Mastermind Solver

Arjun Bhushan
CID:01511448

Background and Task

Mastermind is a code-breaking game that was developed in 1970 by Mordecai Meirowitz. It is a two player game , where the first is the ‘code maker’ and the second is the ‘codebreaker’. The 'code maker’ would choose a code of length 4 which could be 6 different colours - duplicates were allowed. The ‘code breaker’ would then try to guess the code in as few attempts as possible. With each attempt, the ‘code maker’ would give the ‘code breaker’ feedback in the form of pegs. A black peg would indicate that the one of the 4 elements the ‘code breaker’ has chosen was the correct colour and in the correct place, while a white peg would indicate the presence of the right colour in the wrong place. Since the original game 1970, there have been over 20 variations of the game excluding the internet versions. This report summarises several algorithms that can be used to solve a general game with a code of length n and n different colours (where n is any real number between 1 and 15 in minimal attempts and in less than 10 seconds on average.

Give Feedback

The first main function of the program, takes an input attempt, compares it with the sequence generated by the computer - and outputs the amount of black hits and white hits the attempt obtains. It plays the role of the ‘code maker’ after the ‘code breaker’ inputs an attempt.

To calculate the number of black hits, we do not need to create any extra variables, we can just use a simple ‘for’ loop for this - as a black hit is only obtained when the two have the identical values at the same index.

For white hits, a large concern was that when calculating the white hits, the number of black hits would also be included in the result - to prevent this we can create a separate variable that stores to total number of hits , and then calculate the number of white hits by subtracting the black hits from this value. To calculate the number of total hits , a temporary vector that matches the same values of the code generated by the computer is made. 2 for loops are then utilised so that each element of the two vectors are compared against each other, if two elements are equal, the value of the temporary vector is changed to ‘-1’ at that index. The boolean ‘hasalreadybeenhit’ also changes to true so that the same index isn’t counted twice - the variable for the total number of hits then increases by 1. As mentioned before, the white hits is calculated by subtracting the value of black hits from the from the value of the total hits. Whenever the function is called, the values of the white hits and black hits is set to zero at the beginning.

Solver Structure

The solver structure is composed of two main functions ‘create_attempt’ and ‘learn’ .The create attempt function initialises the program with an attempt, and the learn function uses the results from ‘give_feedback’ to create an attempt closer to the true value. Within these, three different algorithms were included. Each algorithm is used to solve a different length of code. Below are the breakdowns of the three algorithms.

Knuth’s Elimination[1][2]

In 1977, Knuth demonstrated that the original mastermind could be completed in five moves or less using this algorithm. This does not use the minimax technique:

I. The first attempt is an incremental number with each digit increasing (for length 4 the attempt would be ‘0012’) as this provides a large amount of information as to what colours are still included in the final code. The number of black and white hits are recorded
II. Using a recursive function, a set of all possibilities is created, and only the codes that have the same black and whites hits relative to the previous attempt (this is done with the give_relativefeedback’ function) are ‘pushed back’ into a separate vector of vectors called ‘possibilities’.
III. The next attempt is then a randomly chosen vector from the ‘possibilities’ vector, and its feedback is given in the form of black and white hits.
IV. We then copy the the contents of ‘possibilities’ a temporary vector, whilst we clear the contents of ‘possibilities’ - we then search the temporary vector for codes that have the same black and white hits relative to the previous attempt and push these back into the ‘possibilities’
V. The possibilities vector decrease with each turn and a random attempt is chosen from the list until all black hits are achieved.

