Mastermind by Importance Sampling and Metropolis-Hastings

by Angela B. Snyder

MASTERS THESIS

Advisor: Marc Adlai Coram

Abstract

Two machine learning algorithms are introduced for playing a one player version of the logical game Mastermind. In the classic game of Mastermind there are two players: the encoder and the decoder. The encoder builds a secret code by selecting a sequence of four pegs, each chosen from six different colors. The decoder then attempts to guess the secret code in as few turns as possible. In this version, the secret code is generated uniformly at random from the set of all possible codes and the computer must guess this code in order to win the game.

The first strategy presented makes guesses according to an exponential distribution on a simple set of features. The model parameters were trained by performing a gradient descent algorithm on an importance sampling estimate of the mean number of turns required to win the game.

The second strategy is based on the Metropolis-Hastings algorithm, and uses a simple cost function to evaluate each code proposed. This strategy requires more turns on average to complete the game, but provides a significant reduction in the number of codes that must be evaluated at each turn. This reduction in computation allows for fast play on versions of the game with more pegs and more colors.

Lastly, the performance of the Metropolis-Hastings based strategy is examined on a modified version of the game where false information is given with some positive probability, and the performance is compared to that of human subjects.

Contents

1 Problem and Goals **5**

1.1 The Game 5

1.2 Overview of My Approach 6

2 History **7**

3 Theory and Mathematics **10**

3.1 Importance Sampling 10

3.1.1 The Importance Sampling Function 10

3.1.2 The Normalizing Constant 11

3.1.3 Importance Sampling in Optimization 13

3.2 Metropolis-Hastings 15

3.2.1 The Normalizing Constant 16

4 General Approach **17**

4.1 Definitions and Notation 17

4.1.1 Game Terminology 17

4.1.2 Versions of the Game 18

4.2 CASE 1. A Simple Strategy for Standard Game Play 19

4.2.1 A Model For Strategy 19

4.2.2 Feature Selection 20

4.2.3 Learning the Strategic Parameters 28
4.2.4 Performance 36
4.3 CASE 2. More Complex Games 37
4.3.1 A Cost Function That Promotes Mixing 39
4.3.2 Metropolis-Hastings Implementation 43
4.3.3 Tricks for Efficient Game Play 46
4.3.4 The Problem of Stopping Time 48
4.3.5 The Problem of Sticking 51
4.3.6 Metropolis-Hastings: A Smarter Search 52
4.3.7 Selecting the Model Parameter 54
4.3.8 Performance 56
4.4 CASE 3: False Information 58
4.4.1 Performance 60
4.4.2 Humans as Master Mind 60
5 Summary **63**

1 Problem and Goals

1.1 The Game

In the classic game of Mastermind there are two players: the **encoder** and the **decoder**. The encoder builds a **secret code** by selecting a sequence of four pegs, each chosen from six different colors. The decoder then attempts to guess the secret code in as few turns as possible.

For each turn, the decoder guesses a code and then receives feedback in the form of black and white pegs. This feedback describes the number of pegs in the guess that were the correct color and in the correct position (represented by black pegs), and the number of pegs in the guess that are of the correct color but in the wrong position (represented by white pegs). Feedback pegs are placed to the right of each guess on the playing board, and the absence of feedback pegs implies that none of the pegs in that guess are of the correct color. There are fourteen possible feedback responses. Note that the response "3 black, 1 white" is not possible. The decoder has eight turns to guess the secret code and win the game.

1.2 Overview of My Approach

In this paper, two algorithms are presented for playing a one player version of Mastermind, where the secret code is generated uniformly at random from the set of all possible codes.

The first strategy is represented by an exponential scoring function on a simple set of features. Game data was simulated under an initial model, and then a gradient descent algorithm on the importance sampling function was used to train the model parameters.

The second strategy is based on the Metropolis-Hastings algorithm, and uses a simple scoring function to evaluate potential codes. This algorithm takes more turns on average to complete the game, but provides a significant reduction in the number of codes that must be evaluated before each turn. This reduction in computation allows play on larger, more complex versions of the game.

The performance of each of these algorithms is evaluated through game simulation. The performance of the Metropolis-Hastings based algorithm on a modified version of the game that allows false replies with some positive probability is examined. Lastly, the performance of the two algorithms is compared to that of human subjects.

2 History

A good amount of research has been published on the topic of Mastermind. The first article published on the subject, "The Computer as Mastermind", by Donald E. Knuth, appeared in the Journal of Recreational Mathematics 1976 [6]. In this paper, Knuth described a strategy that can win the game in at most six guesses. This strategy was formed by always selecting the guess that will minimize the maximum number of remaining possibilities over all fourteen possible responses. The strategy is presented as a decision tree, with a static first guess and each subsequent guess contingent on the previous reply received. This strategy always wins the game in six moves or less, and it was later shown that it takes 4.478 guesses on average. Knuth also presented an alternate strategy that was guaranteed to win in five moves, but required a higher number of guesses on average.

In 1978, Robert W. Irving presented a strategy that required only 4.369 moves on average [5]. He used a greedy algorithm to minimize the expected number of remaining possibilities on the first two guesses, and then performed an exhaustive search to select the optimal subsequent guesses.

The game of Mastermind was "solved" in 1994, when Kenji Koyama and Tony W. Lai presented the optimal decision-tree strategy based on exhaustive

search [7]. They incorporated symmetries and other optimization tricks to reduce computation time. Their resulting strategy requires 4.340 turns on average to win and always wins in at most six moves. They also presented a modified game strategy that always wins in at most five moves and takes 4.341 moves on average.

In 1996, Bernier and Merelo published "Solving Mastermind Using GAs and Simulated Annealing: A Case of Dynamic Constraint Optimization" in Parallel Problem Solving From Nature (PPSN) [1]. They presented three algorithms: a random search algorithm with constraints, a genetic algorithm and a simulated annealing algorithm. Each algorithm was evaluated based on the number of guesses made to win and the number of combinations examined throughout the game, for games consisting of six colors and from four to ten pegs. Their random search and genetic algorithms always play a move that is consistent with all of the information received, while their simulated annealing algorithm plays the best move obtained within a set amount of time. The random search algorithm selects codes uniformly at random and checks them for consistency. The simulated annealing algorithm uses both permutation and mutation to create new combinations from previous ones and evaluates combinations by a cost function that measures their level of

consistency. They found that their genetic algorithm outperformed the random search algorithm in terms of the number of combinations examined, and that their simulated annealing algorithm played much faster and provided a sufficiently good solution to the game.

3 Theory and Mathematics

3.1 Importance Sampling

Importance sampling is a numerical method for approximating an integral. It can be implemented to estimate the mean response for a given sample under an alternate distribution [3].

3.1.1 The Importance Sampling Function

For the general case, suppose that we wish to estimate the integral

$$\theta = \int f_1(x)dx. \tag{1}$$

Then for any function $f_2(\cdot)$ we have that

$$\theta = \int \frac{f_2(x)}{f_2(x)} f_1(x)dx = \int \frac{f_1(x)}{f_2(x)} f_2(x)dx = \int \frac{f_1(x)}{f_2(x)} dF_2(x) \ , \tag{2}$$

where

$$F_2(x) = \int_{-\infty}^{x} f_2(y)dy. \tag{3}$$

Note that if $f_1(\cdot)$ and $f_2(\cdot)$ are both probability distributions, then $F_2(\cdot)$ is simply the cumulative distribution function for the density $f_2(\cdot)$.

Now suppose we are interested in estimating

$$\mu = E_{F_2}[h(x)] = \int h(x) f_2(x) dx. \quad (4)$$

We begin the importance sampling procedure by drawing $x_1, ..., x_m$ independently from some trial distribution $F_1(\cdot)$. We then calculate the importance weights for each sample point

$$w_j = \frac{f_2(x_j)}{f_1(x_j)}, \text{ for j=1,...,m} \quad . \quad (5)$$

Finally, we approximate μ by

$$\hat{\mu} = \frac{w_1 h(x_1) + \cdots + w_m h(x_m)}{w_1 + \cdots + w_m} \quad . \quad (6)$$

The idea here is that we can estimate the mean under an alternate distribution with a weighted average of the sample responses. We up-weight those samples which are more likely under the desired distribution, and down-weight those that are less likely under that distribution.

