Examining the Perfect Match Problem

Matthew Weston

I. Introduction

The subject of our analysis is a television show in which eleven couples must be identified by a contestant. This contestant is able to observe two sets of eleven people each, knowing that each person in one set is the ”Perfect Match” of one person in the other, but does not know who should be matched with who. Each week, the contestant is given one guess at the ideal pairing, after which the number of correct matches is revealed to both the contestant and the audience. The contestant’s task, then, is to iteratively attempt to obtain a flawless matching of all eleven couples in as few weeks as possible. In this writeup, we make note of three interesting routes towards solving the problem, and present a demonstrate that allows us to evaluate any given implementation.

II. Problem Analysis

To begin, we must specify this problem in a mathematical sense. Predicting couples based on their chemistry or appearance using machine learning and facial recognition is rather out of scope right now, as interesting as that might be, so we will assume that each individual here can be perfectly represented by a unique integer. Further, since the order of the pairings does not matter, we can treat one of the two spouse lists as static, and simply search for an ideal ordering of the other list (pairing elements with the same index across the two lists). Then, we can say that this problem can be described as follows:

1) We begin with a list of unique integers A, having size S.
2) There is a unique position in A for each value therein which ’perfectly matches’ it.
3) We may repeatedly guess at an ordering of A, receiving a score V that tells us how many elements are in the correct position.
4) We aim to achieve a guess for which V=S in as few attempts as possible.

We can note here that this problem resembles Mastermind, a popular code-breaking game that has been well explored in Computer Science research. There are, however, two key differences. The first is that repeat guesses are not allowed (we cannot pair one spouse in one set with two in the other set, after all). The second is that, as a side effect of this, we do not have a ’partially correct’ indicator that tells us whether an element is present in the solution but in the wrong place. Nonetheless, this is as good a lead as any, so we begin by examining how AIs for Mastermind have been built.

A. Evaluation Methodology

To aid in our analysis of solutions (and make this paper more interesting to read), we have implemented a simulator for the task at hand, shown in figure 1. This simulator runs an algorithm, specified as a class that implements the interface specified in figure 2, over a certain number of trials t (in our case, t=10000), and averages the running time. This allows us to quickly determine the overall effectiveness of a given solution, not just theoretically, but in practice.

```
# Run a simulation
def runSimulation(size, sol):
  solution = np.arange(size)
  np.random.shuffle(solution)
  model = sol(size)
  attempts = 0
  attempt = None
  prev_score = None
  while (True):
    attempts += 1
    attempt = model.predict(prev_score)
    prev_score = np.sum(attempt==solution)
    if (prev_score == size):
      break
  return attempts # number of attempts

def avgSimulation(size, sol, trials=10000):
  sum = 0
  for i in range(trials):
    sum += runSimulation(size, sol)
  return sum / trials
```

Fig. 1. The code for our simulator.

```
class Solution:
  def __init__(self, size):
    self.size = size
    pass
  def predict(self, prev_score):
    pred = np.range(self.size)
    np.random.shuffle(pred)
    return pred # make a random guess
```

Fig. 2. The interface implemented by any algorithm aiming to solve the Perfect Match problem.

III. Algorithmic Approaches

The legendary Donald Knuth comes through for us. His 1976 paper, ”The computer as master mind”, outlines a worst-case optimal algorithm for determining the solution to a mastermind game of size S=4 in at most five guesses [1]. This solution involves the use of a minimax algorithm that selects a guess that will, at worst, eliminate at least as many possible solutions as any other guess would. There are, however, two caveats:

The first is that the first guess *must* by 1122, which repeats a guess, for the provable ”five or less tries” bound to hold. Luckily, this does not impact us, since, because repeated guesses are disallowed, we have no priors, and the assignment of people to integers is arbitrary, every possible first guess is equally valid. We therefore have no need to concern ourselves with what our first guess should be.

The second is that we must occasionally make an impossible guess, if doing so would more effectively narrow down the solution set. This means that, even if we know that a guess cannot be correct, it may still be the guess that tells us the most about where to find the correct one.

