Solving mastermind

Yuting Xu
Imperial College London
yx8918@ic.ac.uk
CID:01518870

Abstract

Mastermind is a board game invented in 1970 by Mordecal Meirowitz. The game is played as follows: one player is the code setter, while the other player tries to crack the code with the help of feedback from each attempt. Feedback is given by the code setter in terms of number of black and white pegs. Black pegs indicate both colour and position are correct. White pegs indicate that colour is correct but is not in the right position. In the classic mastermind, the total number of positions is four and the number of colours is six. In this report I will recreate the heuristic climbing algorithm proposed by Temporel and Kovacs (see [1]) and compare its performance to other existing algorithms.

Possible Strategies

There are a handful of algorithms for solving mastermind. Berghman, Goossens, and Leus [2], classified the existing algorithm into three categories: full enumeration, heuristics and meta-heuristics.

A typical example of a full-enumeration algorithm is the one proposed by Knuth [3], in which the solver lists all possibilities and reduces the number of them throughout the game. With the help of full enumeration and a fixed opening strategy (always start with 1122), Knuth's algorithm will always find the code within five steps, with an average step number of 4.478. This impressive result, however, is achieved with a drawback: extensive running time. For 15 x 15 mastermind, the possible pool will consist of 15^15 = 4.3789389×10^17 combinations, putting huge strain on memory and computation power,

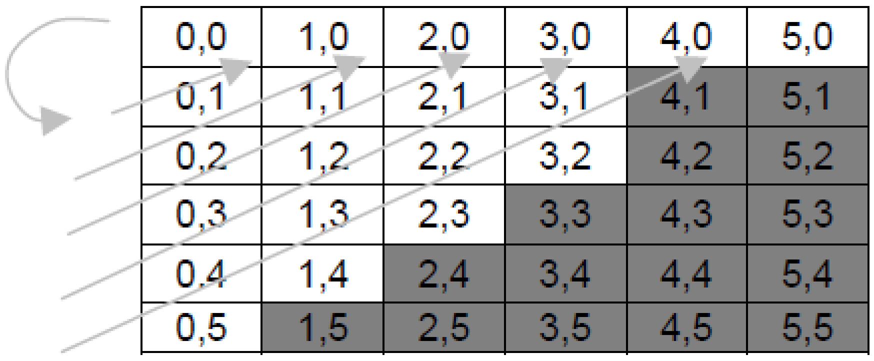
In this particular scenario where the code can only be allowed to run for at most 10 seconds, it is logical to implement other algorithms that require less computation for larger sizes of mastermind. Therefore, I chose the algorithm proposed by Temporal and Kovacs [4], that minimizes the computation needed.

Overview of the chosen algorithm

The code is constructed as follows:

```
Set i = 1;
Play random guess $g_1$;
Get response $b_1$ and $w_1$;     (black hits and white hits)
while $b_i \neq$ length do
    i = i + 1;
    while (attempt is consistent with previous guesses) and (attempt is new) do
        Randomly select $b_i$ numbers and keep them in place;
        Randomly select $w_i$ numbers to shift to different positions;
        Randomly mutate undetermined positions to other numbers;
    end while
    Submit guess $g_i$;
    Obtain feedback $b_i$ and $w_i$;
    Calculate score;
    if feedback of $g_i$ is better than current favorite guess
        best score = score;
        current favourite guess = $g_i$
    endif
end while
```

The current favorite guess is selected using a simple heuristic to estimate how far we are from the goal: N = black hits [1].



0,0	1,0	2,0	3,0	4,0	5,0
0,1	1,1	2,1	3,1	4,1	5,1
0,2	1,2	2,2	3,2	4,2	5,2
0,3	1,3	2,3	3,3	4,3	5,3
0,4	1,4	2,4	3,4	4,4	5,4
0,5	1,5	2,5	3,5	4,5	5,5

Table 1: Possible MM responses-5 peg game

Source: Temporel and Kovacs [1]

To sum up the above-mentioned heuristic:

1. Guesses with more hits are rewarded.
2. White hits are penalized.

This translates in to: only replace current favorite guess if:

1. The new guess has more hits
2. The number of hits are the same but black hits > white hits

Temporal and Kovacs [1] investigated which approach (replace current favorite guess when the submitted guess has the same feedback or better feedback) yields better performance, and found little to no difference.

After extensive testing, the heuristic hill climbing algorithm was unable to finish calculations in situations where P + N > 18. Therefore, a simple shifting algorithm was implemented to supplement the heuristic hill climbing algorithm for P + N > 18 situations.

Shifting algorithm

Assuming that P = 4, N = 6, the algorithm starts with 0000, replace the first digit with 0 + 1= 1 getting 1000. Then it shifts the 1 from the left to the right. If the shifted vector 0100 returns one more black hit, this means that the '1' peg is at the correct position. If it returns one less black hit, this means that the '0' peg is at the correct position. Once a peg's position is determined, it is saved and the algorithm tries to determine the position of the next color. This is repeated until all the pegs have been determined.