It should be noted that choosing a trial distribution $f_1(\cdot)$ that is close to our desired distribution $f_2(\cdot)$ will reduce the error of our estimate [8].

3.1.2 The Normalizing Constant

A nice feature of the importance sampling estimate is that there is no need to calculate a normalizing constant when our probabilities under $f_1(\cdot)$ and

$f_2(\cdot)$ can be expressed in the form:

$$f(x) = \frac{s(x)}{\sum_{y \epsilon \Omega} s(y)} \ , \tag{7}$$

where $s(\cdot)$ is any scoring function and Ω is our sample space. When the probabilities can be expressed in this way, we need only calculate $f_2(x)/f_1(x)$ up to a multiplicative constant in order to obtain the importance sampling estimate.

To understand why, note that

$$w_j = \frac{f_2(x_j)}{f_1(x_j)} = \frac{s_2(x_j)/S_2}{s_1(x_j)/S_1} = \frac{S_1}{S_2} \cdot \frac{s_2(x_j)}{s_1(x_j)} = S \cdot \tilde{w}_j \ , \tag{8}$$

where $S = S_1/S_2$ and $\tilde{w}_j = s_2(x_j)/s_1(x_j)$.

Our estimate for μ therefore becomes

$$\begin{aligned}
\hat{\mu} &= \frac{w_1 h(x_1) + \cdots + w_m h(x_m)}{w_1 + \cdots + w_m} && (9) \\
&= \frac{S\tilde{w}_1 h(x_1) + \cdots + S\tilde{w}_m h(x_m)}{S\tilde{w}_1 + \cdots + S\tilde{w}_m} \text{ , by substituting } w_j = S\tilde{w}_j && (10) \\
&= \frac{S\Big(\tilde{w}_1 h(x_1) + \cdots + \tilde{w}_m h(x_m)\Big)}{S\Big(\tilde{w}_1 + \cdots + \tilde{w}_m\Big)} && (11) \\
&= \frac{\tilde{w}_1 h(x_1) + \cdots + \tilde{w}_m h(x_m)}{\tilde{w}_1 + \cdots + \tilde{w}_m} && (12)
\end{aligned}$$

By these calculations, we only need to know the numerators of the score functions $s_1(\cdot)$ and $s_2(\cdot)$ in order to obtain the importance sampling estimate

of the mean μ. We do not have to compute the normalizing constants S_1 and S_2.

3.1.3 Importance Sampling in Optimization

Importance sampling gives us a way to estimate the mean response under an alternate distribution $f_2(\cdot)$. We may be interested in determining which distribution within a family of distributions will yield the optimal mean response. To that end, we may seek a distribution $\hat{f}_2(\cdot)$ that will minimize (or maximize) the importance sampling estimate of the mean.

In the parametric setting, assume that $f_2(\cdot)$ is a function of a parameter θ which may be high-dimensional. Suppose we wish to find $\hat{\theta}$ such that the importance sampling estimate based on a fixed trial sample $x_1 \ldots x_m$ is minimized. For sufficiently nice families of distributions, we can calculate the gradient of the importance sampling estimate as a function of θ and obtain the minimizer. Alternately, if the gradient is difficult to calculate, we may use a numerical procedure like **gradient descent** to find a minimizer.

One algorithm for gradient descent begins with an initial value of θ, say θ_0. Next,

1. Calculate or approximate $\Delta(\theta)$, the gradient of the importance sam-

pling function at the current value of θ.

2. Replace θ with $\theta_{new} = \theta - \alpha \cdot \Delta(\theta)$, where α defines the size of the update steps in the algorithm.

3. Repeat steps 1 and 2 until the process becomes stable.

Unfortunately, our estimate for $\hat{\theta}$ could have large bias if our maximizing distribution $\hat{f}_2(\cdot)$ is far from our trial distribution $f_1(\cdot)$. More specifically, the optimal distribution constructed from a single trial sample will put all of its weight on those samples with the largest response value. One way to circumvent this problem is to build new trial samples at each step of a gradient descent algorithm. In other words, we evaluate the gradient at each step of the process and update our distribution in the direction of the gradient. We then resample from this new distribution and repeat the process. Our maximizer is selected when the process becomes stable by some predetermined criteria. Henceforth, this procedure will be referred to as **sequential importance sampling**.

3.2 Metropolis-Hastings

The Metropolis-Hastings algorithm is a means of simulating a random sample from a probability distribution [2]. In this procedure, we perform a random walk over the sample space, where each step consists of a proposal of a new state and a subsequent acceptance or rejection of the proposed state according to probabilities defined by the desired distribution. We stop the process after a sufficient number of steps (or **burn-in period**), and observe the current state. In the limit, this observation will be a sample from the desired distribution.

Under Metropolis sampling, we have a proposal distribution $f(\cdot)$, and the probability of accepting a proposed state y_2 when we are currently in state y_1 is given by

$$P(\text{accept}) = \min\left(\frac{f(y_2)}{f(y_1)}\ , 1\ \right). \tag{13}$$

In 1970, Hastings [4] generalized the Metropolis algorithm by framing it with an arbitrary transition probability function

$$q(y_1, y_2) = Pr(\text{moving from } y_1 \text{ to } y_2). \tag{14}$$

The acceptance probability for a proposed state then becomes

$$P(\text{accept}) = \min\left(\frac{f(y_2)q(y_2, y_1)}{f(y_1)q(y_1, y_2)}\ , 1\ \right). \tag{15}$$

3.2.1 The Normalizing Constant

One of the strong points of the Metropolis-Hastings algorithm is that it allows us to obtain a sample from a distribution without calculating the normalizing constant.

For instance, if we express our density $f(\cdot)$ in the form:

$$f(x) = \frac{s(x)}{\int s(y)dy} \ , \tag{16}$$

where $s(\cdot)$ is a corresponding score function, then the acceptance probability reduces to

$$\begin{aligned} P(\text{accept}) &= \min\Bigl(\frac{s(y_2)/\int s(t_2)dt_2}{s(y_1)/\int s(t_1)dt_1} \ , 1 \Bigr) && (17) \\ &= \min\Bigl(\frac{s(y_2)}{s(y_1)} \ , 1 \Bigr) && (18) \end{aligned}$$

Thus, it is not necessary to calculate the normalizing constant

$$S = \int s(x)dx \tag{19}$$

to perform the random walk. We need only obtain the numerator of the probability of the proposed and current states in order to determine with which probability we should accept the proposed state.

4 General Approach

4.1 Definitions and Notation

4.1.1 Game Terminology

Throughout this paper, the term **secret code** refers to the **code**, or ordered combination of colored pegs, that the decoder must guess in order to win the game. The set of all possible codes is called the **codeset**. During each **turn** in the game, the decoder must make a **guess** by choosing a code from the codeset.

In response to each guess, the decoder will receive a **reply**, which will consist of white pegs and/or black pegs, or no pegs at all. Each **black peg** represents one peg in the decoder's guess that is of the correct color and in the correct position when compared to the secret code. Each **white peg** represents a peg in the guess that is of the correct color but in the wrong position. The absence of pegs indicates that none of the colors in the decoder's guess were contained in the secret code.

A **valid code** is one which complies with all of the replies received thus far in the game, and is therefore a potential candidate for the secret code. An **invalid code** is one that is in conflict with at least one of the replies received

thus far in the game. The **valid codeset** is the portion of the codeset that is still valid.

4.1.2 Versions of the Game

The version of the game Mastermind studied here is a one-player version where the secret code is generated uniformly at random from the codeset. The game player must then guess the secret code by making a series of guesses and receiving replies to each guess.

The **complexity of a game** refers to the size of the codeset. Allowing more pegs or more colors in the codes will increase the size of the codeset, and therefore increase the complexity of the game. The **number of pegs** is the length of the codes, and the **number of colors** is the number of color choices for each peg. The **standard game** of Mastermind has codes of length four, where the color of each peg is selected with replacement from six colors.

A second version of the game, %-false, has false replies occur independently according to a Bernoulli distribution with parameter p. Each false reply is selected uniformly at random from the set of all possible replies, with two exceptions:

1. If the decoder guesses the secret code, the **winning reply** (all black pegs) will be given.

2. The winning reply will never be given unless the decoder guesses the secret code.

The reply "all black pegs except for one white peg" (e.g. three black and one white in the standard game) can never occur in a truthful game, since it is impossible to have all but one peg of the correct color and in the correct position, and then have the last peg be of the correct color but in the wrong position. For this reason, I have chosen to eliminate it from the set of possible replies in the p%-false game.

