UNIVERSITY OF LEEDS

Automatic Puzzle Solving: Mastering Mastermind

Final Year Mathematical Project

Start Date: 01/01/2024

by

Mia Glynn
mm21meg@leeds.ac.uk

Under the supervision of

Dr. Adrian Martin

School of Mathematics
University of Leeds

Contents

1 Introduction

1.1 What is Mastermind?

Mastermind was created as a board game in the early 1970s. It has 2 players, a code-maker and a code-breaker. Firstly, the code-maker chooses a code, traditionally consisting of 4 pegs and 6 possible colours (repetitions of colours are permitted). The code-breaker then guesses a 4-peg code. Feedback is given by the code-maker in the form of black and white pegs:

- **black** indicates how many pegs in the code are the correct colour, and in the correct position
- **white** indicates how many pegs in the code are the correct colour, but in the wrong position

This continues, with the code-breaker using the feedback to inform the next guess, until either a correct guess is made (the code-breaker wins) or all guesses, traditionally ten, are used (the code-maker wins). Other ways to play include awarding points to the code-maker for each guess the code-breaker makes, for a set number of games. The winner is the player with the most points after all games are played.

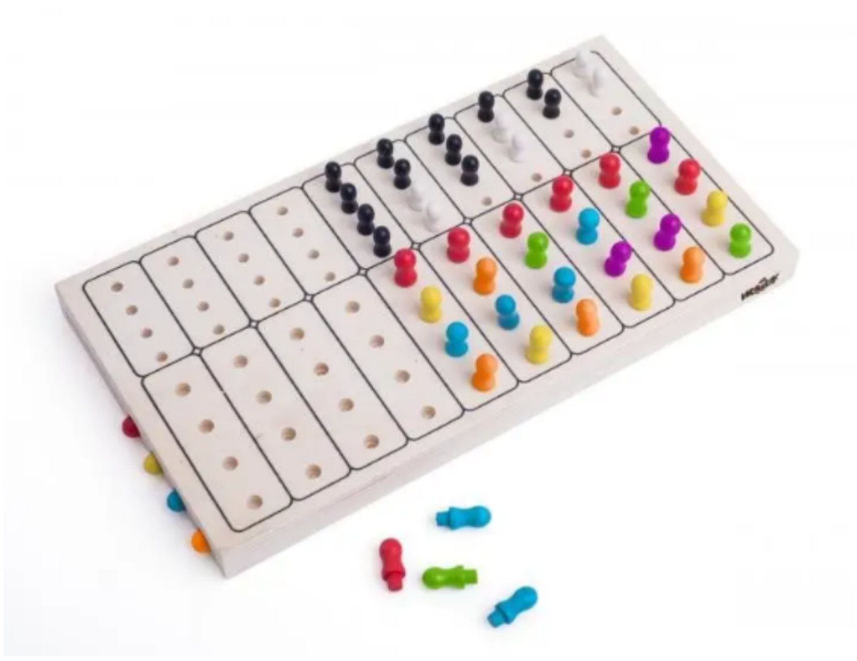

Figure 1: Example of a Mastermind Board (Alza.cz, n.d.)

Figure 1 shows that the code-maker has chosen the hidden code (bottom-left), and the code-breaker has used six of their ten guesses, with feedback given on the left side of the board.

1.2 Ethics and Aims

As Mastermind is generally played as a puzzle game with friends or family, there are few ethical concerns. Fundamentally, it is a fair game, as the role of the code-maker is relatively passive; they can only provide honest feedback to the guesses made by the code-breaker. However, playing Mastermind in a competition may bring ethical considerations. When prizes are introduced, the stakes are higher, so players must show good sportsmanship, engage in fair play and adhere to competition rules. Further to this, the competition should also be structured in a way that gives equal opportunity to all participants (Knight, 1923).

Ethics can also be considered for the report itself. It is vital to ensure all ideas are credited and correctly referenced.

This report explores computational methods for breaking a mastermind code. After assessing the performance of different algorithms, the aim is to investigate how this can influence strategies for the game when played in real life.

1.3 Adapting for Code and Calculations

1.3.1 Representations

In order to interpret the game in Python, the codes are usually processed at lists, but can also be tuples or sets depending on the function. The colours are represented by numbers. This makes it easier to review certain conditions, such as if two colours are in the same position - the algorithm can simply check if the numbers are equal to each other.

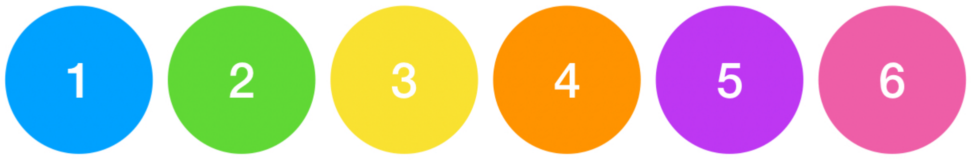

Figure 2: Numbers representing coloured pegs

1.3.2 Notation

Notation	Meaning
x	Number of pegs
y	Number of colours
$\mathrm{MM}(x, y)$	A Mastermind game with x pegs and y colours
$\mathrm{A}(x, y)$	All possible codes with x pegs and y colours
C	Hidden code
G_n	n^{th} guess
f_n	$(B, W) \rightarrow$ Feedback response for n^{th} guess, with B black pegs and W white pegs
$\mathrm{F}(x)$	All possible feedback combinations with x pegs
S_n	Set of possible codes after G_n and f_n
$\mathbb{E}(x, y)$	Expected number of guesses for a Mastermind game with x pegs and y colours
$\sigma(4, 6)$	Standard deviation of guesses for a Mastermind game with x pegs and y colours