This solved smaller length codes in short times and in little attempts, however due to larger lengths having larger possibilities (possibilities = number of colours^ length), a greater set is required , thus this algorithm breakdown at larger length codes with more colours. Below are two examples:

```
enter length of sequence and number of possible values:
4
6
attempt:
0 0 1 2
black pegs: 0  white pegs: 1
attempt:
2 2 2 4
black pegs: 2  white pegs: 0
attempt:
4 4 2 4
black pegs: 1  white pegs: 2
attempt:
5 2 4 4
black pegs: 4  white pegs: 0
the solver has found the sequence in 4 attempts
the sequence generated by the code maker was:
5 2 4 4 And in 0.000975775 seconds of time
```

```
enter length of sequence and number of possible values:
7
10
attempt:
0 0 1 2 3 4 5
black pegs: 0  white pegs: 4
attempt:
3 4 6 7 2 1 1
black pegs: 0  white pegs: 5
attempt:
2 3 4 8 1 6 8
black pegs: 1  white pegs: 4
attempt:
4 1 8 6 1 5 2
black pegs: 0  white pegs: 4
attempt:
2 8 3 1 6 0 7
black pegs: 4  white pegs: 2
attempt:
2 6 3 4 8 0 7
black pegs: 7  white pegs: 0
the solver has found the sequence in 6 attempts
the sequence generated by the code maker was:
2 6 3 4 8 0 7 And in 5.89171 seconds of time
```

[1] https://en.wikipedia.org/wiki/Donald_Knuth

[2] https://en.wikipedia.org/wiki/Mastermind_(board_game)#Five-guess_algorithm

Permuted Elimination

This algorithm first obtains all the correct elements in the code generated by the computer, but in a different arrangement (a permutation of the code). Knuth's Elimination is then used to narrow down the possibilities of the code. The initial part of this algorithm relies solely on the black hits.

I. The initial attempts are filled with the same element in each position (e.g. 0000, 1111, 2222 etc.) . T
II. Give feedback is carried out and the value of the elements in the vector are pushed back the amount of black hits obtained into a separate vector called 'permutatedcode' , (e.g 0000 produces 2 black hits, thus 2 0's are pushed back into 'permutatedcode'). We keep track of the total number of black hits that have occurred.
III. These incremented attempts continue until either all the numbers (or colours) have been tried, or the total number of black hits obtained equals the length of the code.
IV. After this, Knuths Elimination is applied to the code, and a set of possibilities is created based off the 'permutatedcode' is achieved and pushed back into 'possibilities'
V. Random attempts are chosen from this vector of vectors, and the size of the 'possibilities' decreases with each attempt.

This algorithm is good for handling medium sized codes , however when creating the initial set of possibilities, the program goes through every single possibility the code has to offer, before pushing back only the permutations of the 'permutatedcode' - thus making it harder to handle larger codes.

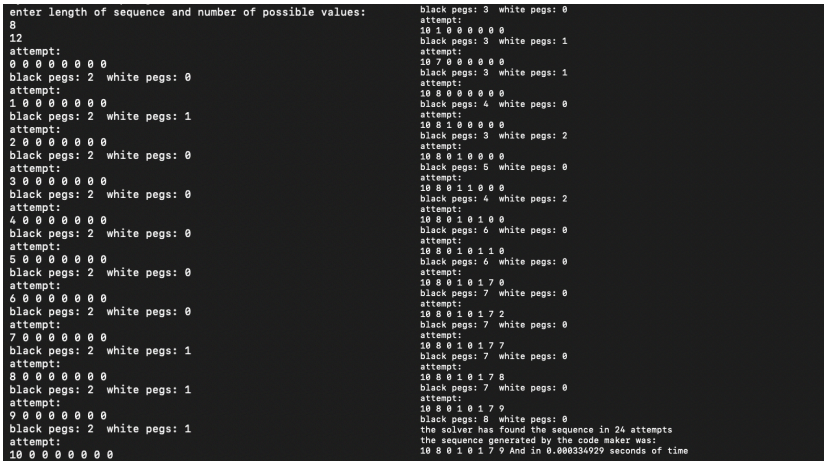
```
enter length of sequence and number of possible values:
4
6
attempt:
0 0 0 0
black pegs: 0  white pegs: 0
attempt:
1 1 1 1
black pegs: 1  white pegs: 0
attempt:
2 2 2 2
black pegs: 0  white pegs: 0
attempt:
3 3 3 3
black pegs: 0  white pegs: 0
attempt:
4 4 4 4
black pegs: 2  white pegs: 0
attempt:
5 5 5 5
black pegs: 1  white pegs: 0
attempt:
1 4 4 5
black pegs: 2  white pegs: 2
attempt:
1 5 4 4
black pegs: 1  white pegs: 3
attempt:
5 4 4 1
black pegs: 1  white pegs: 3
attempt:
4 1 4 5
black pegs: 4  white pegs: 0
the solver has found the sequence in 10 attempts
the sequence generated by the code maker was:
4 1 4 5
dyn3169-62:Desktop arjunbhushan$
```

