Master Minds: The Report

Xinhao Su
`xs2413`

David Xu
`dx2199`

December 22, 2021

1 Problem Overview

Master Mind is a classic codebreaking game first played in 1970. Gameplay goes as follows:

1. The "codemaker" picks four pegs and places them in order. This is the "solution code". Each peg can be one of six different colors and colors can be repeated between pegs.
2. The "codebreaker" then has 8 turns to guess this code. On each turn:
   (a) The codebreaker makes a guess of the code (four pegs in order, each having one of the six colors).
   (b) The codemaker responds with feedback using some number of black and white pegs. Each black peg means a peg of the guess is both the right color and in the right position. Each white peg means a peg of the guess is the right color but in the wrong position.
   (c) Using this information, the codebreaker can formulate his/her next guess.
3. The game ends after the code has been guessed (the codebreaker wins) or eight incorrect guesses have been made (the codemaker wins), whichever occurs first.

A photo of a physical game board is shown below, albeit with slightly different colors.

2 Formal Definition

This game can be formalized as The Mastermind Problem. Given a set of guesses and their corresponding feedback, can we determine what solutions code(s) would produce such behavior? This is effectively a multi-dimensional search problem which has been proven to be NP-Complete[1].

[1]Stuckman, J., & Zhang, G. Q. (2005). Mastermind is NP-complete. *arXiv preprint cs/0512049.*

3 Algorithm

Donald Knuth proposed the "Five-Guess Algorithm[2]", which was named as such because it will always determine the correct code using only at most five of the eight allowable guesses. The algorithm works as follows:

S.1 Generate all possible codes. Call this S, representing the universe of possible codes. Here, $|S| = 6^4 = 1296$.

S.2 Create a set of possible solutions P. At the start, $P = S$.

S.3 Choose an initial guess g_0. This can be hard-coded or randomly selected from P, as no information about the solution code has been obtained yet.

S.4 Play guess g_0 and obtain a response r from the codemaker.

S.5 Filter P to remove all codes which could not possibly be the solution based on this response.

- For a code to possibly be the actual solution, its response to g_0 must also be exactly r.

S.6 Select the best next guess g_1 from S using a minmax algorithm: minimize the maximum possible number of remaining codes in P after this guess.

- The maximum possible number of remaining codes in P after a guess g_2 can be calculated by determining the response of each code in P to g_2. The response which appears with the greatest frequency is the worst case (resulting in the largest P after filtering).
- In the case of a tie between codes, prefer a code which is still a possible solution, i.e. it is in P. As a final tiebreaker, select the numerically lowest code.

S.7 Repeat from Step **S.4** with guess g_1. Repeat until the solution is found ($|P| = 1$).

4 Extension

It is desirable to increase the computational difficulty of the problem so as to produce more reliable timing results. A program running over a longer amount of time will allow for recorded times to be more robust against random noise. As such, we increase the search space of the algorithm by extending the game to use c colors and h holes (previously, $c = 6$ and $h = 4$). As such, the universe of codes expands: $|S| = c^h$. For performance analyses in the remainder of this report, $c = 10$ and $h = 4$ are selected, resulting in $|S| = 10,000$.

5 Sequential Implementation

Segments of the sequential implementation are highlighted within this section. For a full code listing, see Appendix: Code. Among other things, the appendix includes code allowing a human codemaker to play against the algorithm.

5.1 Datatypes

```haskell
type ResponsePegs = (Int, Int) -- (#black, #white)
```

The response providing feedback about a code is presented as a pair consisting of the number of black pegs and number of white pegs in the response.

```haskell
type Code = [Int]
```

A code is a collection of colored pegs. Each peg is an *Int* which correponds to a color in set $[1..c]$.

```haskell
type Possibility = (Int, Bool, Code) -- (Score, Invalid, Code)
```

A *Possibility* is a candidate for the next guess. The score represents the size of P after this guess in the worst case. The invalid flag tracks whether the code is a possible solution (in P) or not. Note that finding the minimum *Possibility* will perform the next guess selection process including tiebreakers as described in Section 3.