4.2 CASE 1. A Simple Strategy for Standard Game Play

4.2.1 A Model For Strategy

Game strategy is modeled as a series of stochastic decisions, where the probability of guessing a given code at a particular stage of the game is based on an exponential scoring model.

Under model parameter θ, a given four-peg code x_i with k-dimensional

feature set $f_i = (f_{i1}, ..., f_{ik})$ has scoring function

$$S_\theta(x_i) = exp\{\sum_{j=1}^{k} f_{ij}\theta_j\} \ , \tag{20}$$

and the corresponding probability of selecting that code is given by

$$P_\theta(x_i) = \frac{S_\theta(x_i)}{\sum_{j=1}^{n} S_\theta(x_j)} \ . \tag{21}$$

In this setting, every code receives a score at the beginning of each turn. An associated probability for choosing each code is calculated by normalizing these scores, and a guess is then selected at random according to this probability distribution. The response variable for each game is the total number of turns needed to win.

4.2.2 Feature Selection

Color Combinations

Two types of features are incorporated in the simple model. The first feature centers around the initial guess made in the game. It would seem likely that

some codes would be better first guesses than others. After taking symmetries into account, each code falls into one of the following categories:

- all four pegs are the same color
- three pegs of one color and one of another
- two pegs of one color and two of another
- two pegs of one color and one each of two other colors
- each peg is of a different color

An estimate of the effect that the color combination of the first guess would have on the eventual length of the game was needed. To do this, the expected proportion of the codeset that would be valid (ie. not yet ruled out) after first guessing a code of each color combination was calculated.

The intuition here is that we will be more likely to win in fewer turns if we have done a better job of thinning-down our valid codeset. The findings are shown in Table 1.

By this reasoning, a code containing three colors would be the optimal first choice, with only 14.3% of the code space remaining in expectation. In contrast, codes containing only one color (e.g. Red,Red,Red,Red) would be

Table 1: Expected proportion of codeset remaining

Color Combination of First Guess	# Colors	E(Prop'n Valid)
2,1,1	3	0.1429541
1,1,1,1	4	0.1452082
2,2	2	0.1578206
3,1	2	0.1820595
4	1	0.3950462

poor choices, with an expectation of 39.6% of the codeset not yet eliminated after the first guess. Based on this preliminary study, it should be clear that the number of colors contained in the first guess is important.

The number of colors in the first guess was therefore included as a model feature, and treated as a factor since the relationship did not appear to be linear. Accordingly, the following three indicator functions were built into the scoring function:

$$f_i = \mathbf{1}_{\text{first guess} \cap \text{code contains i colors}} \quad \text{, for i=1,2,3} \tag{22}$$

There is no indicator function included in the model for having 4 colors in the first guess, because $\{f_1 = f_2 = f_3 = 0\} \Rightarrow \{f_4 = 1\}$. In other words, f_4 is not free.

Consistency Features

Another feature included in the scoring model measures the consistency of a given code x with all past replies. If a code x concides with all of the past information received, we will call it a **candidate code**.

After each guess G in the game we receive a reply in terms of white and black pegs. A code x is considered consistent with that information when the reply to the previous guess G under secret code x would be the same as the actual reply received. Any code that does not match this reply could be ruled out as a potential secret code.

For example, if the previous response was two black pegs and no white pegs, then x would be consistent with that information if the previous guess and x share two pegs in common (same color and in the same position), and none of the other colors overlap (no pegs of the correct color but in the incorrect position).

After a series of n turns in the game, our measure of total consistency with all past information for a code x then becomes whether the response R_i to each of the n previous guesses would have been received if x were in fact the secret code. Thus a natural feature for consistency of a code x when the true secret code is S would be the sum of the difference in the number

of black pegs and the difference in the number of white pegs.

We could then sum the differences for each reply received in the game. A code is consistent if and only if this total difference is zero. We must eventually guess a consistent code in order to win the game, since the true secret code S^* will be in perfect agreement with all replies received. Considering that all codes are consistent at the first guess, a natural feature is therefore the indicator of total consistency. This will be annotated as f_4.

A preliminary simulation of size $n = 100,000$ was conducted to determine the success of the strategy which always chooses uniformly at random from the set of all codes which remain consistent at any given point in the game. This strategy requires 4.648 turns on average to complete the game, with standard deviation $\sigma = 0.882$. The maximum game length in the simulation was only seven games, which means that under the classic game rules which allow eight turns to win, this strategy would have won the game every time.

Thus a strategy that always guesses consistent codes is effective. However, the question arises as to whether it could be beneficial to choose inconsistent codes in the beginning stages of the game in an effort to gain more information.

To examine this possibility, three variations on the consistency feature

aimed to allow more freedom in the initial stages of the game were constructed.

1. **Allow Free Stages.** This feature assigns uniform probability to all codes during a fixed number of opening moves in the game. After the fixed number of turns, the strategy switches to prefer codes which are consistent with all information obtained.

2. **Distance From Compliance.** This feature considers $D(x)$ as a distance function, where higher values of the function mean that a code x is further away from the true secret code, and lower values imply more consistency.

3. **Distance/Free-Stage Combination.** This feature allows for selection of codes with higher "distances" from the true code in the initial stages of the game, and then prefers more consistent codes as the game progresses.

To test the usefulness of each of these features, cell means were calculated in a two-dimensional grid search using 1,000 game simulations at each grid point over a spectrum of parameter values for each feature against a spectrum of parameter values for the total consistency indicator. The parameter values

for the color combination features were set to zero, meaning all codes had uniform probability at the first guess.

The simulation results suggest that these three variations on the consistency feature can not offer any improvement in strategy over the indicator of total consistency, whether alone or in tandem with the total consistency indicator. Thus, only the total consistency historical indicator was incorporated into the final model.

Model Summary

The simple scoring model is

$$S_\theta(x_i) = \exp \sum_{j=1}^{4} f_j \theta_j \tag{23}$$

with the feature set f being a four-dimensional object consisting of three indicator functions for the number of colors in the first guess, and a fourth indicator of total consistency with all past information in the game. Here, θ is the strategic parameter vector.

Under this model, the exponent of the score function for a guess x_i on turn i can be partitioned into the color combination part and the consistency

part:

$$\text{color}\, x_i = \sum_{j=1}^{3} f_j \theta_j \tag{24}$$

$$\text{consistent}(x_i) = f_4 \theta_4 \tag{25}$$

Thus,

$$S_\theta(x_i) = \exp\left(\text{color}(x_i) + \text{consistent}(x_i)\right). \tag{26}$$

Define a realization of a game X as consisting of n guesses $(x_1, ..., x_n)$ and n corresponding replies to each guess $(t_1, ..., t_n)$. In this case the game is of length n. Using Bayes' Rule, the probability of a game realization X under strategic parameter θ is

$$P_\theta(X) = P(x_1)P(x_2|x_1, t_1) \cdots P(x_n|x_1 \cdots x_{n-1}, t_1 \cdots t_{n-1}). \tag{27}$$

Substituting the score function into this equation yields

$$P(x_1) = \frac{\exp\left(\text{color}(x_1)\right)}{k_1 e^{\theta_1} + \cdots + k_3 e^{\theta_3} + \bar{k}} \tag{28}$$

where c is the number of colors in guess x_1, k_j is the number of codes in the codeset containing j colors, and $\bar{k}$ is $6^4 - k_1 - k_2 - k_3$, which is the number of codes in the codeset that contain four colors.

Furthermore,

$$P(x_2|x_1, t_1) = \frac{\exp\left(\text{consistent}(x_2)\right)}{m_1 e^{\theta_4} + \bar{m_1}} \tag{29}$$

where m_1 is the number of codes in the codeset that are consistent with response t_1 to guess x_1, and $\bar{m_1}$ is $6^4 - m_1$.

Likewise,

$$P(x_k|x_1 \cdots x_{k-1}, t_1 \cdots t_{k-1}) = \frac{\exp\left(\text{consistent}(x_k)\right)}{m_k e^{\theta_4} + \bar{m_k}} \tag{30}$$

for guess x_k on turn k of the game.

The probability of a game realization X is thus the product of these conditional probabilities. With this simple strategic model in place, the optimal game parameters in this setting can be obtained using gradient descent on the importance sampling function.

