EECS 126 - Probability and Random Processes

Mastering MasterMind with MCMCs

Nathan Blair, Adarsh Karnati, Darren Lee, Jonathan Lee
April 18, 2018

1 Introduction

In this project, we show how to use the MCMC to solve the classic game, MasterMind[**1**]. MasterMind is a simple game in which a Codebreaker attempts to guess a code that a Codemaker has secretly made, using feedback on each guess. In the original game, the codes are made by choosing a sequence four of six possible colors (repeated colors are allowed) and the Codebreaker has ten tries to find the code. After each guess, the Codemaker will return a reply with a number of black pegs representing the number of colors in the guess placed in the correct position, and a number of white pegs representing the number of correct colors in incorrect positions - information regarding the locations of the correct and incorrect colors is withheld. The Codebreaker then tries to guess the code in the fewest number of guesses given the feedback. The first algorithm guaranteed to solve the classic game in less than five moves was developed by Knuth and the current record is an average of 4.34.

Our results show that using MCMC to generate guesses for the Codebreaker allows for a significant reduction in the size of the state space searched while still requiring relatively few guesses to win. Empirically, our algorithm explores a constant multiple of the square root of the size of the state space, meaning that for larger instances of the MasterMind game, our algorithm provides a significant speed up when compared to traditional algorithms.

We apply this speedup to a variant of the MasterMind problem that reconstructs images. Similar to the traditional variant of MasterMind, a Codemaker constructs a secret image using n pixels and m colors. Then, our algorithm reconstructs the image by treating each pixel as a peg in the traditional mastermind formulation. Using this method, our algorithm is able to efficiently reconstruct small images with very limited information about the properties of the image itself.

2 Methods

The main idea we drew from Snyder's paper[**2**] was the idea of **consistency** of candidate codes with regards to the accumulated evidence. For example, if we received $(B = 2, W = 0)$ from our initial guess, then we would need a code that would produce a response of $(B = 2, W = 0)$ to have a chance of being the secret code. Our notion of consistency was fixing a candidate code as the secret code and comparing the responses of our previous guesses under the candidate code being the secret code to the true responses of the previous guesses. Specifically, we chose to compare the sums of the absolute difference in black pegs, absolute difference in white pegs and two times the absolute difference in total pegs. The sum of this cost function applied to each of our guesses produced our total cost function, and we exponentiated to the negative cost in order to normalize, giving us a scoring function. When a code x is consistent with all the responses so far, then the cost will be zero, and the score will be one.

$$\mathsf{Score(x)} = e^{-\mathsf{Cost}(x)}$$

$$\mathsf{Cost}(x) = |B_{\text{diff}}| + |W_{\text{diff}}| + 2 \times |T_{\text{diff}}|$$

We used MCMC to produce a random walk across the code state space in a way that positions the current state closer to the secret code, rather than use the algorithm to estimate a probability distribution. In our algorithm, it is possible to accept a proposal state that is not consistent, as this could possibly move the current state towards consistent codes. However, once the current state becomes a consistent code, we immediately play it. To cap computation time, we incorporated a **burn in period**, which limits the maximum number of iterations of MCMC before returning the current state. This period plays a tradeoff between making consistent guesses versus playing an inconsistent guess to gain more information. It also ensures that our algorithm does not get stuck in an area with inconsistent states.

The scoring function mentioned above was used to define the MCMC acceptance probability for a proposal and current state. The proposal function we decided to use modified a code by peg two pegs or replacing a color with probability 1/2. This proposal function preserves the simulation of a random walk by incorporating our idea of codes that are "close" to each other. One important point is that when the total number of pegs from a guess equals the total pegs of the game, we change the permutation probability to 1, since we have all the right colors and they just need to be ordered. Below is pseudocode for our MCMC algorithm:

```
def run_mcmc(GUESS, GUESS_DICT):
    curr_state = GUESS
    for i in 1 to BURN_IN_PERIOD:
        proposed_state = propose(curr_state)
        if accept(proposed_state, curr_state, GUESS_DICT):
            curr_state = proposed_state
        if is_consistent(curr_state, guess_dict):
            break
    return curr_state

def simulate_game(SECRET_CODE):
    guess_dict = {}
    guess = get_init_guess()
    guess_dict[guess] = evaluate_guess(guess, SECRET_CODE)
    while guess is not SECRET_CODE:
        guess = run_mcmc(guess, guess_dict)
        guess_dict[guess] = evaluate_guess(guess, SECRET_CODE)
    return
```