Table 1: Notation

1.3.3 Proofs and Values for MM(4,6)

Claim: $|\mathrm{A}(x, y)| = y^x$
Proof: There are x pegs so a colour must be chosen x times. Since repetitions are allowed, there are y choices for each peg. Hence $\mathrm{A}(x, y) = y * y * ... * y$, x times. Therefore, —$\mathrm{A}(x, y)| = y^x$ $\square$

Claim: $|\mathrm{F}(x)| = \frac{x(x+3)}{2}$
Proof: (Ville, 2013, p.3). To set up this proof, it must be noted that $\forall f \in \mathrm{F}(x)$

$$B_f + W_f \leq x$$

meaning that the total number of black and white feedback pegs given cannot exceed the number of pegs in the code. It is important to notice that the combination $(x-1, 1)$ is not valid as, if $x-1$ pegs are correct there is only one peg left to fill. By definition, this cannot correspond to a white feedback peg because if it is the right colour, is must also be in the right place.

To find the number of solutions, a slack variable s is introduced to represent some arbitrary, positive value. The equation becomes

$$B_f + W_f + s = x$$

which can be solved using the formula for choosing k elements from a set of n elements if repetitions are allowed:

$$C(n+k-1,k) = \binom{n+k-1}{k} = \frac{n+k-1!}{k!(n-1)!} \tag{1}$$

so $n = 3$ (three variables B_f, W_f and s) and $k = x$. Then to find the number of feedback possibilities (1) is used, and one is subtracted, to account for the invalid option $(x-1,1)$.

$$\begin{aligned}
|\mathrm{F}(x)| &= C(3+x-1,x)-1 \\
&= C(x+2,x)-1 \\
&= \frac{(x+2)!}{x! \times 2!} - 1 \\
&= \frac{x!(x+1)(x+2)}{x! \times 2} - 1 \\
&= \frac{(x+1)(x+2)-2}{2} \\
&= \frac{x^2+3x+2-2}{2} \\
&= \frac{x^2+3x}{2} \\
&= \frac{x(x+3)}{2} \text{ as required.}
\end{aligned}$$

□

Using these formulas to calculate the values for MM(4,6) gives:

- $\mathrm{A}(4,6) = 6^4 = 1296$
- $\mathrm{F}(4) = \frac{4(4+3)}{2} = \frac{4(7)}{2} = \frac{28}{2} = 14$

Calculating these values is a useful check to ensure the lists of possible codes or feedback are the correct length.

2 Preliminary Functions

This section introduces functions that are required to simplify processes when implemented in the full method. Also, recall the notation described in Table 1.

Function	Input	Output
`possible_codes`	x, y	List of all possible codes for x pegs and y colours, A(x, y)
`random_code`	x, y	Randomly chosen code from possible codes
`feedback`	C, G_n	Feedback for guess, f_n
`possible_feedback`	x	List of all possible feedback for x pegs, F(x)
`is_guess_correct`	f, x	`True` if correct for x pegs, `False` otherwise
`find_poss_codes`	S_{n-1}, G_n, f_n	List of remaining possible codes after n^{th} guess, S_n

Table 2: Preliminary functions

2.1 Counting feedback

The `count_feedback` function is necessary in every function but requires some more explanation than the functions in the table above. The purpose of the function is to count the frequency of possible feedback responses (F(x)), as generated by the remaining possible codes (S). It uses the dictionary feature in Python which stores related data in the format key: value.

```
count_feedback

1. Take inputs of S, a guess, and F(x).
2. Initialise feedback_count as a dictionary, where the keys are the different feedback
   responses and the values are their corresponding frequency.
3. Start iterating through codes in S.
     - Find feedback, f, for the guess with the current code from S.
     - Increase the corresponding feedback count by one.
4. When each code has been used, return feedback_count.
```

3 Algorithms Solving MM(4,6)

This chapter focuses on three different algorithms: Minimax (Five Guess), Maximum Partitions and Maximum Entropy. They were chosen based on results found in Defeating Mastermind by Justin Dowell (Dowell, 2009). Each final solving function takes four variables as inputs: hidden code (C), G_1, x and y. They all output the number of guesses taken to solve code C.

It was necessary to include the initial guess as an input to minimise the code's running time when recording the number of guesses for all 1296 codes.

To automate this process and find how many guesses it takes to solve different hidden codes, the method functions for each algorithm. include a while loop to continue if `is_guess_correct` returns `False` . From here, S_0 will denote the list of all codes.

General method function

1. Find all possible codes (S_0) and feedback responses for use in functions.
2. Find feedback, f_1, for the initial guess, G_1, and check if it is correct, set the guess count to one.
3. Find the remaining possible codes, S_1, using S_0.
4. Start while loop for when the guess is not correct.
   - Find G_2 with the appropriate 'next guess' function.
   - Find f_2 and check if the guess is correct. Increase the guess count by one.
   - Find S_2, using S_1.
5. When the guess is correct, return the guess count.

The following sections outline how the different algorithms aim to minimise the number of guesses to find C. Then, the example where $C = 1256$ will be shown in each case. All three algorithms solve this code with G_3, but the first two guesses differ.