4.2.3 Learning the Strategic Parameters

The goal is to select the parameter vector θ such that the number of turns needed to win the game is minimized within our given strategic structure.

As a starting point, a training set of game data was simulated for $n = 1000$ games under parameter $\theta = (0, 0, 0, 5)$. This parameter value represents the strategy of guessing the first code uniformly at random (without regard to

the number of colors in that guess), and then setting a moderate amount of preference on codes that coincide with all past information for each subsequent guess. The simulation resulted in a mean response of 12.571 turns to win, with a standard deviation of $\sigma = 9.25$.

Using the game data collected from the initial simulation, a gradient descent over the importance sampling function was performed. Here, the importance sampling function is the expected game length. The ultimate goal is to determine the strategy that results in the shortest game length on average. In other words, to find the value of θ that minimizes the importance sampling estimate of mean game length.

To implement the learning algorithm and train the parameter vector, the update scheme

$$\theta_{k+1} = \theta_k - g'(\theta_k) \tag{31}$$

was used, where $g'(\cdot)$ is the gradient of the importance sampling function.

After each update step, $n = 1000$ games were simulated under the revised parameter θ, and the procedure was repeated.

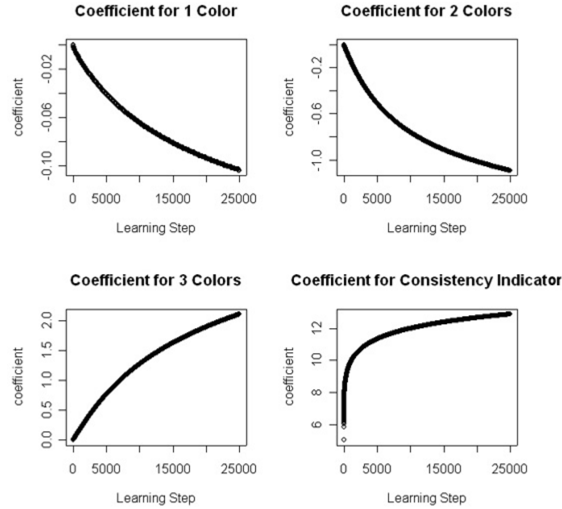

Figure 1: Parameter Values vs. Learning Step

The Trained Strategy

The gradient descent algorithm was allowed to run for 25,000 steps on training sets of $n = 1,000$ simulated games per step, and a fitted parameter estimate of

$$\theta = (-0.488, -1.458, 5.944, 82.371). \tag{32}$$

was obtained. Figure 1 shows the values of the four parameter coordinates throughout the beginning steps of the learning procedure.

From Figure 1, it can be seen that the algorithm quickly determines that the indicator of consistency with past information is crucial in reducing the

number of turns needed to win, as this parameter estimate climbs steadily in the early steps of the learning algorithm. Eventually this curve begins to level-off, as our strategy is overwhelmed by this parameter to the point where selecting an inconsistent guess becomes a negligible event.

As for the features related to color counts, having three colors in the first guess appears to be optimal, as this coefficient climbs steadily. Next best is four colors (fixed parameter value set to 0), and then one color. Having two colors in the first guess appears to be the worst choice, as this parameter drops to the lowest trained value.

Evaluating the Trained Strategy

In order to test the trained model for game strategy, $n = 100,000$ games were simulated under the fitted parameter θ. This strategy required 4.6066 turns on average to complete the game, with a standard deviation of $\sigma = 0.8767$.

In order to evaluate the improvement in the model after implementing the learning algorithm, $n = 50,000$ games were simulated under both the initial parameter values ($\theta_0 = (0, 0, 0, 5)$), and under the fitted parameter values ($\theta = (-0.488, -1.458, 5.944, 0.000, 82.371)$). Histograms of the resulting game lengths in each case are given in Figure 3 and Figure 2.

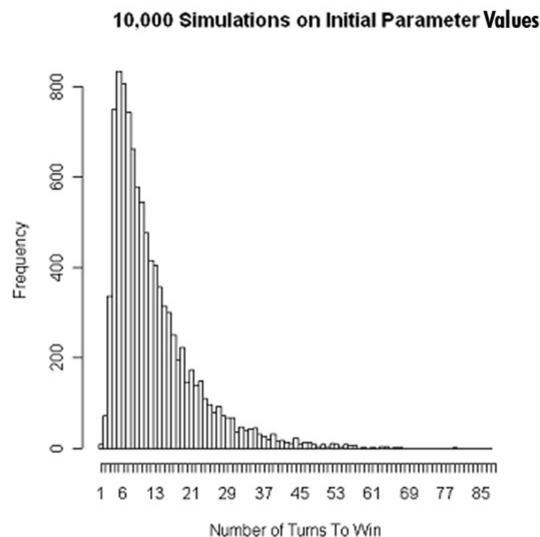

Figure 2: Simulation Results for the Initial Model

Table 4.2.3 gives the summary statistics for each of the two simulations:

It should be clear from the histograms and the summary statistics that the fitted model of strategy does much better than that of the initial model. It wins the game much faster on average, and has greatly reduced variation in game length.

It is interesting to note that the longest game under the final fitted strategy lasted only seven turns, which means that under the standard rules where the decoder has only eight turns to guess the secret code, this strategy would have won the game every time.

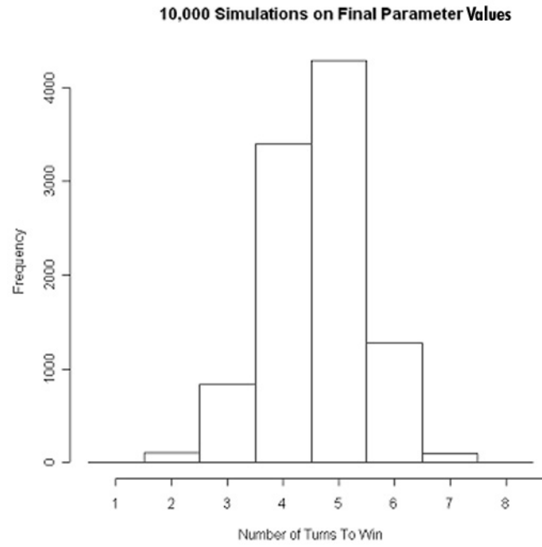

Figure 3: Simulation Results for the Fitted Model (Final)

Table 2: Summary statistics for initial and fitted models

strategy	μ	σ
initial	12.571	9.250
fitted	4.6066	0.8767

After viewing these results, the question arose as to whether the features related to the number of colors in the first guess were truly beneficial in the model, or whether the real strength in the strategy was purely a result of the consistency indicator.

To answer this question, $n = 50,000$ games were simulated under the simple strategy that always selects uniformly from those codes which are consistent with all past information. This strategy required 4.6480 turns on average to complete the game, with a standard deviation of $\sigma = 0.8802$.

Assuming equal variances, a one-sided two-sample t-test for the difference between the two means

$$\mu_1 = \text{mean under fitted strategy} = 4.6183 \quad (33)$$

$$\mu_2 = \text{mean under simple strategy} = 4.6480 \quad (34)$$

yields a p-value of 5.595e-08. Thus, the improvement in the model resulting from the color features is significant, although the reduction in mean number of turns required to win is slight.

Evaluating Importance Sampling Estimates

After the gradient descent algorithm on the importance sampling function was performed, it seemed natural to ask how accurate the importance sampling estimates really were.

To investigate, an initial training data set of size $n = 50,000$ was simulated under initial parameter values ($\theta_0 = (0, 0, 0, 5)$). The gradient descent algorithm was then run for 10,000 steps using only the data from the initial training set. That is, at each step, the parameter values were updated but the data was not re-simulated.

Five different simulations of size $n = 10,000$ were then performed at evenly spaced increments throughout the training procedure, and the mean response in each case was calculated. To clarify, these simulations were run using the learned parameter values at each given stage in the learning procedure. The result is shown in Figure 4. The solid line represents the importance sampling estimates throughout the learning procedure and the X's each annotate the mean game length for the simulated games at the each of the five checkpoints in the learning process.