3 Experiments

Our first experiment was running the original methods on the **classic board game** (4 pegs and 6 colors). Initially, we thought that the previous guess state would hold enough information regarding consistency as we proposed new states, but our prototype struggled to converge on the solution. To maintain tractability and ensure improvement in guesses, we tried to keep track of the current best state based on a heuristic on the number of black pegs; however, this did not improve results significantly. We decided to sacrifice computation time and check for consistency between every guess made, as described in

the Methods section. This change dramatically improved runtime for lower dimension games.

The natural extension after this was **increasing the number of pegs and colors**, for which mastermind is known to be an NP-hard problem. With higher dimension state spaces we ran across the issue of falling into local minima, where the current state would not change very much at each guess, but the current guess would be very close to the secret code. This issue was handled in two ways. Firstly, when the number of black pegs for the most recent guess hit some high fraction of the total pegs (80%), the probability of permutation was set very low. This forced the proposal function to value maintaining black pegs over getting white pegs, which would result from permutations. Secondly, at some higher threshold the algorithm would stop MCMC and move to a brute force method, where all possible remaining consistent codes are attempted. It may seem that this would dramatically increase computation time, but because the algorithm is only implemented near an almost perfect code, complexity is actually reduced. These two fixes allowed for much faster computation of high dimension games and fewer overall guesses.

Our next experiment was testing the robustness of our algorithm against **false information**. With some probability, each guess received a reply that was randomly chosen, rather than the true reply. This could either introduce incorrect candidate solutions that are deemed consistent and accepted - leading the walk in a potentially wrong direction, or it could constrain the candidate solution space - leaving no consistent solutions available. To combat these hazards, we implemented a burn-in loop that runs for some maximum number of iterations (≈ 500) to allow the walk to converge on a good, but inconsistent guess state with high probability (with our specified score function, our walk will spend more time in better states). With this addition, we saw that our algorithm was still able to find the secret code, with minimal extra guesses.

Finally, we extended our project to an **image guessing game**, which comprised of a 45 peg, 3 color game. This has been captured as a GIF in the results section.

4 Results

For the standard game of MasterMind with 4 pegs and 6 colors, we achieve an average of 4.687 guesses per game [4.1]. By comparison, the best known algorithm for the standard game of MasterMind averages 4.340 guesses, using decision trees based on exhaustive search. Running our algorithm on 10000 trials takes approximately 5 minutes, so on average each game takes 0.03 seconds to solve.

