An Information Theoretic Analysis of One-Step-Ahead Mastermind Strategies

Jouke Witteveen Jan van Eijck

April 9, 2014

Abstract

This paper compares various one-step-ahead Mastermind strategies, rephrases some of these strategies in information-theoretic terms, and presents an improved strategy. The paper gives documented Haskell code of all strategies that are discussed, and of their analysis. Our code computes the full analysis of each strategy in a matter of seconds.

Introduction

Mastermind is a knowledge update game that was invented in 1970, by Mordecai Meirowitz, and commercialized by Invicta Toys and Games (see `http://dspace.dial.pipex.com/town/road/gbd76/toys.htm`. Mastermind is usually presented as a two-player game (zero sum, with imperfect information) between a code-maker and a code-breaker, but because the code-maker faces a computationally trivial task it makes sense to study it as a solitaire game for the code-breaker. That Mastermind is a non-trivial game is witnessed by the fact that generalized Mastermind (played with two colours and n pegs) is NP complete [de Bondt, 2004].

Solitary Mastermind fits the class of key-guessing games where the player figures out a designated element (the *key*) of a set (the *code space*) based on the information provided by the structured feedback to repeated guesses. The aim of these games is to minimize the expected number of guesses needed to find the key. In Mastermind, the key is assumed to be picked at random from the code space, under the uniform distribution.

Example 1. We can visualize strategies for key-guessing games as trees. Suppose the set of possible keys is $\{1, 2, 3, 4, 5, 6\}$ and the feedback we get for a guess is whether the guess is equal to, greater than ($>$), or less than ($<$) the key. Now, given a set of possible keys that are consistent with the feedback history, we choose a guess. These choices, `<consistent set>` $\rightarrow$ `<guess>`, can be placed as nodes on a tree, where the edges are induced by the possible feedback to the guess. For a certain procedure to determine our choices (a strategy) we can get a tree as in Figure 1. It is easy to see that this strategy is optimal for this particular game.

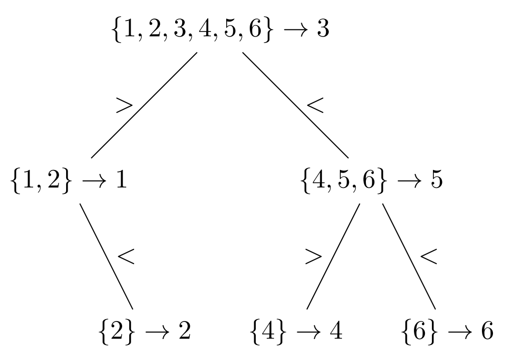

Figure 1: A possible strategy for a key-guessing game on the set $\{1, 2, 3, 4, 5, 6\}$, with ordinary comparison as feedback. Every path in the tree corresponds to a play of the game.

The Mastermind code space consists of all fixed length sequences of *pegs* coloured from a palette. Usually, the length of the sequences is four and the palette contains six colours. The resulting code space of $6^4 = 1296$ codes will be our reference code space throughout this report.

Feedback in Mastermind consists of pairs (b, w), where b is the number of positions where the colour of the peg in the guessed code matches that of the peg in the key, and w is the number of coloured pegs of the key that are in the wrong position in the guess. E.g., if the key is $(\mathrm{blue}, \mathrm{green}, \mathrm{blue}, \mathrm{red})$, then the guess $(\mathrm{green}, \mathrm{yellow}, \mathrm{red}, \mathrm{red})$ will solicit feedback $(1, 2)$.

With each guess, the subset of the code space consistent with the guessing history decreases, hence our knowledge about the key increases. It is known that the optimal strategy for Mastermind takes 4.340 guesses on average to guess the key. This result was found using exhaustive search [Koyoma and Lai, 1993] and verified by Ville [2013]. In this report we will focus on *one-step-ahead* strategies that consider only the possible outcomes of the next guess.

Implementation: The Mastermind Framework

We present our code in literate programming style [Knuth, 1992], in Haskell [The Haskell Team]. If the module is loaded as object code in the GHC interpreter (`ghci -fobject-code Mastermind`), then the `summary` function defined in the Results section will compute its output in a matter of seconds.

```haskell
module Mastermind where
import Data.List
import qualified Data.Map as M
```

We will treat the palette as an alphabet. Using letters instead of colours, we get the following datatype synonyms for codes and feedback:

```haskell
type Code       = String
type Feedback   = (Int, Int)
```