3.1 Minimax Algorithm

The Minimax - or Five Guess - Algorithm was published by Donald Knuth in the Journal of Recreational Mathematics. He outlines a method that is guaranteed to solve any MM(4,6) in five or fewer moves by the code-breaker. It iterates through all codes (S_0) to consider which will minimise the maximum possible codes (S_n) after the n^{th} guess. Therefore, sometimes it will guess a code which isn't a possible answer. This is because sometimes an impossible code will be more useful to differentiate between possible answers. (Knuth, 1976)

3.1.1 Deciding the Next Guess

The function `minimax_next_guess` is used to determine the initial guess or the next guess for each of the code-breaker's turns. It takes inputs of the possible feedback responses, S_0 and S_{n-1} (for the initial guess this is equal to S_0). It calls the function `count_feedback` for every code in S_0. The frequencies of each feedback response will be called 'hits'. The table below shows how many hits 1122 would result in as the initial guess.

f_1	Hits
0, 0	256
0, 1	256
0, 2	96
0, 3	16
0, 4	1
1, 0	256
1, 1	208
1, 2	36
1, 3	0
2, 0	114
2, 1	32
2, 2	4
3, 0	20
4, 0	1

Table 3: Example of counting feedback ‘hits’ for initial guess - 1122

The function finds that the maximum hits is 256. This value represents the maximum size that S could be reduced to if this code is guessed. This repeats for every code in A, saving the maximum for each one. Next, the minimum of the maximum hits is found.

Initial guess	Minimax score
1111	625
1112	317
1122	256
1123	276
1234	312

Table 4: Minimax scores for possible initial guesses

In this initial case, 256 hits for the guess 1122 is the minimum. So, 1122 is the optimal guess at this stage. It minimises the maximum number of remaining possible codes, meaning in the worst case there will only be 256 codes remaining. This is advantageous in terms of a process of elimination - when possible codes are discounted, the algorithm is closer to discovering the correct one (Forsyth, 2018).

In some cases, there will be more than one code that achieves the Minimax value. Knuth states “If this minimum can be achieved by a ‘valid’ pattern (making ‘4 black hits’ possible), a valid one should be used” (Knuth, 1976, p.3). Therefore, the algorithm first checks if any of the ‘minimum’ codes are also in the previous possible codes (the inputted S). The choice of code from here is arbitrary as they all achieve the same minimum. If a ‘valid’ code exists, the one with the lowest numerical value is chosen. If not, the algorithm goes back to list of all ‘minimum’ codes and chooses the lowest numerically of those.

3.1.2 An Example Game

Below is an example game showing which guesses the Minimax Algorithm makes and their feedback.

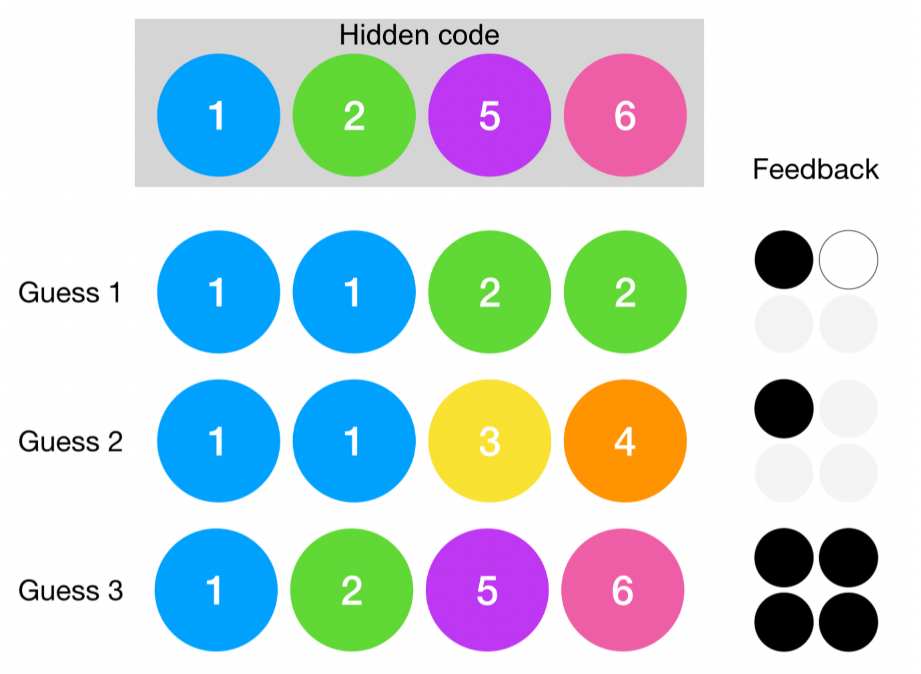

Figure 3: Minimax Algorithm game

The following breaks down what the algorithm calculates after each guess.

Guess 1

As shown above, 1122 is the initial guess indicated by the Minimax Algorithm. The function `minimax_initial_guess` applies the algorithm to A to find the most efficient first guess.

```
In: minimax_initial_guess(4, 6)
Out: (1, 1, 2, 2)
```

Then `minimax_method` begins. Set guess count to 1.

```
Out: guess_count = 1
```

After making the initial guess, `feedback` compares this to the hidden code and reveals 1 back peg and 1 white peg.

```
In: feedback((1, 2, 5, 6), (1, 1, 2, 2))
Out: [1, 1]
```

Then check if this feedback indicates a correct guess with `is_guess_correct`.

```
In: is_guess_correct([1, 1],4)
Out: False
```

Then use `find_poss_codes` to find S. This discounts any codes that would not generate this feedback i.e only leaves codes with 1 number the same and in the same position, and 1 number the same but in a different position. This reduces the possible codes from 1296 to 208.

```
In: find_poss_codes(A, (1, 1, 2, 2), [1,1])
Out: [(1, 2, 3, 3), (1, 2, 3, 4), (1, 2, 3, 5), (1, 2, 3, 6), ... ,
      (6, 5, 1, 2), (6, 5, 2, 1), (6, 6, 1, 2), (6, 6, 2, 1)]
```