Thus the importance sampling estimates prove to be quite accurate throughout the procedure, falling only slightly below the estimates obtained via sim-

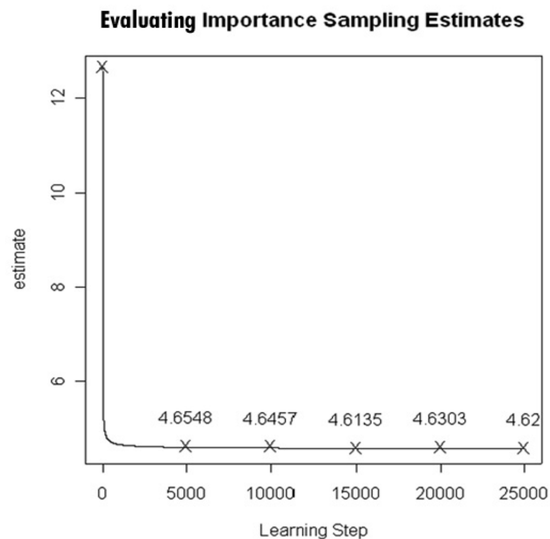

Figure 4: Checking Importance Sampling Estimates

ulation in the later stages of the gradient descent.

4.2.4 Performance

The optimal strategy based on exhaustive search has been shown to require 4.3 moves on average to solve the problem [7]. The fitted strategy here requires 4.618 moves on average to win a game. A t-test shows that this strategy outperforms the simpler model which selects codes uniformly from the set of all consistent codes. These results show that Importance Sampling can be a useful tool in constraint optimization problems.

4.3 CASE 2. More Complex Games

In Case 1, a playing strategy that performed quite well at the standard game was derived. Unfortunately, this strategy is not particularly efficient in terms of computation time. It requires an exhaustive search through each code in the codeset for every turn in the game. Furthermore, in order to implement such a strategy we must keep track of a score for each code in the codeset, then calculate the normalizing constant, and finally sample from the resulting distribution.

For the standard game with six colors and six pegs, there are only $6^4 = 1296$ possible codes, and so these calculations can be performed quite easily on a computer. However, as the number of colors and pegs in the secret code in increased, this exhaustive search method soon becomes unrealistic. Consider the game with ten colors and ten pegs. There are $10^{10} = 10$ billion possible codes in this game. Calculating the normalizing constant in such a game is no simple task.

Thus, it would be nice if we could find a playing strategy that would require neither an exhaustive search through all possible codes, nor the calculation of a normalizing constant.

An Alternative Approach: Metropolis-Hastings

As was discussed in the previous chapter, the Metropolis-Hastings algorithm can be useful in generating a random sample from a specified distribution without the calculation of a normalizing constant. In order to implement the Metropolis-Hastings algorithm, two things must be chosen: a proposal scheme for generating new codes in our random walk, and a corresponding distribution on codes that mixes well (allows movement throughout the codeset).

The usual proposal scheme for this random walk involves generating a new state by manipulating the current state in some simple way. In our case, it seems natural to begin the random walk for each turn in the last state (code) guessed, since in the later stages of the game our previous guesses should be "close" to the secret code in some sense. A natural proposal might be one that manipulates a code by either performing a permutation on the pegs or by switching the color of one or more pegs in the sequence.

Next, in order to be able to use the Metropolis-Hastings algorithm to find the true secret code, a distribution on codes that mixes well and favors *better* codes must be found.

4.3.1 A Cost Function That Promotes Mixing

The most natural way to evaluate a code y during the game would be to consider what the replies would have been if the secret code were in fact y. That is, for each turn i, evaluate the reply to guess x_i under secret code y. Call this the **test response for turn i under secret code y**, or $R_i(y)$. Compare this to the actual reply received on turn i, which is denoted R_i. If $R_i = R_i(y)$ for all turns i, say that y **coincides** with all of the information received and is therefore a candidate for the secret code.

In order to obtain a scoring function on codes that will make the Metropolis-Hastings algorithm move in the right direction (i.e. move toward the secret code), it is necessary to be able to do more than simply evaluate whether a code is still possible or not. A scoring function is required, so that one can evaluate whether a given code that has already been ruled out is better or worse than another, and so that the algorithm will move in the *better* direction and eventually land on a candidate code.

To see this more clearly, consider a scenario where the set of all codes has been narrowed down to a small set – say three – of possible codes. Suppose the random walk starts at our previous guess, which is a code that is no longer possible. We begin the random walk of permutations and color switching in

an attempt to find a candidate code. However, if the only thing known about each code that is proposed is whether or not it is still possible, then there is no inclination of which way to walk if the state proposed is not one of the three candidate codes. The algorithm simply walks about aimlessly until it eventually happens to propose a possible code. This could take a very long time.

Therefore, a scoring function is required that reveals more than simply whether a proposed state is a candidate code. It must imply whether a proposal is getting closer to or farther away from the secret code. And if the intention is to evaluate how well a code y that is already ruled-out coincides with the set of replies received, then it is sensible to examine the absolute difference in the both the number of black pegs and the number of white pegs between R_i and $R_i(y)$ for each turn i. The absolute difference in the number of black pegs is denoted by B_{abs} and the absolute difference in the number of white pegs by W_{abs}. Note that a code that is still possible will have $B_{abs} = 0$ and $W_{abs} = 0$.

To better understand these quantities, consider the following example: Suppose that for your first turn in a game you guessed code $x_1 = (1, 1, 2, 3)$ and received the reply R_1=(B=2,W=0). In selecting your second guess, you

wish to compare your initial state $x_1 = (1,1,2,3)$ with a proposal state $\omega = (1,1,2,4)$. $R_1(x_1) = (B = 4, W = 0)$ since x_1 *was* your first guess, and thus if x_1 were in fact the secret code, you would have received the "win" reply of four black pegs and no white ones. Now, if ω were in fact the secret code, we would have received reply $R_1(w) = (B = 3, W = 0)$.

Clearly both x_1 and w have already been ruled-out as the secret code, but you wish to determine which one is a better guess, so that you can take a step in that direction and move toward the true secret code. For x_1 you have $B_{abs} = |4-2| = 2$ and $W_{abs} = |0-0| = 0$. For ω you have $B_{abs} = |3-2| = 1$ and $W_{abs} = |0-0| = 0$. ω is therefore the better choice, since it has a smaller value for B_{abs} and the same value for W_{abs}. You may think about it this way: x_1 has at least two pegs that are incorrect, and ω has only one that is known to be incorrect. Therefore, ω is the better choice, and you would want the scoring function (and corresponding probability distribution) to favor the proposal state ω over the initial state x_1 in this case. It is a bit more tricky to think about how to choose between two codes when the responses differ in W_{abs} as well as B_{abs}.

Another quantity to consider is the absolute difference between the total number of white and black pegs in a reply, and the total number in the

test response. Call the true difference T and the absolute difference T_{abs}. Intuitively, if a test response for a code y has less total response pegs than the actual reply received, then y is missing some of the correct pegs from the guess x . Alternately, if the test response has more total pegs, then y has too many pegs in common with the previous guess and therefore some pegs are wrong.

For example, in the previous scenario you guessed $x_1 = (1, 1, 2, 3)$ and received reply $R_1 = (B = 2, W = 0)$. The goal is to evaluate the fitness of code $y = (1, 1, 2, 4)$. Here, $R_1(y) = (B = 3, W = 0)$ and thus $T = 3 - 2 = 1$, and $T_{abs} = |T| = |1| = 1$. $T \neq 0$ means that you would be guessing some pegs that you know to be wrong. In this case, $T > 0$, so you know that y contains too many of the pegs from x_1. This makes sense, since y is identical to x_1 in the first three positions: (1,1,2). But the reply to x_1 was $R_1 = (B = 2, W = 0)$, so only two of the pegs in x_1 were in the secret code at all. Therefore if you decide to guess y on your next move, you will be guessing at least one peg that you already know is wrong.

Thus, it would be sensible for a scoring function that evaluated codes to include B_{abs}, W_{abs}, and T_{abs}. The following cost function was incorporated

into the exponential scoring model:

$$\text{Cost}(x) = B_{abs}(x) + W_{abs}(x) + 2T_{abs}(x) \tag{35}$$

It is referred to as cost, because *better* codes receive lower scores.

4.3.2 Metropolis-Hastings Implementation

The Metropolis-Hastings algorithm was implemented using the scoring model,

$$S(x) = \exp\{-\theta \, \text{Cost}(x)\} \tag{36}$$

where $\theta > 0$ is the model parameter that determines how much preference is placed on codes with lower cost. Candidate codes have $Cost = 0$, and therefore receive the highest score of 1.0 and thereby the most preference.