The most general definition of a strategy is that of a mapping of a subset of the code space, embodying all knowledge about a key at some point, to a code that will be the guess. If the guess is part of the set of consistent (with the feedback history) codes, we say that the guess is *consistent*, otherwise we call it *inconsistent*.

```
type Strategy    = [Code] -> Code
```

A more subtle kind of strategy rates all possible feedback outcomes of the next guess and picks the next guess on the basis of these ratings. Because only the outcomes of the next guess are considered, these strategies can rightfully be called *one-step-ahead* strategies.

```
type Strategy1 a = Count (Counts Feedback) -> a
type Count a     = (a, Int)
type Counts a    = M.Map a Int
```

Generalized Mastermind code spaces can be generated from their alphabet and the number of pegs:

```
mastermind :: [Char] -> Int -> [Code]
mastermind _       0    = [""]
mastermind colours pegs = [c:cs |
  c  <- colours,
  cs <- mastermind colours (pegs - 1)]
```

The usual game uses six colours and four pegs:

```
theGame = mastermind ['A' .. 'F'] 4
```

The feedback to a guess, generated by a key:

```
feedback :: Code -> Code -> Feedback
feedback guess key = (black, white)
  where black = length $ filter (uncurry (==)) $ zip guess key
        white = occursCount key guess - black

occursCount :: Code -> Code -> Int
occursCount _   []     = 0
occursCount key (x:xs)
  | x `elem` key       = 1 + occursCount (delete x key) xs
  | otherwise          = occursCount key xs
```

We will need an auxiliary function for partitioning based on a classifying function:

```
partitionBy :: Ord b => (a -> b) -> [a] -> M.Map b [a]
partitionBy f xs = M.fromListWith (flip (++))
  [(f x, [x]) | x <- xs]
```

Example use of this function:

```
GHCi> partitionBy (`mod` 3) [1 .. 10]
fromList [(0,[3,6,9]),(1,[1,4,7,10]),(2,[2,5,8])]
```

Strategies

Elementary Strategies

In our deterministic, stateless framework, arguably the simplest strategy is the naive strategy [Shapiro, 1983]. This strategy selects the first consistent code as a guess:

```
naivePlay :: Strategy
naivePlay = head
```

naivePlay

Observe that the naive strategy has time and space complexity $\mathcal{O}(1)$ and is thus very much a *zero-steps-ahead* strategy. Nevertheless, for some key-guessing games this strategy is optimal. In particular it is optimal for searching the index of a designated element in an unordered array. In this game, the index set forms the code space and the feedback is one of $\{\neq, =\}$.

For our analysis of one-step-ahead strategies, it is instructive to look at how a one-step-ahead strategy is transformed into a general strategy. We realize such a transformation as follows:

1. Generate profiles of all possible feedback for all candidate guesses;
2. Partition the candidate guesses according to their feedback profile;
3. Run the one-step-ahead strategy on all feedback profiles and select the (union of the) set(s) of candidate guesses that minimizes the value assigned by the one-step-ahead strategy;
4. Pick the first consistent code from this set, or the first inconsistent code if there are no consistent codes in it.

In our implementation, at step 3, we only take the first code from each set into consideration, as the other codes will be discarded in the next step anyway. This is because either all codes from a partition cell are consistent (if the feedback indicating success is part of the feedback profile), or they are all inconsistent (otherwise).

Furthermore, we make the set of candidate guesses into a parameter, which defaults to the set of consistent guesses when empty.

```
strategy1 :: Ord a => [Code] -> Strategy1 a -> Strategy
strategy1 candidates strat cons =
  minimum $ if null cs then is else cs
  where
    (cs, is) = partition (`elem` cons) $ bestCodes ratings
    ratings  = M.foldrWithKey acc [] $ partitionBy fbs from
    acc f xs = (:) ((strat (f, length xs)), [head xs])
    fbs x    = M.fromListWith (+)
                 [(feedback x k, 1) | k <- cons]
    from     = if null candidates then cons else candidates

bestCodes :: Ord a => [(a, [Code])] -> [Code]
bestCodes [(_, xs)]  = xs
bestCodes (xs:ys:zs) = case compare (fst xs) (fst ys) of
  LT -> bestCodes (xs:zs)
  GT -> bestCodes (ys:zs)
  EQ -> bestCodes ((fst xs, snd xs ++ snd ys):zs)
```

Observe that if `profile` is the feedback profile for a class of guesses, then `(M.elems . fst) profile` is a list of sizes of the consistent sets one step ahead. Relating to Example 1, it is a list of the sizes of the consistent sets in the children of a node. If we are unlucky in a game, we end up with the largest of the possible resulting sets of consistent codes after a guess. As we have to reduce the size of the set of consistent codes to 1, eventually, this can be considered a *worst case scenario*. Minimizing this maximum size is suggested in [Knuth, 1976–1977] and can be taken as a strategy to minimize the maximum number of guesses needed.

```
wcPlay :: Strategy1 Int
wcPlay = maximum . M.elems . fst
```

`wcPlay`

As we will see in the section on results, this approach indeed gives an optimal worst case scenario, on the condition that inconsistent guesses are allowed (the strategy would be expressed as `strategy1 theGame wcPlay`). When applied to the game of Example 1, this strategy would yield the optimal strategy displayed in Figure 1.