Guess 2

Guess 1 was incorrect, so `minimax_next_guess` finds the next move to make, using S calculated above. In this case the algorithm selects an impossible code (i.e not in S).

```
In: minimax_next_guess(PossF, A, S)
Out: [1, 1, 3, 4]
```

Increase guess count by 1.

```
Out: guess_count = 2
```

Calculate feedback for Guess 2.

```
In: feedback((1, 2, 5, 6), (1, 1, 3, 4))
Out: [1, 0]
```

Check if this feedback indicates a correct guess.

```
In: is_guess_correct([1, 0],4)
Out: False
```

Find S. This reduces the possible codes from 208 to 22.

```
In: find_poss_codes(A, (1, 1, 2, 2), [1,1])
Out: [(1, 2, 5, 5), (1, 2, 5, 6), (1, 2, 6, 5), (1, 2, 6, 6), ... ,
      (5, 2, 2, 4), (5, 2, 3, 2), (6, 2, 2, 4), (6, 2, 3, 2)]
```

Guess 3

Guess 2 was incorrect, so the while loop continues. In this case the algorithm selects an possible code (i.e is in S).

```
In: minimax_next_guess(PossF, A, S)
Out: [1, 2, 5, 6]
```

Increase guess count by 1.

```
Out: guess_count = 3
```

Calculate feedback for Guess 3.

```
In: feedback((1, 2, 5, 6), (1, 2, 5, 6))
Out: [4, 0]
```

Check if this feedback indicates a correct guess.

```
In: is_guess_correct([4, 0],4)
Out: True
```

The guess is correct so the function returns the guess count

```
Out: 3
```

3.2 Maximum Partitions Algorithm

The Maximum Partition method was suggested by Barteld Kooi in 2005. It follows similar principles to the Minimax method as it also analyses how S_n is partitioned by different feedback. However, while the Minimax Algorithm concentrates on the **size** of the partitions, this method focuses on the **number** of partitions (Kooi, 2005).

3.2.1 Deciding the Next Guess

When considering the maximum partitions, `count_feedback` is used again to obtain the number of hits for each feedback response. Below, the table shows the hits for the code 1123 as the initial guess.

f_1	Hits	f_1	Hits
0,0	81	1,2	84
0,1	276	1,3	4
0,2	222	2,0	105
0,3	44	2,1	40
0,4	2	2,2	5
1,0	182	3,0	20
1,1	230	4,0	1

Table 5: Example of counting feedback 'hits' for initial guess - 1123

The function finds that every possible feedback combination receives at least one hit. This indicates that the set, S, will be partitioned into 14 - the maximum, as there are 14 different feedback possibilities. Once all the codes that achieve this maximum are found, again, the choice is arbitrary so the one with the lowest numerical value is chosen.

If the number of partitions is maximised, by default the possible the size of those partitions is reduced. For example, if 52 playing cards were divided into three partitions, the average size of them would be about 17 in each. This compared to if there were only two partitions, where the average size would optimally be 26, shows how increasing partitions can narrow down possibilities.

Of course these are ideal partition sizes, and there is still a possibility the result could be a very large partition. Table 5 shows that the sizes of partitions generated by the guess 1123 range from containing 1 code to 276 codes - but the expected value $\left(\frac{1296}{14}\right)$ is about 93.

Initial guess	Partitions
1111	5
1112	11
1122	13
1123	14
1234	14

Table 6: Number of partitions for possible initial guesses

As shown in Table 6, both 1123 and 1234 generate the maximum 14 partitions - so 1123 should be chosen as the first guess. Unlike the Minimax Algorithm, it makes no difference here to choose a valid or invalid code. This may be because the guesses that maximise partitions are always valid.

3.2.2 An Example Game

Below is an example game showing which guesses the Max Parts Algorithm makes and their feedback.

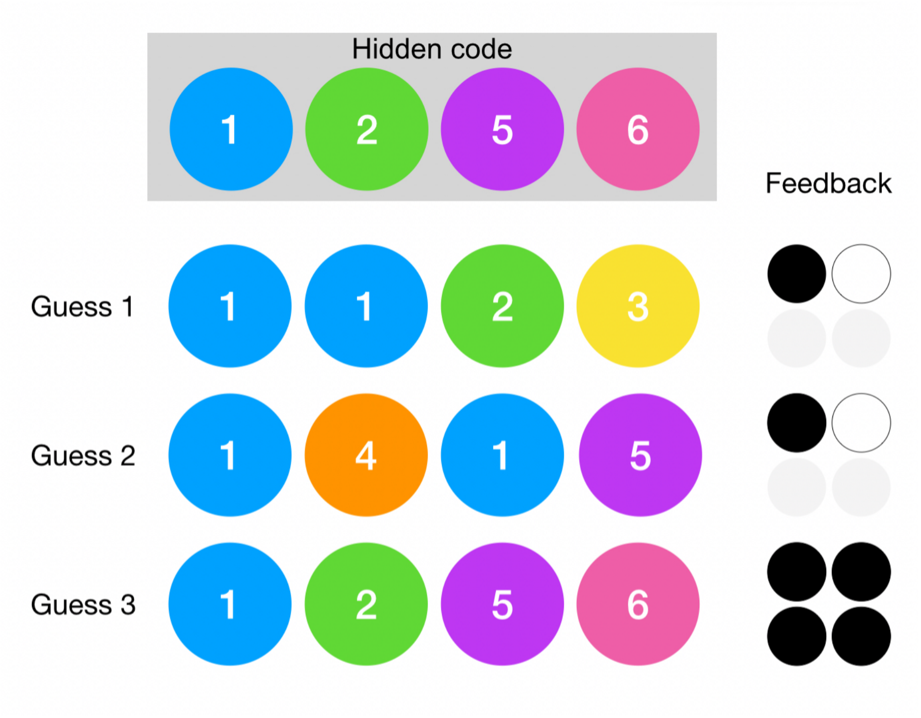

Figure 4: Maximum Partitions Algorithm game

The following breaks down what the algorithm calculates after each guess.