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

5.2 Generating a Response

```haskell
guessResult :: Code -> Code -> ResponsePegs
guessResult ans guess = (numBlack, numWhite)
    where numBlack = length $ filter id $ zipWith (==) ans guess
          numWhite = sum (map minCodeCount $ nub guess) - numBlack
          minCodeCount v = min (count v ans) (count v guess)
          count v ls = length $ filter (==v) ls
```

The number of black pegs (*numBlack*) can be calculated by looking for pegs in the same position with the same color. For determining the number of white pegs (*numWhite*), each color is iterated over. The minimum number of times that color appears in both codes corresponds to the number of white pegs that color will generate (e.g., three green pegs in the guess and two green pegs in the solution means two white pegs are generated). However, the number of black pegs must be subtracted from this sum (as if a peg is both the right color and in the right position, it will generate a black peg *instead* of a white peg).

5.3 Scoring a Candidate Guess

```haskell
scoreGuess :: CodeSet -> Code -> Possibility
scoreGuess possible code = (score, not valid, code)
    where
        valid = code `elem` possible
        allResponses = map (guessResult code) possible
        score = getMaxCount allResponses
        getMaxCount xs = maximum $ map snd $ getCounts xs
        incCount o [] = [(o, 1)]
        incCount o (x@(v, c) : xs)
            | v == o = (v, c + 1) : xs
            | otherwise = x : incCount o xs
        getCounts xs = foldr incCount [] xs
```

Given the set of remaining possibilities P (*possible*) and some candidate guess *code*, the objective is to assign a score to the code representing the maximum size of P after this guess in the worst case, as described in Section 3. The *guessResult* function is used to determine what the codemaker would respond to g in the case that each potential solution was the actual solution. The *getCounts* function is used to count how many times each response occurs. The maximum count, determined by $getMaxCount$, corresponds to the score.

5.4 Filtering the Possible Set

```haskell
filterCodeSet :: CodeSet -> Code -> ResponsePegs -> CodeSet
filterCodeSet set guess response =
    filter ((response ==) . guessResult guess) set
```

Once a guess is made and a response is received from the codemaker, some of the possible solutions in P can be ruled out. Specifically, solutions which produce a response to the guess different from the one received cannot be the actual solution.

5.5 Playing the Game

```haskell
playMastermind ::  Code -> Code -> Int -> CodeSet -> CodeSet -> IO Int
playMastermind guess solution k fullSet possibleSet = do
  putStrLn $ "Guessing: " ++ show guess
  let response@(blk, wht) = guessResult guess solution
  putStrLn $
    "Response: " ++ show blk ++ " black and "
      ++ show wht ++ " white"
  if blk == length guess then do
    putStrLn $ "Solved: " ++ show guess
    return k
  else do
    let possibleSet' = filterCodeSet possibleSet guess response
    let possibilities = map (scoreGuess possibleSet') fullSet
    let (_, _, nextGuess) = minimum possibilities
    playMastermind nextGuess solution (k + 1) fullSet possibleSet'
```

Each turn, a guess is played by the algorithm (codebreaker) and a response received from the codemaker. If a number of black pegs equal to the number of holes is returned, the code has been found! In some cases, the algorithm may luckily guess the correct solution without having to reduce $|P|$ to 1. Otherwise, P is filtered using the response and then the universe of possible codes S is searched for the best next guess.

5.6 Performance

It is important to keep in mind that the performance of the algorithm is dependent on both the initial guess and the solution. For example, consider the trivial case where the initial guess is equal to the solution. The algorithm will succeed without having to search S at all. For this report, performance was analyzed on a 2018 MacBook Pro with 8 logical cores. The game was configured to use 10 colors and 4 holes. A shell script was used to feed input to the program, removing the need for human interaction. Using an initial guess of 1111 and a solution of 2613, the algorithm succeeded in 14 seconds (note that timings are rounded to the nearest second to show proper significance, as timings vary slightly between equivalent program executions). Using a solution of 8765 instead results in a 50 second execution time as shown below.

5.7 Fixing that Bump

One may notice that towards the end of the execution above, a sudden increase in activity occurs. Analysis using Threadscope shows that this is due to stack overflows, causing the computation to stop and start again multiple times. It was determined that this was largely because of the `minimum` and `maximum` functions which are evaluated on lists of size c^h. This necessitates the construction and evalulation of a large redex, causing a stack overflow[3]. To avoid this behavior, `minimum` was replaced with `foldl1' min`. A corresponding change was made replacing `maximum` with `foldl1' max`. Doing so alleviates the stack overflow issue (although an activity increase at the end still occurs due to computation).

6 Parallelization

A variety of techniques were applied to parallelize the algorithm. The most successful attempts are discussed within this section.