Information Theory

While `wcPlay` manages to minimize the maximum number of guesses needed, our goal is to minimize the expected number of guesses needed. More visually, considering the tree corresponding to a strategy as was done in Example 1, `wcPlay` minimizes the height of such a tree, whereas we would like to minimize the expected depth of the nodes that represent consistent guesses in such a tree. In turn, this can be interpreted as an optimality criterion for a kind of encodings of the code space.

As shown in Figure 1, the branches of the tree corresponding to a strategy are related to feedback values. Thinking of all possible feedback values as the *feedback alphabet*, a strategy thus induces an encoding of the code space with the feedback alphabet. In this setting [Cover and Thomas, 1991], the tree corresponding to the strategy is known as the encoding tree.

Example 2. In the strategy depicted in Figure 1, the code ‘1’ would be encoded as ‘>’, the code ‘2’ would be encoded as ‘><’, the code ‘3’ would be encoded as ‘’ (the empty string), and so on.

Now that we have seen that goodness of Mastermind strategies is equivalent to optimality of a kind of encodings, we might want to apply Huffman coding to the code space. However, Huffman coding works in a bottom-up fashion, so applying it requires knowing which subsets of the code space constitute possible consistent sets. Unfortunately, for Mastermind we know of no efficient way to do so. An exhaustive search through all possibilities resembles a brute force strategy and as such is outside the scope of this report.

A top-down alternative to Huffman coding exists in the form of Shannon–Fano coding (called *Fano coding* in [Cover and Thomas, 1991]). In order to use it for our cause, we must first formulate it in a more general form, that is not restricted to binary codes. Whereas ordinary Shannon–Fano coding prescribes to split a set of codes into two parts that are maximally similar in a probabilistic sense, in our case not all partitions are allowed and the number of parts (the branching factor in the encoding tree) need not be two. Remark that sorting the probabilities of the code, as is part of Shannon–Fano coding is not relevant to us, since Mastermind assumes a uniformly distributed key.

The generalized Shannon–Fano coding simply picks the partition of a set of codes that minimizes the expected encoding length. In the context of arbitrary prefix free codes, the expected (and optimal) encoding length of a set of size n is $\log_b n$, where b is the size of the alphabet. Hence for a partition of a set of N uniformly distributed elements into sets of sizes $n_1, n_2, \dots, n_k$ respectively, the expected encoding length is $\sum_{i=1}^{k} \frac{n_i}{N} \log_b n_i = \frac{1}{N \log b} \sum_{i=1}^{k} n_i \log n_i$. The leading factor $\frac{1}{N \log b}$ is independent of the partition, so the partition that is selected in general Shannon–Fano coding is the one that minimizes the sum.

```
sfPlay :: Strategy1 Float
sfPlay = sumMap (f . fromIntegral) . fst
  where f n = n * log n

sumMap :: Num b => (a -> b) -> M.Map k a -> b
sumMap f = M.foldr' ((+) . f) 0
```

sfPlay

Note that this is indeed equal to traditional Shannon–Fano coding when the allowed partitions are precisely all possible partitions into two parts.

We will see that `sfPlay` performs badly for Mastermind. The reason for this is that in Mastermind the expected encoding length of a set of n codes is *not* $\log_b n$. This is because:

1. Strategies for key-guessing games such as Mastermind are *not* prefix free codes (see: Example 2);
2. The full Mastermind feedback alphabet is almost never present in an allowed partition, which effectively means that we are dealing with varying subsets of the feedback alphabet and thus with varying branching factors in our encoding tree.