Guess 1

As shown above, 1123 is the initial guess indicated by the Max Parts Algorithm. The function `max_parts_initial_guess` applies the algorithm to A to find the most efficient first guess.

```
In: max_parts_initial_guess(4, 6)
Out: (1, 1, 2, 3)
```

Set guess count to 1.

```
Out: guess_count = 1
```

After making the initial guess, `feedback` compares this to the hidden code and reveals 1 back peg and 1 white peg.

```
In: feedback((1, 2, 5, 6), (1, 1, 2, 3))
Out: [1, 1]
```

Next, the feedback is used to check if it indicates a correct guess with `is_guess_correct`.

```
In: is_guess_correct([1, 1],4)
Out: False
```

Then, `find_poss_codes` is used to find S. This discounts any codes that would not generate this feedback i.e only leaves codes with 1 number the same and in the same position. This reduces the possible codes from 1296 to 230.

```
In: find_poss_codes(A, (1, 1, 2, 3), [1,1])
Out: [(1, 2, 4, 2), (1, 2, 4, 4), (1, 2, 4, 5), (1, 2, 4, 6), ... ,
      (6, 5, 1, 3), (6, 5, 2, 1), (6, 6, 1, 3), (6, 6, 2, 1)]
```

Guess 2

Finally, `max_parts_next_guess` is used to find the next move to make, using S calculated above.

```
In: maxt_parts_next_guess(PossF, A, S)
Out: [1, 4, 1, 5]
```

Increase guess count by 1.

```
Out: guess_count = 2
```

Calculate feedback for Guess 2.

```
In: feedback((1, 2, 5, 6), (1, 4, 1, 5))
Out: [1, 1]
```

Check if this feedback indicates a correct guess.

```
In: is_guess_correct([1, 1], 4)
Out: False
```

Find S. This reduces the possible codes from 230 to 39.

```
In: find_poss_codes(S, (1, 4, 1, 5), [1, 1])
Out: [(1, 2, 4, 2), (1, 2, 4, 4), (1, 2, 4, 6), (1, 2, 5, 2), ... ,
      (6, 1, 1, 6), (6, 1, 3, 5), (6, 4, 2, 1), (6, 5, 1, 3)]
```

Guess 3

Find the next guess to make.

```
In: max_parts_next_guess(PossF, A, S)
Out: [1, 2, 5, 6]
```

Increase guess count by 1.

```
Out: guess_count = 3
```

Calculate feedback for Guess 3.

```
In: feedback((1, 2, 5, 6), (1, 2, 5, 6))
Out: [4, 0]
```

Check if this feedback indicates a correct guess.

```
In: is_guess_correct([4, 0],4)
Out: True
```

The guess is correct, so the function returns the guess count

```
Out: 3
```

3.3 Maximum Entropy Algorithm

Entropy is a concept used in information theory to measure the average level of information associated with a particular outcome. It was first defined by Claude Shannon in 1948 (Shannon, 1948). Firstly, the *Shannon information constant*, $I(x_i)$, for an event, x_i, indicates the level of surprise, i.e. how unlikely the event is.

Intuitively, if $p(x_i)$ is how likely an event is, then the reciprocal of that, $\frac{1}{p(x_i)}$, is how unlikely the event is. Then, Shannon outlines the three reasons it is convenient to use the logarithm of this value:

$$I(x_i) = \log_2 \left[\frac{1}{p(x_i)}\right] \tag{2}$$

1. Practicality
   While $p(x_i)$ is bounded between 0 and 1, the reciprocal is not bounded above. It is clear that if the probability is infinitely small, then $\frac{1}{p(x_i)}$ is infinitely large. The **log** operation compresses these values into a more manageable scale.

2. Intuition
   A real-life example of a similar logarithmic scale is the lottery. Matching all the numbers and winning the lottery has a low probability, and would be an extremely surprising event. However, matching one number has a high probability and would not be surprising. Similarly, it would not be much more surprising to match two numbers. This can be characterised by a logarithmic scale, where the level of surprise increases disproportionately as the probabilities of the events decrease.

3. Suitability
   Taking the logarithm with base two means that the units are bits, or binary digits - the most basic unit of information in computing. This compresses the data, which improves efficiency. An example could be guessing a card from eight possibilities with 'yes/no'questions. Since $\log_2(8) = 3$, the limit of the expected number of questions to determine the card is three. Logarithms can also be easier to manipulate mathematically, as shown below.

$$I(x_i) = \log_2 \left[\frac{1}{p(x_i)}\right] = \log_2 \left[(p(x_i))^{-1}\right] = -\log_2 \left[p(x_i)\right] \tag{3}$$

Entropy, denoted by $H(X)$ is defined as the *average information content* and is derived from the expected value of $I(X)$:

$$H(X) = \mathbb{E}[I(X)] = \sum_{i=1} p(x_i) \cdot (-\log_2[p(x_i)]) = -\sum_{i=1} p(x_i) \cdot \log_2[p(x_i)] \tag{4}$$

The following proves a property of entropy:

Claim: $H(X) \geq 0$ (Mackay, 2003)
Proof: It can be shown that this is true by proving that H(X) is never negative.

It is known that $p(x_i)$ is bounded between 0 and 1 so:

$$0 \leq p(x_i) \leq 1$$

Then $\log_2(p(x_i))$ is always negative.