Incremental

The algorithms solves the code by brute force and is most efficient when used for larger codes, as it uses a large amount of attempts for smaller codes.

I. The first attempt always fills the vector with 0's - with this we keep track of the initial amount of black hits.
II. We then increment the first number in the code , so 0000 would then become 1000 for a code of length 4.
III. The next step taken is dependent on the feedback received from the program:
    a) If the amount of black hit's hasn't changed the number is further incremented , fo 1000 would become 2000
    b) If the amount of black hits increases , the incremented number is correct, and we start to increment the next index - we again take note of the number of black hit's now
    c) If the number of black hits decreases, the previous number was attempt was correct, and we revert the code to the previous attempt , and then start incrementing the next index.
    d) We also look at the white hits. If the black hits stay the same, and the white hits either decrease or stay the same at the value of 0, we take this increment value out of the incrementing index. This means when any of the values are incremented for an attempt, the increment will skip the value we have just taken out.
IV. We repeat this process till we reach the end of the code length.

Below is an example split in two.



Testing

100 random codes were used to test the three different algorithms. In these tests, I recorded the largest number of attempts required, the smallest amount of attempts and the average amount. As well as this, the average time required to solve the code was calculated. A large spread of code lengths and number of elements, however, for many of the larger code lengths , a timeout error occurred, preventing data from being recorded. Below are the results obtained.

Test For Knuth's Elimination

Elimination					
Length	Number	Minimmum Amount of Attempts	Max Amount of Attempts	Average Amount of Attempts	Average time taken for one attempt
2	3	2	5	3	2.15428E-04
2	6	2	8	5	7.21724E-04
2	9	3	11	6	1.4689E-02
2	12	4	13	8	2.62332E-03
2	15	4	16	9	4.00948E-03
4	3	2	6	4	8.19582E-04
4	6	4	8	5	7.01616E-03
4	9	3	10	6	3.44503E-02
4	12	5	11	7	1.17509E-01
4	15	6	13	8	3.09596E-01
6	3	3	7	5	5.14226E-03
6	6	5	8	6	2.50303E-01
6	9	6	10	7	2.98162E+00
6	12	6	12	8	1.72976E+01
6	15	N/A	N/A	N/A	Timeout
8	3	4	8	5	5.07441E-02
8	6	5	10	7	1.1266E+02
8	9	N/A	N/A	N/A	Timeout
8	12	N/A	N/A	N/A	Timeout
8	15	N/A	N/A	N/A	Timeout
10	3	5	9	6	6.10978E-01
10	6	N/A	N/A	N/A	Timeout
10	9	N/A	N/A	N/A	Timeout
10	12	N/A	N/A	N/A	Timeout
10	15	N/A	N/A	N/A	Timeout

This table depicts changes in the number of attempts for Knuth's algorithm as both the length and number of symbols in the code change. For several attempts, the solver cannot complete these task and time outs. At 8 by 6 and 6 by 12, we can also see that the solver takes longer than 10 seconds - before these points in the final program, the program should switch to another algorithm to ensure maximum efficiency.