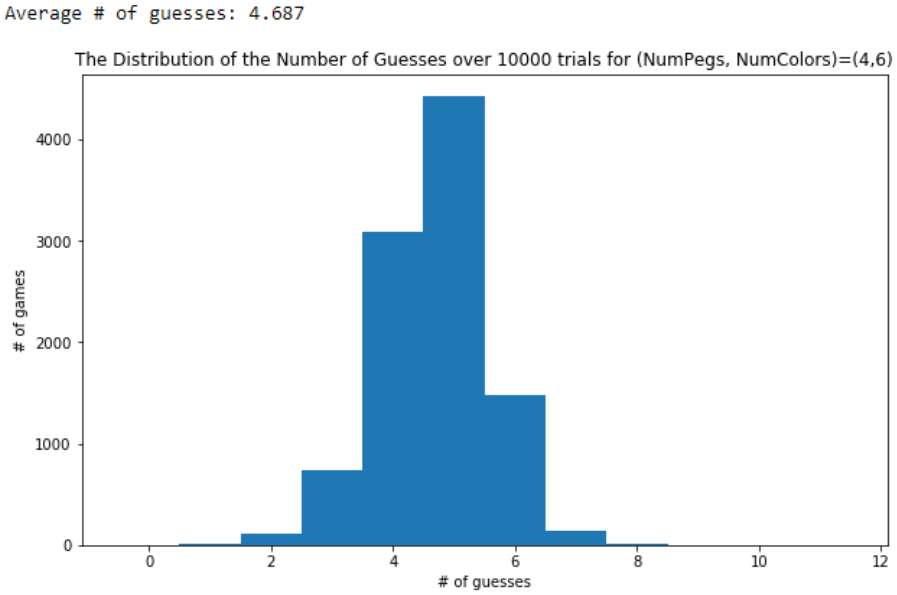


Figure 4.1: Number of guesses for standard game of MasterMind

We also looked at the tractability of our algorithm by increasing the number of colors and the number of pegs, and recording the average number of guesses need to win each game. We found that increasing the number of pegs dramatically increased the number of guesses necessary, while the number of colors increased the number of guesses at a slower rate. This makes sense, because increasing the number of pegs increases the size of the state space at an exponential rate, while increasing the number of colors increases the size of the state space at a polynomial rate ([4.2], [4.3]).

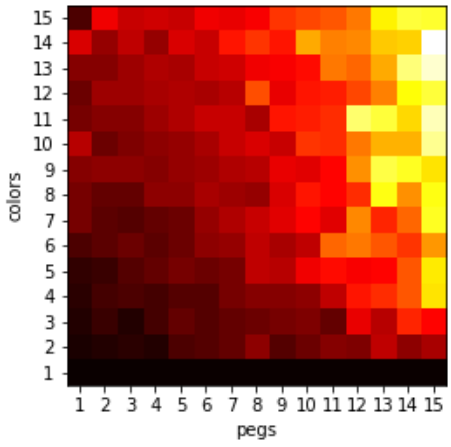
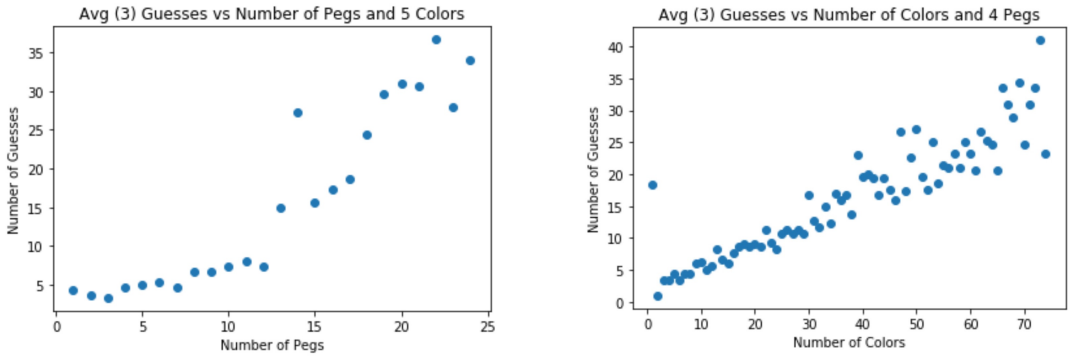


Figure 4.2: Heat map of number of guesses with variable number of colors/pegs. Hotter (*white* = 28 *guesses*) colors denote more guesses; Colder (*black* = 1 *guess*) colors denote fewer guesses.