Since $0 \leq p(x_i) \leq 1$ it always positive or 0, so:

$$\begin{aligned} p(x_i) \cdot \log_2[p(x_i)] &\leq 0 \\ \sum_{i=1} p(x_i) \cdot \log_2[p(x_i)] &\leq 0 \\ -\sum_{i=1} p(x_i) \cdot \log_2[p(x_i)] &\geq 0, \end{aligned}$$

as $p(x_i) \times log_2(p(x_i))$ is also negative. Then, the sum of these values will be negative. Finally, the negative of the sum is taken, so $H(X)$ is always positive, or 0. □

N.B. In the case where $p(x_i) = 0$, although $I(x_i)$ is undefined, it results $0 \times \log_2(0) \equiv 0$ when calculating H(X).

3.3.1 Deciding the Next Guess

When calculating the entropy of the n$^{\text{th}}$ guess in Mastermind, `count_feedback` is used again to determine the hits. Then,

$$p(x_i) = \frac{\text{No. of Hits}}{\text{Length of } \mathrm{S}_{n-1}}$$

where $x_i \in \mathrm{F}(x)$ is a possible feedback response (Neuwirth, 1982).

$\mathbf{f_1}$	**Hits**	$p(x_i)$	$p(x_i) \cdot \log_2[p(x_i)]$
$0,0$	16	0.012	-0.078
$0,1$	152	0.117	-0.363
$0,2$	312	0.241	-0.495
$0,3$	136	0.105	-0.341
$0,4$	9	0.007	-0.050
$1,0$	108	0.083	-0.299
$1,1$	252	0.194	-0.459
$1,2$	132	0.102	-0.336
$1,3$	8	0.006	-0.036
$2,0$	96	0.074	-0.093
$2,1$	48	0.037	-0.177
$2,2$	6	0.005	-0.036
$3,0$	20	0.015	-0.046
$4,0$	1	0.001	-0.008

Entropy	3.057

Table 7: Example of counting feedback ‘hits’ and calculating entropy for initial guess - 1234

Table 7 shows how Equation 4 is used to calculate entropy for the initial guess 1234. Working with such small numbers here means that the rounding does affect the sum. In the algorithm, this will be repeated for each possible guess in S_{n-1}.

Initial guess	**Entropy**
1111	1.498
1112	2.693
1122	2.885
1123	3.044
1234	3.057

Table 8: Entropy for possible initial guesses

When testing the initial guesses as shown above in Table 8, it is found that the code 1234 maximises entropy.

3.3.2 An Example Game

Here is an example of a Mastermind game implementing the Maximum Entropy Algorithm.

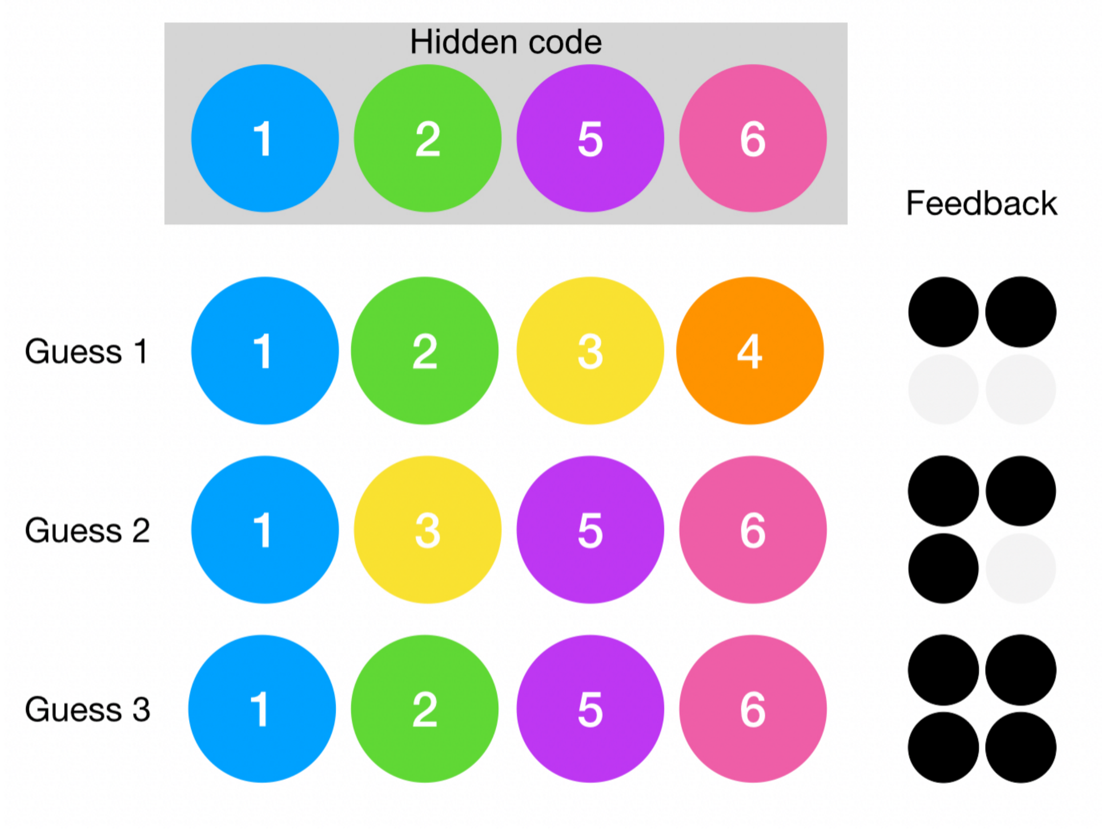

Figure 5: Maximum Entropy Algorithm game

The following breaks down what the algorithm calculates after each guess.