Test For Permuted Elimination

Permutation					
Length	Number	Minimmum Amount of Attempts	Max Amount of Attempts	Average Amount of Attempts	Average time taken for one attempt
2	3	2	6	4.34	1.25829E-05
2	6	2	9	6.73	3.03376E-05
2	9	2	12	9	4.3528E-05
2	12	2	15	10.73	6.20514E-05
2	15	2	18	12.83	8.79553E-05
4	3	2	8	6.11	6.86794E-05
4	6	4	13	9.21	4.80119E-04
4	9	6	16	12.09	1.90891E-03
4	12	7	19	14.26	5.81815E-03
4	15	7	22	16.69	1.40035E-02
6	3	4	11	7.39	4.36712E-04
6	6	7	14	10.92	1.86465E-02
6	9	11	17	13.9	2.05956E-01
6	12	N/A	N/A	N/A	N/A
6	15	N/A	N/A	N/A	N/A
8	3	6	11	8.51	4.01129E-03
8	6	N/A	N/A	N/A	N/A
8	9	N/A	N/A	N/A	N/A
8	12	N/A	N/A	N/A	N/A
8	15	N/A	N/A	N/A	N/A
10	3	7	12	9.4	4.39371E-02
10	6	N/A	N/A	N/A	N/A
10	9	N/A	N/A	N/A	N/A
10	12	N/A	N/A	N/A	N/A
10	15	N/A	N/A	N/A	N/A

This table depicts changes in the number of attempts for Knuth's algorithm as both the length and number of symbols in the code change. It is easy to see that this algorithm is much less efficient than Knuth's elimination and also breaks down at more combinations - making it less useful than Knuth's Elimination. This algorithm will most likely not be used in the final code.

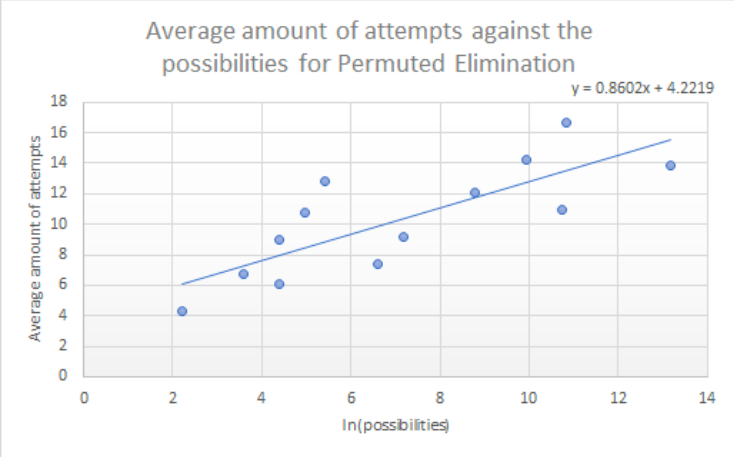
Test For Incremental (Brute Force)

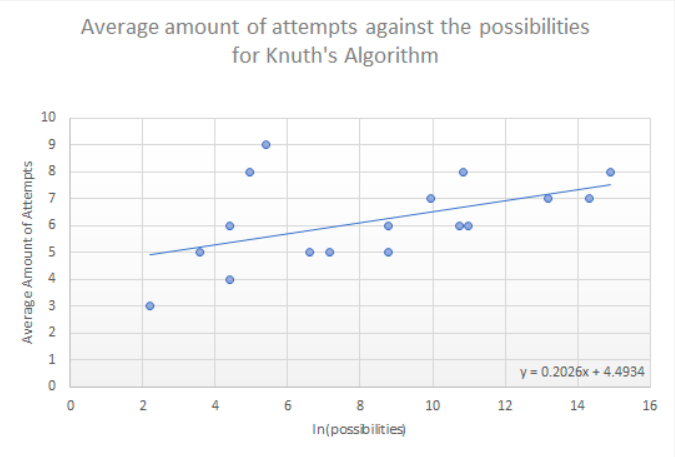
Incremental					
Length	Number	Minimmum Amount of Attempts	Max Amount of Attempts	Average Amount of Attempts	Average time taken for one attempt
2	3	2	5	4.17	3.71346E-06
2	6	2	8	6.27	7.97079E-06
2	9	2	11	8.4	1.06678E-05
2	12	4	14	10.34	8.18245E-06
2	15	3	17	12.23	1.49856E-05
4	3	2	9	6.5	1.03155E-05
4	6	6	13	9.93	1.68237E-05
4	9	9	16	12.91	2.06797E-05
4	12	8	19	15.06	2.19118E-05
4	15	7	22	15	3.53872E-03
6	3	5	12	9	2.5207E-05
6	6	9	20	14.22	2.49451E-05
6	9	10	23	17.43	3.20811E-05
6	12	13	27	20.59	3.25844E-05
6	15	16	29	23.54	3.64944E-05
8	3	8	15	11.81	2.52506E-03
8	6	12	26	18.41	3.46446E-05
8	9	15	35	22.95	4.16126E-05
8	12	18	34	26.43	4.45714E-05
8	15	19	40	29.79	4.44935E-05
10	3	10	18	14.44	2.1654E-05
10	6	11	31	22.77	3.33494E-05
10	9	20	39	28.89	4.95961E-05
10	12	23	47	34.26	6.39701E-05
10	15	27	54	38.05	7.28668E-05