A. *Baseline: A brute force solution*

To begin, we evaluate a brute force solution, implemented in figure 3. For the sake of running time, we limit ourselves to a size of six, rather than eleven - there are $11! = 39916800$ possible orderings in that case, and our below algorithmic solutions, though they would not require nearly that many guesses, are at least linear in compute time with the number of possible orderings when making each guess. Note that there is a solution that sidesteps this issue and would be substantially more viable for rapidly conducting many computations for high values of S - we discuss it after exhausting non-machine-learning-based solutions.

```
class ExhaustiveSolution(Solution):
  def __init__(self, size):
    self.size = size
    self.count = 0
    self.num_pos = np.math.factorial(self.size)
    self.range = list(range(size))
  def predict(self, prev_score):
    prediction = []
    x = self.size
    rem = self.count
    l = self.range.copy()
    while (x > 0):
      f = np.math.factorial(x-1)
      ix = rem // f
      prediction.append(l[ix])
      del l[ix]
      rem = rem % f
      x -= 1
    self.count += 1
    return np.array(prediction)
```

Fig. 3. A brute force solution for Perfect Match, which iteratively attempts every possible ordering until success is achieved.

This solution achieves an average performance of 360 guesses before a correct ordering, which is half of $6!$, or 720. Needless to say, we can improve upon it.

B. *Baseline II: A smarter brute force solution*

The most obvious improvement to make, when establishing an improved baseline, is eliminating guesses that our previous guesses have shown cannot be the correct solution. This can be done in relatively simple fashion - since score is the number of correct matches, we can say that any potential solution that shares fewer pairings with our previous guess than our previous guess's score cannot be correct. Similarly, any potential solution that has fewer different pairings than we need points must also be wrong - a near-exact duplicate of a completely wrong answer can't be correct. We must have exactly as many differences in position as we need points.

This optimization alone reduces the average number of guesses for a problem of size six from 360 to 5.8899. Code is provided on Github [1], to conserve space.

C. *Knuth's MiniMax Algorithm*

The next improvement to make is adding Knuth's MiniMax Algorithm in full. To achieve this, we evaluate each permutation of our variables against each remaining *possible* permutation, aiming to minimize the maximum number of possible solutions after the next guess. The result we obtain is 5.6592, which, while not especially large, does demonstrate that we are able to achieve some measure of improvement through minimax. As discussed later, this comes at significant cost in terms of computational complexity, however.

Note that, here, we optimize to reduce the worst case, while our metric evaluates the average case. It is possible that optimizing against the average case would allow for better performance, though I do not anticipate that the difference in performance would be large.

IV. Black-peg Mastermind Variant

Michael Goodrich introduces a variant of Mastermind in which, similar to our own variant, there is only a single 'score' value provided after each guess. He provides an algorithm for single-count match queries, beginning with guesses that repeat each element exclusively, in order to identify each element's cardinality. This is not relevant to the problem we aim to solve. He then resolves the remainder of the problem using a divide and conquer algorithm, attempting to solve for each half of the solution independently, once again making use of cardinality. While a larger portion of his algorithm depends on the possibility of repeated guesses, it may be worthwhile to examine the usefulness of divide and conquer algorithms here.

V. Genetic Algorithms

While the use of machine learning may seem like overkill for such a simply-written problem, genetic algorithms have shown ([3]) to be very effective for solving Mastermind in a very scalable fashion - [2] even models his black-peg Mastermind variant as identifying a unique genome by subverting a privacy-preserving homeomorphic cryptography mechanism. The most direct implementation of Knuth's algorithm, with no optimization, has a running time of $S**2 - SE$ *per-guess*, where E is the number of eliminated possibilities, since every possible permutation of inputs must be examined against every non-eliminated permutation to determine which guess will be the most effective at eliminating a guaranteed large portion of

[1]https://github.com/MatthewCWeston/Perfect_Match

the dataset. The computational complexity of a large set of pairings quickly increases.

Genetic algorithms, on the other hand, simply propose a solution, examine its score, and propose new solutions that resemble existing ones. At no point does the full solution-space need to be examined at all, making them ideal for large-scale code-breaking problems. The power and relatively low spinup time of a genetic algorithm from a generic library mean that, should someone need to solve Perfect Match very quickly at a very large scale, they may be the ideal solution.

References

[1] Knuth, Donald E. ”The computer as master mind.” Journal of Recreational Mathematics 9.1 (1976): 1-6.

[2] Goodrich, Michael T. ”On the algorithmic complexity of the Mastermind game with black-peg results.” Information Processing Letters 109.13 (2009): 675-678.

[3] Berghman, Lotte, Dries Goossens, and Roel Leus. ”Efficient solutions for Mastermind using genetic algorithms.” Computers & operations research 36.6 (2009): 1880-1885.