Guess 1

As shown above, 1234 is the initial guess indicated by the Maximum Entropy Algorithm. The function `max_ent_initial_guess` applies the algorithm to A to find the most efficient first guess.

```
In: max_ent_initial_guess(4, 6)
Out: (1, 2, 3, 4)
```

Set guess count to 1.

```
Out: guess_count = 1
```

After making the initial guess, `feedback` compares this to the hidden code and reveals 2 black pegs and no white pegs.

```
In: feedback((1, 2, 5, 6), (1, 2, 3, 4))
Out: [2, 0]
```

Then check if this feedback indicates a correct guess with `is_guess_correct`.

```
In: is_guess_correct([2, 0],4)
Out: False
```

Then use `find_poss_codes` to find S. This discounts any codes that would not generate this feedback i.e only leaves codes with 2 numbers the same and in the same position. This reduces the possible codes from 1296 to 96.

```
In: find_poss_codes(A, (1, 2, 3, 4), [2,0])
Out: [(1, 1, 1, 4), (1, 1, 3, 1), (1, 1, 3, 3), (1, 1, 3, 5), ... ,
      (6, 3, 3, 4), (6, 4, 3, 4), (6, 5, 3, 4), (6, 6, 3, 4)]
```

Guess 2

Guess 1 was incorrect, so `max_ent_next_guess` is used to find the next move to make, using S calculated above. In this case the algorithm selects an impossible code (i.e not in S).

```
In: max_ent_next_guess(PossF, A, S)
Out: [1, 3, 5, 6]
```

Increase guess count by 1.

```
Out: guess_count = 2
```

Calculate feedback for Guess 2.

```
In: feedback((1, 2, 5, 6), (1, 3, 5, 6))
Out: [3, 0]
```

Check if this feedback indicates a correct guess.

```
In: is_guess_correct([3, 0],4)
Out: False
```

Find S. This reduces the possible codes from 96 to 2.

```
In: find_poss_codes(A, (1, 3, 5, 6), [3, 0])
Out: [(1, 2, 5, 6), (1, 3, 3, 6)]
```

Guess 3

Guess 2 was incorrect, so the while loop continues. In this case the algorithm selects an possible code (i.e is in S).

```
In: minimax_next_guess(PossF, A, S)
Out: [1, 2, 5, 6]
```

Increase guess count by 1.

```
Out: guess_count = 3
```

Calculate feedback for Guess 3.

```
In: feedback((1, 2, 5, 6), (1, 2, 5, 6))
Out: [4, 0]
```

Check if this feedback indicates a correct guess.

```
In: is_guess_correct([4, 0],4)
Out: True
```

The guess is correct so the function returns the guess count

```
Out: 3
```

4 Comparing the data

After running the three algorithms from Chapter 3 for MM(4,6), the results shown below in Table 9 are found.

Algorithm	**Games completed in N guesses**					
	1	**2**	**3**	**4**	**5**	**6**
Minimax	1	6	62	533	694	0
Max Parts	1	12	72	635	569	7
Max Ent.	1	4	71	612	596	12

Table 9: Games Completed in N Guesses

From this data, the expected value and standard deviation can be calculated in each case.

Algorithm	First Guess	Max Guesses	$\mathbb{E}(4,6)$	$\sigma(4,6)$
Minimax	1122	**5**	4.476	**0.618**
Max Parts	1123	6	**4.373**	0.649
Max Ent.	1234	6	4.415	0.631

Table 10: Algorithm Performance

From Table 10, it can be seen that the expectations and standard deviations are generally very similar. The range of $\mathbb{E}(4,6)$ is 0.103 and the range of $\sigma(4,6)$ is 0.031.

If the focus is to guarantee fewer guesses, the Minimax Algorithm can ensure a win within five. This also leads to Minimax having the lowest standard deviation, as the data is spread over fewer guesses. However, referring back to Table 9, Minimax completes its greatest number of games on G_5, while the other two have their maximum completions on G_4. This is reflected by their lower expected values. Therefore, when aiming to minimise the number of guesses on average, Max Parts should be used as it has the lowest expected value.

Reflecting on the ways to play outlined in Section 1.1, it is clear that the traditional 10 guesses are more than enough to solve the code computationally. If the aim is to win a single game in five guesses, Minimax will be the best strategy. However, if multiple games are played and points are given away for every guess made, Most Parts would be the best procedure to ensure minimal guesses are made.

In summary, all three of these computational methods perform very well in the MM(4,6) game. The choice of algorithm will depend on what is being optimised in the game.

5 Practical Methods

5.1 Random/Simple Guess

These algorithms could be described as most similar to how the game might be played in real life. Intuitively, when a person plays Mastermind, they will most likely use the previous feedback to inform each guess. The player will choose an arbitrary guess that might produce the same feedback as their previous guess, which can be described as a Random guess. The results below have been found by repeating the random guess method for each code ten times and recording the guess count for each.

In order to keep the results consistent, this method can be adapted to always choose the minimum guess, i.e. code with the lowest numerical value. This is a Simple guess, as Shapiro named it when he first outlined the algorithm. It uses the previous guess and feedback to narrow down the code possibilities with `find_poss_codes` and chooses the first guess from that list, i.e. the code with the lowest numerical value (Shapiro, 1983).

Algorithm	First Guess	Max Guesses	$\mathbb{E}(4,6)$	$\sigma(4,6)$
Random	N/A	8	4.649	0.884
Simple	1111	9	5.765	1.049

Table 11: Random and Simple Algorithm Performance

The statistics for the random method were obtained by running the method 10 times and using that data to calculate $\mathbb{E}$ and σ. As the choices are random, these results are not entirely reproducible. Therefore, they cannot reflect the data as fairly as they do for the Simple Algorithm, for which the values will always be replicated. This data is suitable for the scope of this project, but future work could include testing the method over more trials to reduce uncertainty in these results.