High values of the model parameter θ will force the M-H random walk to accept only those proposal codes with lower or equivalent scores. Selecting values of θ that are too high could create cause the random walk to get stuck in local minimums. Low values of θ will allow the random walk to move about more freely and take some steps in the upward direction. Selecting values of θ that are too low could result in a random walk that moves about aimlessly and never settles into the global minimum. Therefore, an appropriate value

for θ must be chosen that will make the search algorithm move quickly toward the global minimum while still allowing a certain amount of *mixing*.

It should be noted here that we discovered through simulation that the set of candidate codes at any given point in the game is not necessarily *connected*, in the sense that there need not exist a connected path of pair-permutations or single-peg color switches that passes through the entire candidate set.

Evaluating the Proposal Scheme

The proposal scheme implemented would permute two pegs with probability $p = 0.5$, and switch the color of a single peg with probability $1 - p = 0.5$. In order to test if $p = 0.5$ was a good choice for this proposal, game simulations were run under $\theta = 5$ for a spectrum of values of p, and the subsequent mean number of turns required to win was examined. The results are given in Figure 5. The lower curve represents the standard **6/4 game** (6 colors and 4 pegs), the middle represents the 8/5 game, and the top curve the 10/6 game. Each X on the plot represents the mean game length for $n = 100$ game simulations at the corresponding value of the proposal probability p. From this plot, it appears that choosing $p = 0.5$ was indeed reasonable, as the curve appears at a minimum near $p = 0.5$.

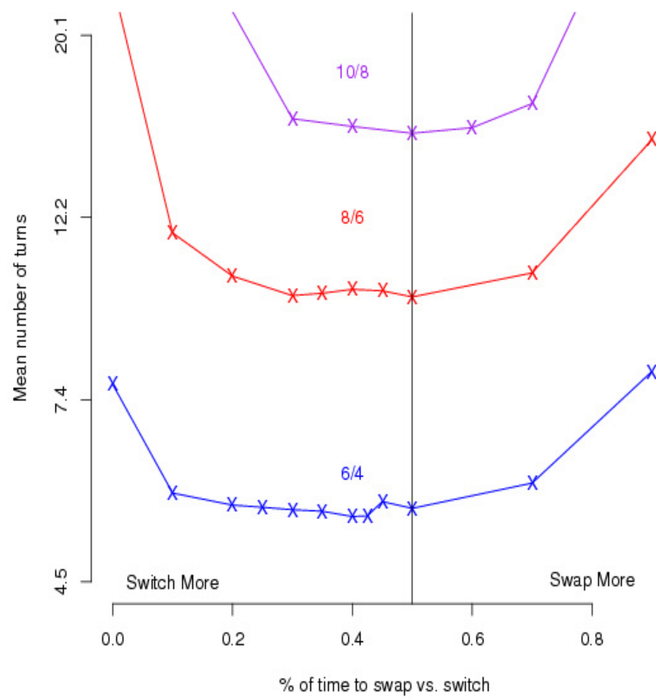

Figure 5: Checking The proposal probability

The next investigation performed was to determine whether including a **subset permutation** (a permutation of more than two pegs at a time) in the proposal function might improve the mixing tendency of the algorithm, and thereby speed up the search for a candidate code. To find out, data was simulated for $n = 10,000$ games under both the 6/4 case and the 10/10 case using $\theta = 5$ with both the original proposal of $\mathbb{P}(\text{pair permutation}) = \mathbb{P}(\text{single color change}) = 0.5$, and the alternate proposal:

$$\mathbb{P}(\text{subset permutation}) = 0.1$$
$$\mathbb{P}(\text{pair permutation}) = 0.45$$
$$\mathbb{P}(\text{single color change}) = 0.45.$$

One-sided two-sample t-tests for the resulting mean game lengths were performed, and the results are given in Table 3. From these data it seems clear that incorporating a subset permutation into the proposal scheme does not provide a reduction in mean game length, and so the original proposal scheme should be reasonable.

4.3.3 Tricks for Efficient Game Play

In the classic Metropolis-Hastings algorithm, we must allow the random walk to proceed for a long time (in theory, to infinity) in order to achieve a proper

Table 3: Testing the inclusion of subset permutation in the proposal scheme

Game Version	With Subset	Without Subset	p-value
6/4	4.8802	4.8679	0.7848
10/10	13.3431	13.3548	0.4038

"random" sample from the desired probability distribution. In practice, it is common to select a fixed number of steps a priori, after which the process will be stopped and the current state selected as the pseudo-random sample.

For this project, the goal was to find a candidate code rather than to obtain a truly random sample from the scoring distribution. Therefore, it is sensible to stop the procedure at any step in which it encounters a candidate code, and select that candidate code for the next guess in the game.

Furthermore, it is reasonable to select the first guess of each game uniformly at random (without any M-H searching), since all codes have an equal scores of 1.0 in the beginning of the game.

Another trick for improving the efficiency of game play is that whenever a reply with the total number of black and white pegs equalling the number of pegs in the codes is received, the proposal scheme changes to simply $\mathbb{P}(\text{Pair Permutation}) = 1.0$. This is sensible because we know that we have all of the correct colors in the last guess, and therefore we would only want

to permute the positions of these colors.

4.3.4 The Problem of Stopping Time

Another concern is how to the reduce of the amount of calculations required to complete the game. It would be possible to simply run the algorithm until a candidate code is discovered for every turn, but this could require a great deal of computation. It could be better in some situations to simply guess a code that we know is close in order to gain more information.

In order to determine good stopping times for the algorithms, $n = 10,000$ simulations of the 5/3, 6/4, 7/5, 8/6, 9/7 games with $\theta = 1$ were performed, where the process was run until a candidate code was discovered on every turn. The "optimal" mean game lengths for each of the versions of the game were tabulated.

Next, $n = 1,000$ simulations for each version of the game were run using stopping times ranging from 100 steps per turn to 2400 steps per turn, in increments of 100 steps. Stopping point for each game version were then selected to be the point at which mean game length slipped below 110% of the optimal game length. The results are shown in Figure 6, and corresponding values are given in Table 4.3.4.

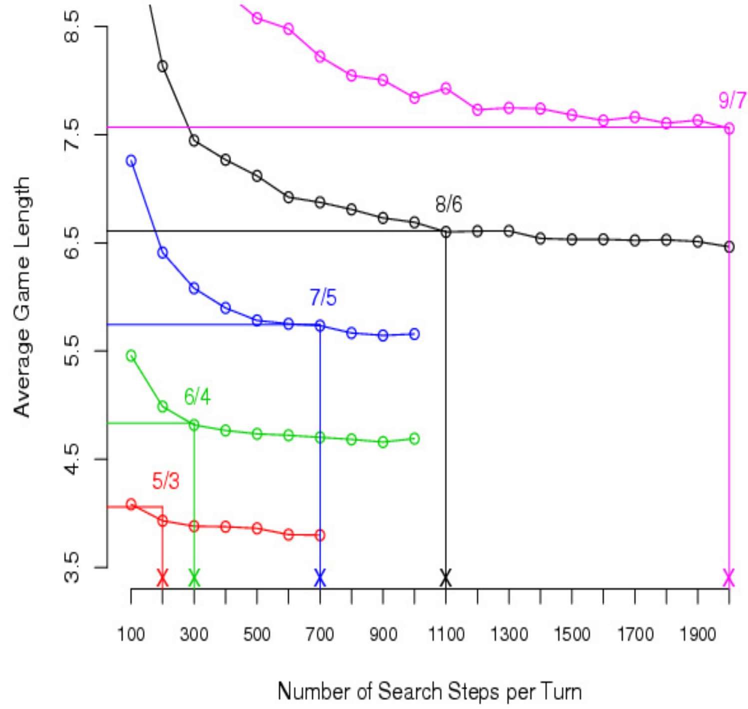

Figure 6: Selecting the Stopping Time

Table 4: Selecting the Stopping Time

Game Version	Optimal Mean Game Length	sd	# Steps to 110% of optimal
5/3	3.866	0.8688	200
6/4	4.645	0.8762	300
7/5	5.472	0.8520	700
8/6	6.297	0.8942	1100
9/7	7.210	0.9332	2000