Performance and comparisons

Table 1. Results for P = 4, N = 6

Algorithm	Average	Maximum number of guesses
Bestavros and Belal (MaxEnt)	3.835	-
Bestavros and Belal (MaxMin)	3.86	-
Knuth [3]	4.478	5
Berghman et al. [2]	4.39	7
Original algorithm	4.64	-
Recreated algorithm	4.66	7

Source: Berghman et al. [2]

We can observe that full-enumeration algorithms achieve great results. Bestarvros and Belal [4] obtained an impressive average of 3.835 by combining the full-enumeration method with information theory. In each step they calculate the information entropy of each instance in the pool and select the maximum one as the next guess (MaxEnt). Their MaxMin approach implements a one-step lookahead feature that submits guesses which minimizes the pool in the next turn.

However, these methods still have a downfall which is their computation time. As P and N increases, the increase in computation time does not show a linear relationship. The computation time needed increases drastically. It should be noticed that the genetic algorithm proposed by Berghman et al. [2] also produces good results but is faster compared to full-enumeration methods,

It can be seen that the recreated algorithm performs on par with other competitors, though slightly worse than the results claimed in the paper by Berghman et al [1].

Table 2. Results for P = 5, N = 8

Algorithm	Average
Bento et al. [5]	6.866
Berghman et al. [2]	5.618
Kalisker and Camens [6]	6.39
Merelo-Guervós et al. [7]	5.904
Ugurdag et al. [8]	6.169
Recreated algorithm	5.988

Source: Berghman et al. [2]

Table 3. Results for P = 6, N = 9

Algorithm	Average
Berghman et al. [2]	6.475
Rosu [9]	6.67
Shapiro [10]	6.39
Swaszek [11]	7.41

Ugurdag et al. [8]	7.079
Recreated algorithm	6.73

Source: Berghman et al. [2]

Although the original paper only covered results for the classic mastermind (P = 4, N = 6), I carried out multiple tests on different sizes and compared them to other competitors. The average number of guesses is still promising, but the computation time lags behind that of Berghman et al. [2].

Discussion / further work

As for the heuristic hill climbing algorithm, there many ways to optimize its performance. The limiting factor in the code was the PRNG (Psuedo-random number generator). rand() % is now deprecated. One of its problems being that the '%' operator does not produce a uniform distribution. Performance should increase if rand() were replaced by a well-performing random engine (such as the mt19937) and a uniform distribution. Moreover, the code tracker mentioned in the original paper [1] was too vague. Therefore, the code tracker was not implemented in this report. Therotically, the code tracker will speed up the computation as it avoids attempts that have already been evaluated. Furthermore, since new attempts are randomly generated from the current favourite guess, it it possible to use multithreads to generate new attempts and evaluate them at the same time. This should theoretically boost the speed by X times where X is equal to the the number of threads.

As for how to optimize the supplementary shifting algorithm, the first X (where X is equal to the N, the number of colors) guesses could be devoted for finding out how many pegs of a certain color are present. Then carry out operations for the following guesses.

Conclusion

The recreated heuristic hill climbing algorithm manages to be both efficient in terms of memory usage and effective in finding the code at the same time. It would be interesting to improve the algorithm's speed in the future.

References

[1] Temporel, A. and Kovacs, T., 2003, September. A heuristic hill climbing algorithm for Mastermind. In *UKCI'03: Proceedings of the 2003 UK Workshop on Computational*

Intelligence, Bristol, United Kingdom (pp. 189-196).

[2] Berghman, L., Goossens, D. and Leus, R., 2009. Efficient solutions for Mastermind using genetic algorithms. *Computers & operations research*, 36(6), pp.1880-1885.

[3] Knuth, D.E., 1976. The computer as master mind. *Journal of Recreational Mathematics*, *9*(1), pp.1-6.

[4] Bestavros, A. and Belal, A., 1986. Mastermind a game of diagnosis strategies. In *Alexandria University*.

[5] Bento, L., Pereira, L. and Rosa, A., 1999, February. Mastermind by evolutionary algorithms. In Proceedings of the 1999 ACM symposium on Applied computing (pp. 307-311). ACM.

[6] Kalisker, T. and Camens, D., 2003, July. Solving Mastermind using genetic algorithms. In Genetic and Evolutionary Computation Conference (pp. 1590-1591). Springer, Berlin, Heidelberg.

[7] Merelo-Guervós, J.J., Castillo, P. and Rivas, V.M., 2006. Finding a needle in a haystack using hints and evolutionary computation: the case of evolutionary MasterMind. Applied Soft Computing, 6(2), pp.170-179.

[8] Ugurdag, H.F., Sahin, Y., Baskirt, O., Dedeoglu, S., Goren, S. and Kocak, Y.S., 2006, June. Population-based FPGA solution to Mastermind game. In null (pp. 237-246). IEEE.

[9] Rosu, R., 1999. Mastermind. Master's thesis, North Carolina State University, Raleigh, North Carolina.

[10] Shapiro, E., 1983. Playing mastermind logically. ACM SIGART Bulletin, (85), pp.28-29.

[11] Swaszek, P.F., 2000. The mastermind novice. Journal of Recreational Mathematics, 30(3), pp.193-198.