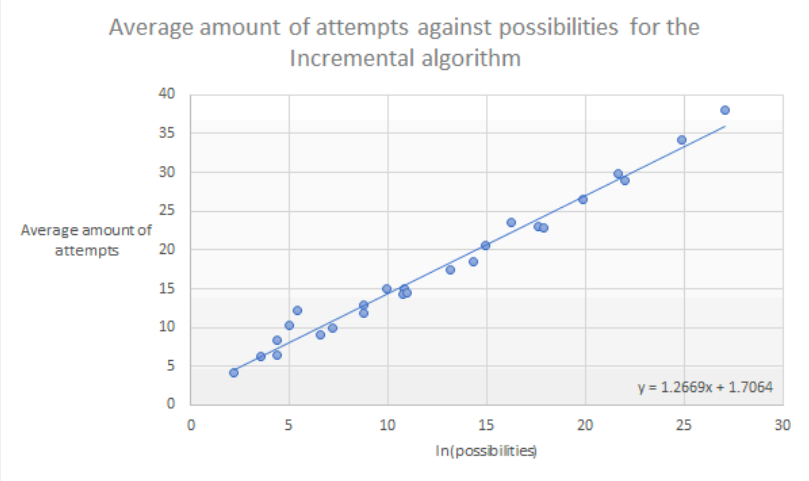
This table depicts changes in the number of attempts for the Incremental algorithm as both the length and number of symbols in the code change. This algorithm works for all lengths and number of symbols and is extremely quick when executed - there are no problems with time. However, for smaller lengths of code, the algorithm is inefficient and takes many more attempts than both Knuth's and Permuted Elimination.

Graphical Observations

When constructing the graphs to display the results, it was decided that the raw data would be processed so that the graphs used would have greater meaning and could be applied for optimisation. For example, as opposed to using the length of code and number of symbols - the data was processed to form a new variable , the amount of possibilities. This is the number of symbols to the power of the length of sequence. The following three graphs all represent the relationship between the logarithm of the possibilities and the average amount of attempts.