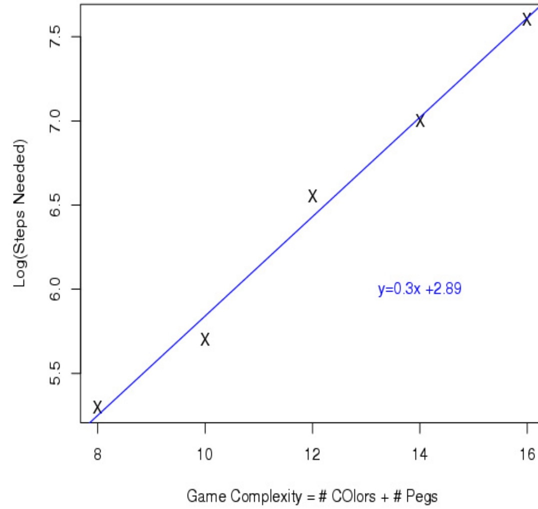

Figure 7: Determining a Formula for the Stopping Time

It is clear from these data that the procedure requires more M-H steps as the number of pegs and colors in the game increases. Figure 7 shows the relationship between the stopping time at which the procedure reaches 110% of the optimal mean game length and the log of the complexity of the game. Game complexity was measured by taking the sum of the number of pegs and the number of colors allowed. The plot suggests that this relationship is linear in nature, and the regression line for these data is given by

$$y = 0.30x + 2.89 \tag{37}$$

Thus, the relationship between game complexity and stopping time can

be described by

$$\# \text{ Steps} = \exp\{0.30(\text{Complexity}) + 2.89\}. \tag{38}$$

The efficiency tricks and this formula for stopping time were incorporated into the algorithm.

4.3.5 The Problem of Sticking

As was mentioned previously, there is a trade-off to consider when selecting the value of the model parameter θ. Small values will result in aimless wandering about, while large values can cause the algorithm to become *stuck* in a local minimum.

The problem of sticking is particularly bothersome in larger versions of the game (e.g. 10/10, or ten colors and ten pegs), where the candidate set becomes more disparate and disjoint. Various means of unsticking the algorithm were explored, and one proved to be both effective and reasonably efficient.

The algorithm was deemed stuck when it was not able to make any improvement in the score for three turns. That is, if the process was unable to make any progress in its search by the end of the stopping time for three consecutive turns in the game.

The algorithm used to unstick the process once it had been deemed stuck consisted of re-starting the search over at a new state (selected uniformly at random), and then feeding it the information from each prior turn in reverse order, allowing for the usual number of search steps between each "turn" as determined by the stopping time formula. In other words, the unsticking process began as though no information had been obtained. It would then proceed as usual, at each "turn" being fed new information after it made the required number of steps. The idea here was that the new search would be unlikely to end up in the same place as the previous one, and would therefore push the algorithm out of the local minimum that it was stuck in.

One drawback to this method of unsticking is that it increases the number of search steps required to play the game whenever it is implemented.

4.3.6 Metropolis-Hastings: A Smarter Search

A great benefit of this algorithm for game play is that it does not require a search through the entire codespace at each turn. Table 4.3.6 shows the average proportion of the codespace searched for $n = 10,000$ simulations of different versions of the game under this algorithm. Here, $|\mathcal{X}|$ is the size of the codespace.

Table 5: Proportion of the Codeset Searched

Game	Game Length (μ, σ)	# Codes Searched (μ, σ)	$\lvert\mathcal{X}\rvert$	% Searched
5/3	(3.918, 0.979)	(288.2 , 99.4)	125	230.6
6/4	(4.812, 1.041)	(604.2 , 236.3)	1,296	46.62
7/5	(5.715, 1.138)	(1234.0 , 533.3)	16,807	7.342
8/6	(6.618, 1.174)	(2437.4 , 1107.5)	262,144	0.9298
9/7	(7.529, 1.494)	(4620.6 , 2875.3)	4,782,969	0.0966
10/8	(8.477, 1.585)	(8586.1 , 7703.4)	100,000,000	0.0000859

Thus, as the complexity of the game is increased, the proportion of the codespace searched is greatly reduced. Figure 8 shows the relationship between game complexity (as the sum of the number of pegs and number of colors in the game) and the log of the percentage of the codespace searched throughout a game.

Figure 9 shows that as the game complexity is increased, the rate of increase for the number of codes searched is slower than the rate for the size of the codespace. The steeper curve gives the number of codes searched, and the more gradual curve represents the size of the codespace. The number of codes is given in the log base ten scale.

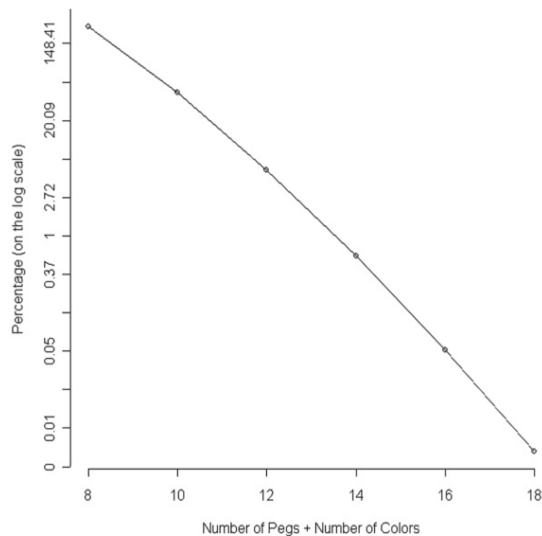

Figure 8: Percentage of the Codespace Searched

4.3.7 Selecting the Model Parameter

In section two of this chapter, importance sampling was implemented to determine the optimal parameter values for a playing strategy. Unfortunately, it does not seem possible to use importance sampling to optimize the model parameter θ in this algorithm. This is due to the fact that the algorithm does not ascertain a true sample from the scoring distribution. It is especially true when the algorithm becomes stuck in a local minimum. If a gradient descent algorithm was run on the importance sampling function in this case, the parameter would train to be larger and larger, applying more preference to

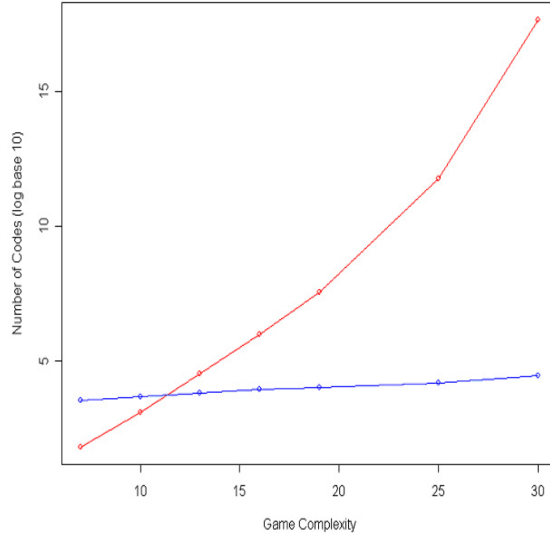

Figure 9: Size of Codespace and Number of Codes Searched

downward motion. But we know that after a certain point, larger values of θ will only increase the potential for the algorithm to become stuck in a local minimum.

So why does the importance sampling procedure train in this way? It is because each time one of the simulated games becomes stuck, it produces a sequence of turns that select higher cost codes (for instance, the same code over and over again). The stuck games last longer, and therefore serve to add evidence in favor of choosing codes with low cost values. There is no way for the gradient descent on the importance sampling function to capture

the fact that increasing θ will only make matters worse.

So what can be done to train the parameter? The optimal value can be estimated by looking at plots of simulation results under a spectrum of parameter values. The curve in Figure 10 represents the mean game lengths for the standard 6/4 game under various values of the model parameter θ. Note that the game take longer for very small values of θ, since when θ is too small there is not enough preference given to codes that have low cost. Also note that the game length climbs steadily as θ gets into the higher values. This is a result of putting too much preference on codes with low cost values, and thereby increasing the tendency to become stuck. It should be clear from the plot that there is indeed an optimal choice for the model parameter. I chose $\theta = 2$ for my final model, and henceforth I will refer to this game play strategy as the **M-H algorithm**.