If the expected encoding length is not the optimal encoding length, it must be longer. A very pessimistic alternative, is to consider an encoding tree where all nodes represent consistent guesses, and with a branching factor of one. In this case, the expected encoding length of a set of size n is $\frac{1}{2}n$, which is a worst case for fully consistent strategies. This leads to a strategy that minimizes the expected (after one move) size of the consistent set [Irving, 1978–1979]:

```
exPlay :: Strategy1 Int
exPlay = sumMap (^ 2) . fst
```

exPlay

This strategy turns out to do quite well on an enlarged Mastermind code set. Unfortunately, the varying branching factor makes it unfeasible to make a proper estimate of the true expected encoding length of a given set. Complementary to the attempt of finding better estimates of the expected encoding length is the strategy that greedily maximizes the effective size of the encoding alphabet. For a guess corresponding to a partition of a set of s uniformly distributed elements into k parts, the entropy of the set is greedily approximated by $-\log_k \frac{1}{s} = \log_k s$. The strategy to minimize this quantity can be expressed by:

```
branchPlay :: Strategy1 Float
branchPlay (x, _) = logBase k s
  where k = fromIntegral $ M.size x
        s = fromIntegral $ sum $ M.elems x
```

branchPlay

As s is the same for all possible guesses at any point in a game, we find that (for $s > 1$) `branchPlay` is equivalent to the strategy suggested by Kooi [2005], which is simply to maximize the size of the partition. As we are in a setting of minimization, this strategy takes on the form:

```
kooiPlay :: Strategy1 Int
kooiPlay = negate . M.size . fst
```

kooiPlay

An important observation about `kooiPlay` is that for any key-guessing game, along any path in the tree corresponding to this strategy the branching factor is non-increasing. This is easily proved by contradiction. Consequently, we improve on `kooiPlay` when we manage to reduce the expected rate of decrease of the branching factor. Part of the information available to our one-step-ahead strategies is the number, let us call it n, of possible guesses that generate a given feedback profile. If the feedback profile consists of k parts, we are thus given n ways of splitting some consistent set into k parts. Now, if k and n are reasonably high, this suggests we can maintain a reasonably high branching factor for a reasonably high number of guesses.

We turn this line of thought into a strategy by anticipating that if the feedback profile gets selected as optimal, then all n possible guesses will remain close to optimal for as long as they are consistent in the further play of the game. This means these elements do not contribute greatly to the entropy and we can treat all these n guesses as one. The modified `branchPlay` becomes:

```
branch2Play (x, y) = logBase k $ s − n + 1
  where k = fromIntegral $ M.size x
        s = fromIntegral $ sum $ M.elems x
        n = fromIntegral y
```

This, however, needs some tweaking. For instance `logBase 1 1` is undefined, so we catch it separately and choose the value zero, for continuity in the second argument. Additionally, inconsistent codes are not part of our entropy calculation, so in case of inconsistent guesses, there are no codes to exclude. For this reason, we add a parameter to the strategy, which should be set to the feedback solicited by a correct guess, by which we can tell whether the guesses under consideration are consistent or inconsistent.

```
branch2Play :: Feedback −> Strategy1 Float
branch2Play correct (x, y)
  | s == n     = 0
```

branch2Play

```
  | otherwise = logBase k $ s - n + 1
  where
    k = fromIntegral $ M.size x
    s = fromIntegral $ sum $ M.elems x
    n | M.member correct x = fromIntegral y -- consistent
      | otherwise          = 1              -- inconsistent
```

Contrary to the case of `branchPlay`, this strategy has no immediate shorter equivalent.

Results

To analyze Mastermind strategies, we implement a routine that counts the depths of the consistent nodes in the tree below a given consistent set. When this routine is run on the entire code space, we find how many guesses are needed how often to guess a key.

```
summary :: Strategy -> [Code] -> Counts Int
summary strat cons =
  M.mapKeysMonotonic (+ 1) $
  M.unionsWith (+) $
  (++) [M.singleton 0 1 | guess `elem` cons] $
  map (summary strat) $  -- parallelization possible
  M.elems $
  partitionBy (feedback guess) $
  delete guess cons
  where guess = strat cons
```

The simplest use of this is an analysis of the `naivePlay` strategy:

```
GHCi> summary naivePlay theGame
fromList [(1,1),(2,4),(3,25),(4,108),(5,305),(6,602),(7,196),
          (8,49),(9,6)]
```

A more complicated statement is needed for the analysis of `branch2Play` with inconsistent guesses allowed on the code space of the Mastermind variant with four colours and three pegs:

```
GHCi> let n  = 3
GHCi|     mm = mastermind ['A' .. 'D'] n
GHCi| in summary (strategy1 mm (branch2Play (n, 0))) mm
fromList [(1,1),(2,7),(3,33),(4,23)]
```

The values for our strategies, on the usual Mastermind code space with six colours and four pegs, restricted to consistent guesses are assembled in Table 1. The values obtained when inconsistent guesses are allowed are assembled in Table 2.