6.1 Control.Monad.Par(parMap)

```haskell
playMastermindParMap ::  Code -> Code -> Int -> CodeSet -> CodeSet -> IO Int
playMastermindParMap guess solution k fullSet possibleSet = do
  putStrLn $ "Guessing: " ++ show guess
  let response@(blk, wht) = guessResult guess solution
  putStrLn $
    "Response: " ++ show blk ++ " black and "
      ++ show wht ++ " white"
  if blk == length guess then do
    putStrLn $ "Solved: " ++ show guess
    return k
  else do
    let possibleSet' = filterCodeSet possibleSet guess response
    let possibilities = runPar $ parMap (scoreGuess possibleSet') fullSet
    let (_, _, nextGuess) = minimum possibilities
    playMastermindParMap nextGuess solution (k + 1) fullSet possibleSet'
```

[3]https://stackoverflow.com/questions/40948153/find-min-elements-index-of-a-large-list-in-haskell

The main change here is that the *map* over candidate guesses has been replaced with *parMap*. The result is a 21 second runtime on 8 cores, representing a performance improvement factor of 2.38. Using different numbers of cores affected this performance, as shown in the graph below. As seen, increasing from 1 to 4 cores caused a significant performance improvement. Increasing past 4 cores results in minimal further improvement. This is likely because our algorithm is largely CPU-bound and the machine only has 4 physical cores mapped to 8 logical cores.

At attempt was also made at using *parMap* within the *scoreGuess* function to parallelize the evaluation of each guess-solution pair. However, the result was extremely slow performance and a 1.2 GB event log. Evidently, too many sparks were created.

Even with the successful parallelization above, a large number of sparks are created.

6.2 Chunks

Aiming to create fewer sparks, we designed a parallelization strategy which splits the candidate codes into n chunks and then parallelizes computation over those chunks.

```haskell
bestFromChunk :: CodeSet -> CodeSet -> Possibility
bestFromChunk possibleSet chunk = foldl1' min $ map (scoreGuess possibleSet) chunk

-- In chunks
playMastermindChunkStrategy ::  Int -> Code -> Code -> Int -> CodeSet -> CodeSet -> IO Int
playMastermindChunkStrategy numChunks guess solution k fullSet possibleSet = do
  putStrLn $ "Guessing: " ++ show guess
  let response@(blk, wht) = guessResult guess solution
  putStrLn $
    "Response: " ++ show blk ++ " black and "
      ++ show wht ++ " white"
  if blk == length guess then do
    putStrLn $ "Solved: " ++ show guess
    return k
  else do
    let possibleSet' = filterCodeSet possibleSet guess response
    let chunks = splitToChunks numChunks fullSet -- TODO: Tune the number of chunks
    let possibilities = map (bestFromChunk possibleSet') chunks `using` parList rseq
    let (_, _, nextGuess) = foldl1' min possibilities
    playMastermindChunkStrategy numChunks nextGuess solution (k + 1) fullSet possibleSet'
```

Notice that this strategy, aside from splitting the scoring of candidate codes into parallelized chunks, also selects the best code from each chunk. Then the best of these best-in-chunk codes is selected as the next guess. Doing so reduces the peak memory usage of the algorithm, contributing to the improved performance. Below, a Threadscope screenshot with $n = 64$ chunks is shown. The run time is 17 seconds, representing a performance improvement of 2.94 times over the sequential implementation and 1.18 times over the parMap implementation running on 8 cores. Notice however that the computational load is not balanced evenly between cores. Increasing the number of chunks allows for better load balancing, increasing performance. The screenshot below shows the distribution of work between cores with $n = 512$ chunks. The algorithm is able to finish in 15 seconds. Increasing n past 512 results in minimal performance gains, as the load is already evenly balanced.

It is also important to consider that on later turns, the possible solution set P has shrunk. Thus, fewer chunks should be necessary for proper load balancing. An implementation was tested with various n which split computation into $\frac{n}{2^k}$ chunks on turn k, however performance dim

7 Conclusion

To conclude, the chunked strategy was able to parallelize the Master Mind algorithm to run almost 3 times faster than the sequential version. We found that for our test setup, splitting computation at each turn into 512 chunks allowed for maximum performance. However, we suspect that the optimal number will differ based on the number of computational cores available. The game configuration (number of colors and holes) may also have an impact on determining how many chunks should be used. An interesting further investigation would be to have the algorithm adaptively determine how many chunks to use at each turn based on the game configuration, number of possible solutions remaining, and available hardware resources.

8 Appendix: Code

TODO