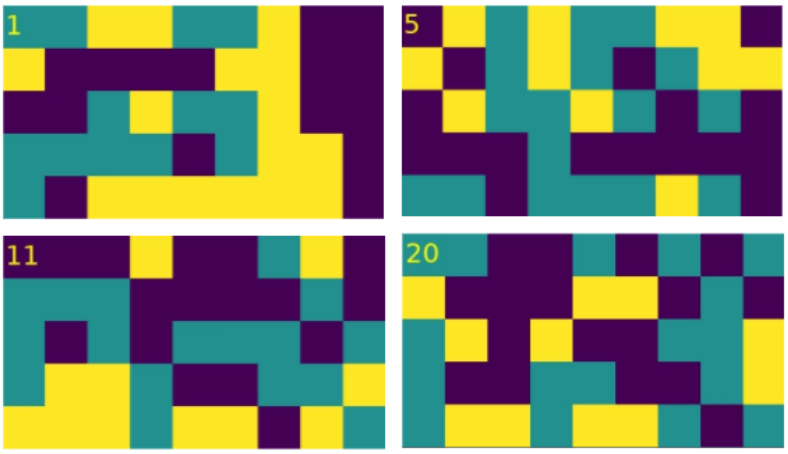


(a) (b)

Figure 4.3: Avg number of guesses vs number of colors/pegs

We also ran our MasterMind solver on images. In this demonstration, we generate a 5x9 image with 3 colors, so we have 45 pegs and 3 colors. Our solver averages 45 guesses to guess the image and completes in approximately 2 minutes. Here is a gif of the full guess evolution for a game that completed in 40 guesses: [gfycat]. Figure 5 ([4.5]) displays selected guesses to the secret code displayed in.

Figure 4.4: Secret image



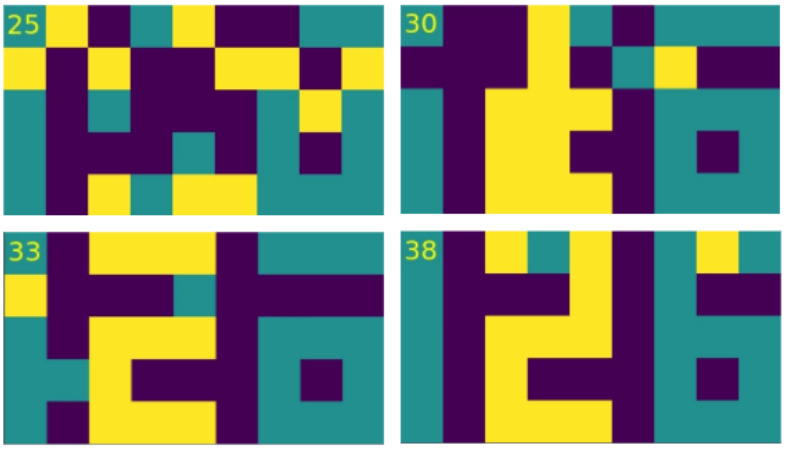


Figure 4.5: Selected screenshots of guesses. Number in the top left is the guess #.

5 Discussion / Limitations

While our algorithm provides a significant reduction in the size of the state space explored, we are still left with an exponentially sized exploration space. This is due to the fact that the Codebreaker has extremely limited information about the secret being decoded. This puts our algorithm in an interesting position. On one hand, it is more efficient than solving the problem outright. On the other, our algorithm is still intractable for very large spaces. Because of this, our algorithm is applicable for a unique selection of MasterMind problems, that are too hard to solve outright, but are easy enough to explore a medium fraction of the state space.

One extension of this project that we could explore is using more information about the secret to solve more applicable problems. One example of this comes in the form of generative networks, where we could replace the codemaker with a classifier and allow the codebreaker to guess sequences that it thinks will be more likely to be classified as good. Consider a cat-image classifier, that scores images on their resemblance to cats. We could use this as our codebreaker while an image generator uses MCMC to try to create cat images that will score higher in the eyes of the classifier.

On the flip side, using a function of black and white peg responses would also be interesting extension, similar perhaps to the aquasport game "Marco-Polo". Additionally, only receiving black or only receiving white pegs could give insight into how valuable each response type is to solving the game. An interesting aspect to how MCMC is using the information in the algorithm is that most of the work is done in the initial guesses, dramatically reducing the state space by applying the consistency check. This essentially squeezes the most information out of a very sparse evidence pool, and may be applicable in situations where incorrect estimates are dramatically penalized.

References

[1] https://en.wikipedia.org/wiki/Mastermind_(board_game)

[2] http://www.webpages.uidaho.edu/~stevel/565/projects.old/mastermind.pdf