4.3.8 Performance

A simulation of $n = 10,000$ games under model parameter $\theta = 2$ resulted in a mean game length of $\mu = 4.805$ and standard deviation of $\sigma = 1.130$ turns. The average number of codes searched per game was $\phi = 589.8$ with a standard deviation of 252.8 codes. Table 4.3.8 shows a comparison of these

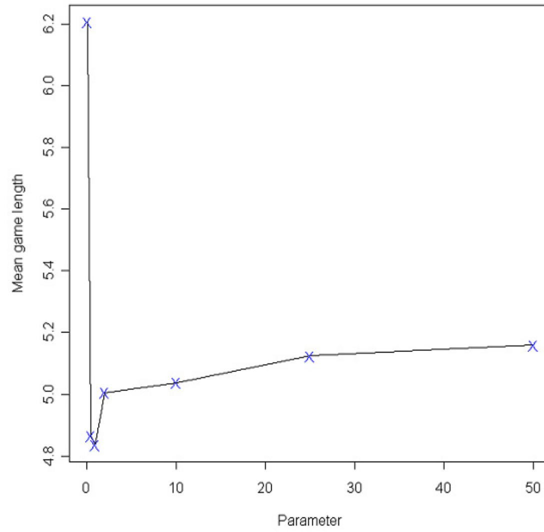

Figure 10: Cost Parameter vs. Mean Game Length

results with those from Case 1. Note that the average number of codes searched under the method from Case 1 would be $\phi = 1,296 * 4.606 = 5969.4$ codes.

Consequently, the model in Case 1 outperforms this model in terms of mean game length, but this model provides a significant reduction in the total number of codes evaluated throughout the game.

Table 6: Comparison of Case 1 and Case 2

Case	Game Length (μ, σ)	Mean # Codes Searched (ϕ)
1	(4.606, 0.8767)	589.8
2	(4.805, 1.130)	5969.4

4.4 CASE 3: False Information

Another question of interest is whether these algorithms can successfully play Mastermind in the presence of false information. That is, will they be able to determine the secret code when each reply has some probability p of having been selected purely at random, and a probability $1-p$ of being a truthful reply. The phrase p%**-false** will refer to the 6/4 game with p% false replies.

In determining the false replies, each time a guess is made it is first compared to the secret code to see if the player has in fact won the game. If the code guessed was not the secret code, then with probability p a reply is selected uniformly at random from the set of all possible replies (excluding the reply of all black pegs, which would falsely suggest that the player had won the game).

Any game strategy capable of playing this p-false game would need to sift through all the information received, then determine which codes were most favored. Under the strategy in Case 1, we were only able to distinguish

between candidate codes and non-candidate codes at any point in the game. With the introduction of false replies, it becomes possible for ALL codes to be non-candidates in the strictest sense, since the true code may not coincide with a randomly selected reply.

However, the strategy proposed in Case 2 is quite capable of playing the p-false version of the game, because it accepts or rejects proposal states based on a comparison between the two states' scores.

Suppose that, at a given point in the game, a false reply is received that forces all codes in the codeset to be deemed non-candidate. Then every code has a positive, non-zero cost, and corresponding score less than 1.0. The M-H process will still favor those codes with lower cost values, which would be those codes that fit more of the information received.

In the long run, since all false replies are generated uniformly at random, the false replies should not concentrate preference on particular codes, and the effect should therefore be outweighed by truthful replies. It is therefore reasonable to expect that the algorithm will seek out the true secret code.

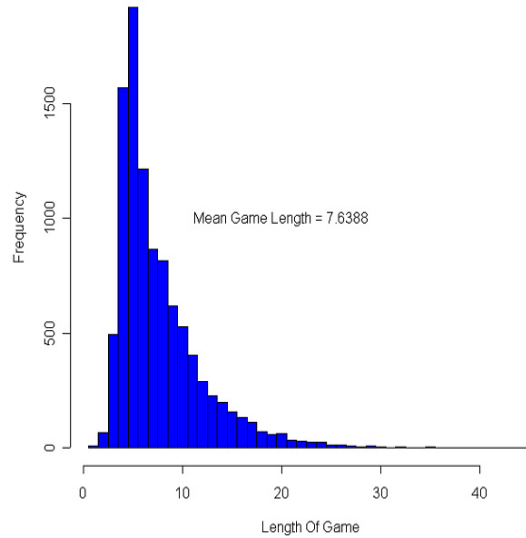

Figure 11: Performance of the M-H algorithm on the 20%-false game

4.4.1 Performance

In fact, the M-H algorithm performs quite well in the presence of false information. For the 20%-false game, $n = 10,000$ simulated games yielded a mean game length of $\mu = 7.6388$ turns with a standard deviation of $\sigma = 4.351$ turns. A histogram of these data is shown in Figure 11.

4.4.2 Humans as Master Mind

It is natural to ask how well humans can perform on the standard 6/4 game and the 20%-false game, and an experiment was performed to find out. There

were ten participants in the study: one professor of Statistics, seven Ph.D. students in Statistics, a stock analyst with a Ph.D. in Mathematics, and a film school graduate.

The subjects were asked to play six games of 6/4 Mastermind. The first three games were without any false information. The last three games were the 20%-false version. Subjects were allowed up to ten turns to win the game. If the subject did not guess the secret code within the first ten guesses, then a score of 11.0 was recorded. Full instructions were given for each version of the game, and the randomization process for the generation of secret codes and false replies was explained in detail. The six secret codes and all false replies were randomly generated prior to the study in order to make results more comparable. That is, the secret codes, the timing of each false reply and the false replies themselves were all predetermined, and identical for all subjects.

All subjects in the study played the game using an online game simulator designed for the study. All game results were recorded automatically and later tabulated.

Subsequently, $n = 1,000$ simulations of each of the six predetermined games were performed using the final M-H algorithm. Games that were not

Table 7: Man vs. Machine

Subject	Standard Game		20%-False Game	
	# Games Won (μ, σ)	# of Turns (μ, σ)	# Games Won (μ, σ)	# of Turns (μ, σ)
Humans	(2.8, 0.421)	(6.77,1.693)	(1.5,1.080)	(9.23,1.735)
M-H Algorithm	(3.0,0.00)	(4.76,1.236)	(2.34,0.585)	(7.59,4.51)

won within ten turns received a score of 11.0. Each sequence of the six games was treated as one unit response.

Summary statistics for the performance of both the human subjects and the M-H algorithm are given in Table 4.4.2. Overall, the M-H algorithm outperformed the human subjects on the standard 6/4 game and the 20%-false game. The best human scores were 4.66 turns on average in the standard game and 6.0 turns on average in the 20%-false game, both of which can be credited to the stock analyst. Only the stock analyst and the Statistics professor won all six games in ten turns or less.

5 Summary

Two algorithms were presented for playing a one player version of Mastermind: A simple model based on an exponential scoring function, and a more complicated algorithm that used the Metropolis-Hastings algorithm to search codes at each turn.

The first strategy was effective, with a mean game length of 4.618 turns. However, this strategy required an exhaustive search through every code at each turn in the game.

The second strategy required 4.805 turns on average to complete the game, but provided a 90% reduction in the number of codes searched per game.

The Metropolis-Hastings algorithm performed well even in the presence of false information, and could outperform humans on both the standard game and the version of the game which included false information.

The success of the first game playing strategy was based on a gradient descent over the importance sampling function on simulated training data. This method of optimization can be a useful tool in machine learning, especially in cases when simulations are expensive to run.

The Metropolis-Hastings algorithm that can be helpful in reducing the

proportion of space searched in function optimization over a large sample space.

References

[1] Bernier, J. Solving mastermind using gas and simulated annealing: A case of dynamic constraint optimization. *Parallel Problem Soving From Nature IV* (1996), 554–563.

[2] Gill, J. *Bayesian Methods: A Social and Behavioral Sciences Approach*. CRC Press, New York, USA, 2002.

[3] Hammersley, J. M., and Handscomb, D. *Monte Carlo Methods*. Wiley, New York, USA, 1964.

[4] Hastings, W. K. Monte carlo sampling methods using markov chains and their applications. *Biometrika 57* (1970), 97–109.

[5] Irving, R. Twoards an optimum mastermind strategy. *Journal of Recreational Mathematics 11* (1978), 81–87.

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

[7] Koyama, K., and Lai, T. An optimal mastermind strategy. *Journal of Recreational Mathematics 25* (1993), 251–256.

[8] Liu, J. S. *Monte Carlo Strategies in Scientific Computing.* Springer-Verlag New York Inc., New York, USA, 2001.