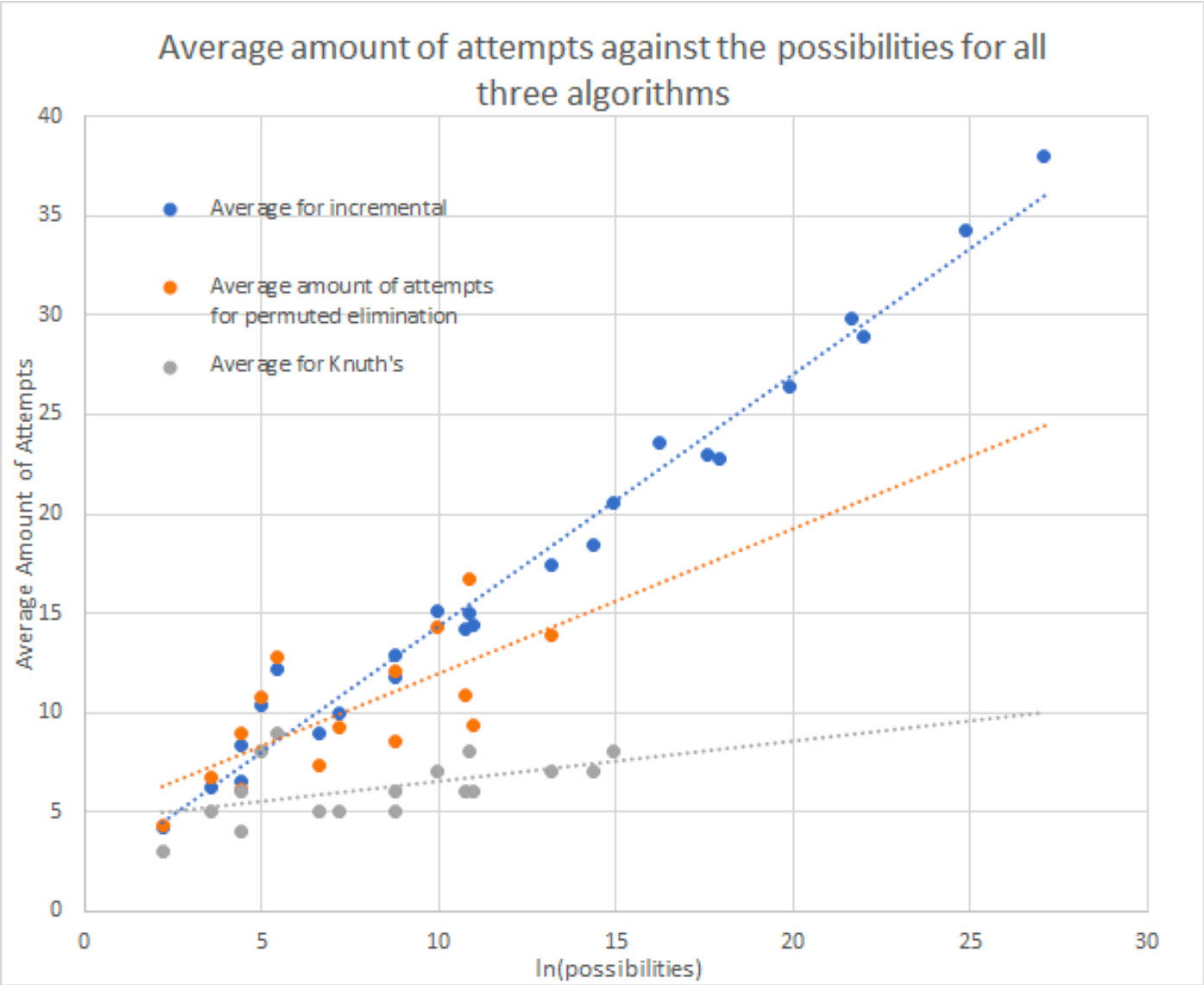






These graphs all show an exponential relationship between the average amount of attempts and the number of possibilities.

When combining all three graphs we can compare the gradients to see their efficiency.
It is easy to see that Knuth's elimination is the most efficient followed by the permuted code then lastly the incremental algorithm.



Optimisation

Using the graphs and data we have, we can optimise the program to switch between the algorithms so that it is as efficient as possible.

Although Permuted Elimination is more efficient than the brute force algorithm, it breaks-down at the same point as Knuth's Elimination, this results in the algorithm being included in the code, however not being used, no matter the circumstance, as it would not operate at larger code lengths, and at smaller lengths, Knuth's elimination is more efficient.

10000000 possibilities was chosen to be the point where the algorithm will switch from Knuth's to the incremental algorithm, as this is when Knuth's algorithm breaks down.