Project 3 - Group 8

Jeffrey Tao, Martin Ristovski, Vikram Rajan

November 15, 2022

Contents

1 Introduction

A Magical Code is the third project of COMS W4444 taught by Professor Kenneth Ross in the Fall 2022 semester at Columbia University. In class, we discussed several different ideas on how to most accurately and robustly encode messages with a deck of cards. Our group consists of 3 members: Jeffrey Tao, Martin Ristovski, and Vikram Rajan. In this report, we will define the specifications of the problem, discuss our initial thoughts and ideas, and then detail the final solution we implemented.

1.1 Problem Specification

The premise of A Magical Code is to encode a secret message you want to send to someone, but only using a standard deck of playing cards. As there are only 52 cards, there is a limit on the length of message we can send, but our job was to see just how long of a message we could send and still send meaningful information. Another limiting factor we faced was the ability of our decks to withstand random shuffling. In the war analogy this problem was presented to us as, this was to circumvent suspicious guards by showing them there could not possibly be a message contained in the deck of cards, as you can randomly shuffle the top cards many times. Given these constraints, we have to be able to encode and decode a message in the deck, but also be able to label random decks as NULL.

The goal is to be able to send as long as a message as possible given the constraints. Another option is to recover a partial message, as this would be more advantageous than recovering gibberish. These partial matches have to be specifically indicated, thus requiring some bits to do so. It also must be an exact prefix match, so it cannot randomly splice together parts of the original message, as that might not mean anything to the receiver.

2 Initial Implementation

The first idea that came to mind was to take our message of characters and encode it into a bitstring. To do this, we got a frequency distribution of the letters of the alphabet in English, and used that to generate a Huffman coding. Now that we had a mapping from characters to bits, we needed a mapping from bits to cards. To do this, we take our bitstring and the number n cards we want to encode it in. Assuming the integer value of the binary string is less than n!, we divide that up into n bins of size (n-1)!. Then we go segment by segment of the bitstring, and choosing the biggest bin number based off the remaining cards, as the cards can not be reused. These cards go to the bottom of the deck, to remain the most resistant to shuffles, and the rest of the cards are placed on top.

The decode function works in exactly the reverse of this, iterating card by card and checking what the corresponding bin number would be, and combining

those to ultimately convert the deck back into the original bitstring. Once we retrieve the intended bitstring from the deck of cards, we break it up into the individual words, as we know the length of the message and bits per word. We convert each of these individual bitstrings into their respective integers, and assuming they are a valid index in our list of words, we translate them.

To make sure that our solution had few false positives, we implemented a checksum generator, which took as input the bitstring of the compressed message, and then we appended the 8-bit output of that checksum to our bitstring.

This allowed us to encode messages that consist of lowercase characters in the English alphabet and reliably decode them for high numbers of shuffles. A plot of our performance in this message domain is given below.

¡TODO¿: INSERT PLOT.

3 Improvements

While our soltuion performed well within this limited message domain, to do well on the tournament there were a number of extensions that we'd need to make.

3.1 Checksum

One of the easier adjustments we made was in order to reduce the rate of checksum collision to further avoid false positives. The purpose of the checksum is to prevent the decoder from deciphering random decks as meaningful messages; however, with the random shuffles, there would be times where the checksum would pass for an invalid message. With an 8-bit checksum, we expected this false positive rate to be 1 out of 256, but the rate we recorded was significantly higher. To accomplish this, we increased the length of our checksum from 8 to 10 bits, which gave us a 4x reduction in the false positive rate.

The variable-length checksum function we implemented took advantage of a cryptographic hashing function called sha256 from Python's hashlib library. This takes in the bitstring, creates a hash as a long hexstring, which we then convert to binary and take the lowest x bits.

3.2 Multiple Domains

Then, we added support for multiple message domains. Implementing multiple domains allowed us to use a more efficient encoding scheme for each domain, which in turn decreased the length of our message and made it more resistant to shuffles. However, we also had to dedicate 3 bits of our bitstring to let the decoder know which scheme was used to encode the message. In our testing, this was a worthwhile tradeoff. We added a number of character-based encoding schemes, which also used Huffman coding but with a different set of charcters and thus a different frequency distribution. In particular, we added a domain containing only numbers, a domain containing charcters typically found

in English writing (uppercase and lowercase letters, numbers, space, and a few punctuation marks), and a domain containing all printable ASCII characters, which we used as a fallback of sorts, since it is able to encode any message at the cost of efficiency.

3.3 Word-based Encoding Schemes

We further extended our multiple domain approach by implementing word-based encoding schemes. These would map words to integers that referrenced each word's position in a dictionary, allowing the decoder to simply look up each word. This meant that if all words in a message are present in our dictionary, we could encode much longer messages than if we were to use a character-based encoding scheme. The dictionaries we initially included were one of a large number of words in the English language, one of names of people, and one of names of places.

¡TODO¿: INSERT PLOTS.

4 Eight Tournament Domains

On November 1, 2022, the class collectively decided to limit the types of messages to ones belonging to 8 domains, one proposed by each group, and each defined by a generator function. Our initial multi-domain apporach allowed us to easily adapt to this part of the tournament specifications, as we were able to build specific encoding schemes for each domain.

The domain we proposed is that of messages of type ___ ¡TODO¿ ___, as we thought it would be a reasonable proposition that could realistically be used by a spy agency.

5 Tournament Performance

6 Limitations