We see that `wcPlay` indeed has the best worst case performance, with at most five guesses needed when inconsistent guesses are allowed. For the calculation of the averages in Table 3, the following was used:

	1	2	3	4	5	6	7	8	9
`naivePlay`	1	4	25	108	305	602	196	49	6
`wcPlay`	1	12	99	468	662	54			
`sfPlay`	1	13	103	502	620	57			
`exPlay`	1	13	114	530	595	43			
`branchPlay`	1	13	115	556	563	46	2		
`branch2Play`	1	13	114	564	561	42	1		

Table 1: The number of possible keys in a game of Mastermind as a function of the number of guesses needed to guess the key, by strategy, allowing only consistent guesses.

	1	2	3	4	5	6	7	8	9
`naivePlay`	1	4	25	108	305	602	196	49	6
`wcPlay`	1	6	62	533	694				
`sfPlay`	1	4	71	612	596	12			
`exPlay`	1	10	54	645	583	3			
`branchPlay`	1	12	72	635	569	7			
`branch2Play`	1	12	96	606	557	24			

Table 2: The number of possible keys in a game of Mastermind as a function of the number of guesses needed to guess the key, by strategy, allowing inconsistent guesses.

```haskell
avgGuesses :: Counts Int -> Float
avgGuesses counts =
  sumMap ((/ total) . fromIntegral) $ M.mapWithKey (*) counts
  where total = fromIntegral $ sum $ M.elems counts
```

The results for the inconsistent versions of all but the new `branch2Play` were found before [Kooi, 2005, Ville, 2013]. Our new strategy outperforms all other one-step-ahead strategies, both in its consistent variant and in its inconsistent variant. However, on larger code spaces we find that it does not do very well. For Mastermind with six colours and five (instead of four) pegs, the expected number of guesses per strategy are listed in Table 4. The table does not pay attention to worst case behaviour, which is best for the inconsistent versions of `wcPlay` and `exPlay` — which comes as no surprise given their motivations — with a maximum of six guesses. All other strategies take a maximum of seven guesses. We conjecture that the small size of the usual Mastermind code space prevents quirks from averaging out, hence putting `sfPlay` in particular at a disadvantage.

	`naive`	`wc`	`sf`	`ex`	`branch`	`branch2`
consistent	5.765	4.497	4.465	4.415	4.399	4.390
inconsistent	5.765	4.476	4.415	4.395	4.373	4.372

Table 3: Expected number of guesses in a game of Mastermind for the strategies of Table 1 and Table 2.

	`wcPlay`	`sfPlay`	`exPlay`	`branchPlay`	`branch2Play`
consistent	4.8808	4.7975	4.7926	4.8008	4.8484
inconsistent	4.8823	4.7694	4.7755	4.7746	4.8233

Table 4: Expected number of guesses in a non-standard game of Mastermind with six colours and five pegs for our one-step-ahead strategies.

This is further suggested by the effectiveness of the greedy-but-not-too-greedy `branch2Play` on the very small code space of Mastermind with four colours and three pegs. There, the consistent version of `branch2Play` manages an expected number of guesses of 3.219, which is optimal [Ville, 2013]. None of our other strategies manages to achieve this. The first guess of `branch2Play` on this code space is `AAB`, which results in a branching factor of eight. The related `branchPlay` picks `ABC`, which results in a branching factor of nine. Higher, but apparently not better.

References

T. M. Cover and J. A. Thomas. *Elements of Information Theory.* Wiley-Interscience, New York, 1991.

M. de Bondt. NP-completeness of Master Mind and Minesweeper. Technical Report 0418, Department of Mathematics, Radboud University Nijmegen, 2004.

R. Irving. Towards an optimum Mastermind strategy. *Journal of Recreational Mathematics*, 11(2):81–87, 1978–1979.

D. Knuth. The computer as master mind. *Journal of Recreational Mathematics*, 9(1):1–6, 1976–1977.

D. Knuth. *Literate Programming.* CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

B. Kooi. Yet another Mastermind strategy. *ICGA Journal*, 28(1):13–20, 2005.

K. Koyoma and T. Lai. An optimal Mastermind strategy. *Journal of Recreational Mathematics*, 25(4):251–256, 1993.

E. Shapiro. Playing Mastermind logically. *SIGART Newsletter*, 85:28–29, 1983.

The Haskell Team. The Haskell homepage. `http://www.haskell.org`.

G. Ville. An optimal Mastermind (4, 7) strategy and more results in the expected case. *CoRR*, abs/1305.1010, 2013.