When evaluating the results available, it becomes clear that there is little benefit to implementing the Simple Algorithm. It can be argued that a random choice is more effective in reducing both the expected and maximum number of guesses. In particular, the expected value for the Random Algorithm better aligns with the values found in Section 4.

5.2 An Alternative Approach

The descriptions in Section 3 show that the Minimax, Max Parts, and Max Ent. methods would not be easy to apply in a real-life game. However, the random and simple methods do not perform as well. To address this, the table below outlines how fixing the first two guesses based on initial guesses Section 3 and then implementing the Simple Algorithm could improve human strategies.

Algorithm	First Guess	Second Guess	Max Guesses	$\mathbb{E}(4,6)$	$\sigma(4,6)$
Minimax	1122	3344	7	4.823	0.847
Max Parts	1123	4456	7	4.857	0.852
Max Ent.	1234	3456	7	4.678	0.856
Minimax + Max Parts	1122	3345	7	4.822	0.846
Minimax + Max Ent.	1122	3456	7	4.894	0.841
Max Parts + Max Ent.	1123	3456	7	**4.664**	**0.808**

Table 12: Two Fixed Guesses Algorithm Performance

Table 12 clearly shows that by simply presetting the first two guesses, the performance of the Simple Algorithm significantly improves. This is beneficial as the consistency of this method allows for reliable outcomes. If the aim is to guarantee a win by G_7 and expect a win by G_5, the Two Fixed Guesses approach provides more certainty than the Random Algorithm.

On the other hand, a human strategy could include two fixed guesses followed by random guesses. While this does increase uncertainty, the statistics calculated in Table 11 show that an arbitrary code may lead to solving C in fewer guesses. The combinations in Table 12 suggest that the pairing of optimal first guesses for Max Parts and Max Ent. produces the most favourable results. All yielded a maximum of seven guesses, which does increase the standard deviation.

In summary, fixing the first two guesses is easy to implement and clearly improves human strategy. However, the above methods could still be improved to achieve optimal results in a real-life game.

6 Conclusions and Future Work

How does automatic puzzle solving contribute to mastering Mastermind?

This report found that there are multiple ways to solve a Mastermind code with computational methods. Minimax, Maximum Partitions and Maximum Entropy algorithms all have different strengths. However, they are impractical when trying to implement in a real-life game. The Random and Simple algorithms are more accessible for human players and, together with two fixed guesses, can produce comparable results.

One of the main benefits of automatic puzzle solving is the program can automate processes to evaluate different possibilities in a much shorter period than a human could. While this may not be able to be replicated in a game, the results can act as a guide for the best strategy. Hence, investigating the performance of these methods can inform players of Mastermind to make more efficient guesses.

Future work could improve upon these strategies. This may include:

- Conducting more trials for the Random Algorithm to reduce uncertainty in the results for $\mathbb{E}(4, 6)$ and $\sigma(4, 6)$
- Investigating why the Maximum Parts Algorithm does not differ when valid codes are prioritised (Section 3.2.1).
- Combining algorithms. When a point is reached where an arbitrary choice must be made, it may be beneficial to use another method to differentiate. This investigation could also test which order of implementation achieves the best results.
- Construct an automated process to find the most informative first two guesses. The options in Section 5.2 are simply the lowest numerical codes that maximise different digits. This may still be the most efficient, but could be checked.
- These methods could also be tested for various values of x and y. There are many similar games, such as Logik which has 5 pegs and 8 colours (Heeffer and Heeffer, 2014).

References

Alza.cz (n.d.). Woody wooden logic game.
URL: *https://image.alza.cz/products/HRAif10735/HRAif10735.jpg?width=1000height=1000*

Dowell, J. (2009). Defeating mastermind.
URL: *http://mercury.webster.edu/aleshunas/Support%20Materials/Analysis/Dowelll%20-%20Mastermind%20v2-0.pdf*

Forsyth, A. (2018). Rubyconf 2018 - beating mastermind: Winning with the help of donald knuth by adam forsyth.
URL: *https://www.youtube.com/watch?v=Okm*$_t5T1PiA$

Heeffer, A. and Heeffer, H. (2014). Near-optimal strategies for the game of logik.

Knight, F. H. (1923). The ethics of competition *The Quarterly Journal of Economics* **37**, 590–591.
URL: *https://doi.org/10.2307/1884053*

Knuth, D. E. (1976). The computer as master mind *Journal of Recreational Mathematics* **9(1)**, 1–6.
URL: *http://colorcode.laebisch.com/links/Donald.E.Knuth.pdf*

Kooi, B. (2005). Yet another mastermind strategy *ICGA Journal* **28**.

Mackay, D. J. C. (2003). *Information Theory, Inference, and Learning Algorithms* Cambridge University Press.

Neuwirth, E. (1982). Some strategies for mastermind *Zeitschrift für Operations Research* **26**, B257–B278.
URL: *https://link.springer.com/article/10.1007/BF01917147*

Shannon, C. E. (1948). A mathematical theory of communication *Bell System Technical Journal* **27**, 379–423.
URL: *https://dl.acm.org/citation.cfm?id=584093*

Shapiro, E. (1983). Playing mastermind logically *ACM SIGART Bulletin* **85**, 28–29.

Ville, G. (2013). An optimal mastermind (4,7) strategy and more results in the expected case.
URL: *https://arxiv.org/pdf/1305.1010.pdf*