world applications with similar characteristics.

REFERENCES

1. Berghman, Lotte, Dries Goossens, and Roel Leus. 2009. “Efficient Solutions for Mastermind Using Genetic Algorithms.”
2. OIJEN, V. van. Genetic Algorithms Playing Mastermind. 2018. Bachelor's Thesis.
3. Mastermind (board game), Wikipedia.
4. https://github.com/egeromin/mastermind.git - Environment class code, for the q-learning agent.

LINK TO THE CODE

https://github.com/SamaMilhem/AI_Project.git