Mastermind Strategies

PAP Java Assignment 2, Question 2 Part iii

There are number of different ways to approach the problem of calculating the correct combination in a game of Mastermind.

Unless otherwise stated we will assume that four pegs can be chosen from a choice of six colours.

One of the simplest strategies to solve the problem is to use a brute force approach. This would involve enumerating a list of each of the possibilities and then using linear search to check each of them in turn. If any one is the correct answer (as indicated by having four black pegs) we can exit the algorithm, otherwise we continue searching until we find the correct answer. In the best case scenario, the correct combination will be the first possibility generated, but in the worst case it’ll be the last. So we can say that it is an algorithm. The average case will therefore be . In our example, this means that in the worst case could take guesses, with an average case of 648.5.

Though this approach will always find the solution if the number of guesses is unlimited, in practice the maximum number of guesses is usually capped preventing such brute force methods from working in most cases, meaning the algorithm will often fail before it gets to the correct solution. More fundamentally though, it doesn’t make use of the feedback on whether we have any of the colours correct, therefore useful information is wasted.

One way that we can make more effective use of the information available to us is by pruning our list of all possibilities after every guess. We could use the information about the number of black and white from the last go to eliminate any possible guess that could not possibly be the correct answer. For example, if we were to choose (R, R, G, G) as our initial guess and we got back a response of 0 blacks and 0 whites, then we can immediately eliminate every combination that contains either a red or a green somewhere in the pattern, drastically reducing the number of possibilities.

The pseudocode for such an approach looks like the following:

remaining\_possibilites = set of all remaining possible combinations (initially contains all possibilities)

FUNCTION remove\_impossible\_guesses(outcome):

FOREACH possibility IN all\_possibilities:

current\_outcome = GET score for possibility against current guess

IF current\_outcome NOT EQUAL TO outcome:

REMOVE possibility FROM all\_possibilities

END IF

END FOREACH

END FUNCTION

FUNCTION guess():

REPEAT:

MAKE A GUESS

outcome = GET score for current guess against hidden solution

remove\_impossible\_guesses(outcome)

UNTIL outcome is 4 blacks

END FUNCTION

In the above pseudocode the main thing that we are doing is comparing every remaining possible combination against the result from the last guess and checking whether or not the result would be the same. If it is not, then it cannot be the correct guess (as highlighted by the earlier example).

Now that we have a way of eliminating possibilities that cannot be correct one must consider next, how do we choose the next guess at each stage? The most obvious strategy for this may simply be to choose any arbitrary element of the set of remaining possibilities. The advantage of this is that it is computationally inexpensive as the operation can be completed in constant time. However, it is not always going to give the optimal result as we are not performing any analysis or look-ahead to see if this will help finish the game earlier.

That being said, in my implementation in Q2vii after doing 1000 sets of runs of the game trying every possible result (for a total of 1,296,000 runs) I have determined that this approach yielded the correct solution with an average case of 4.636 and a worst case of 8. This can generally be considered a reasonable result as it completes in under 5 turns on average and even in the worst case it still completes in less than the 10 guesses allowed in the standard version of the game.

As previously mentioned, choosing an arbitrary element doesn’t always give us the best possible result, so we must explore other strategies if we want to find a truly optimal solution. One such solution is to use the “minimax” technique for this game, as identified by Donald Knuth[[1]](#footnote-1).

This goal of the algorithm is to choose the next guess to be the one that is likely to eliminate the most combinations from our set of remaining possibilities. It does this by iterating through all possible combinations, not just those remaining possible combinations, and then calculating its score. The score in this case is the minimum number of possibilities it might eliminate from the set of remaining combinations. The possibility with the largest score is what is used as the guess for the next turn, with a preference being for a guess that is also in the set of remaining possibilities.

The advantage of using this method is that is gives us an optimal worst case of 5, however it is very computationally intensive. Since every possible combination should be compared to every remaining possibility and every possible outcome it could require as many as iterations before returning the next guess.

In my implementation I have opted to use a slight variation on the minimax method by only looking at combinations that are still possible solutions. Though this gives a suboptimal solution, it vastly improves the speed that each guess takes to compute providing a compromise between program speed and the end result. I have determined that this technique yielded an average case performance of 4.482 with a worst case of 7.

1. “Five-guess algorithm”. <https://en.wikipedia.org/wiki/Mastermind_(board_game)>. Retrieved on 25th March 2015 [↑](#footnote-ref-1)