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

2.1 Separating Messages, Bits, and Cards

As discussed in class, we implemented separate schemes for encoding string messages to bits and for encoding bits to cards. This allowed us to work on these parts independently and optimize them separately. As the project wore on, this strategy bore fruit as it enabled us to write many different message-to-bits encoders for the different tournament message domains.

2.2 Message-to-bits: Huffman Coding

Our initial intuition for encoding messages to bits was to exploit the disparity between how frequently letters occur in the English language. To do this, we found a frequency distribution of the letters of the alphabet in English and used that to generate a Huffman coding. Huffman coding works on the principle

that more common characters should have shorter encodings, and less common characters can have longer encodings. Huffman coding is also a *prefix encoding*, meaning that decoding requires no backtracking, as each character has a unique prefix such that decoding is unambiguous. When a character is matched, it is safe to emit that character.

2.3 Bits-to-cards: Card Index Encoding

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 c cards we want to encode it in. Assuming the integer value of the binary string is less than $c!$, we divide that up into c bins of size $(c-1)!$. We proceed bit-by-bit, choosing indexes according to the ordering of the remaining un-used cards, as cards can not be reused. Finally, the message encoded in c cards goes at the bottom of the deck, as cards towards the bottom are more resistant to shuffles. The top of the deck is padded with the remaining $52 - c$ cards in order.

Decoding requires a known value of c (discussed below in section 2.4). Given c, we first filter out all card with values $> c$, as they are irrelevant and may have been mixed into the bottom cards due to shuffles. We recover the input bit string by performing the reverse indexing process: for each place value, we know the bin width, and can find the index of the bin via the index of the current card in the ordering of all currently remaining cards. After recovering the bit string, we can try to decode the message.

2.4 Metadata

We store metadata at the end of our message, where it is most durable. Our initial message metadata consisted of a length byte and a checksum byte. As the project went on, we changed our metadata logic to make these fields variable in length, removed the length byte, and added bits to denote different string-to-bits message encodings.

We do not actually know the value of c used to encode bits in the deck. As such, in order to recover the bit string, we need to try every possible value of c. However, without some protections, we could recover any bit string from any deck, making recovering the original mesasge impossible. As such, we introduced a checksum to the bitstring. The checksum is computed by taking some function over the bits of the message and appending the output bits to the message. As long as the checksum is of a known length, we can recover messages by trying many values of c, taking the last bits as the message checksum, re-computing the checksum function over the remaining bits, and seeing if they match. If they do match, we have high confidence that this is the message that we intended to recover. If they do not match, we are guaranteed that this is not the message we wanted to recover.

Furthermore, due to our bits-to-cards integer encoding scheme, we found that leading 0's would be dropped when decoding the bit string from cards.

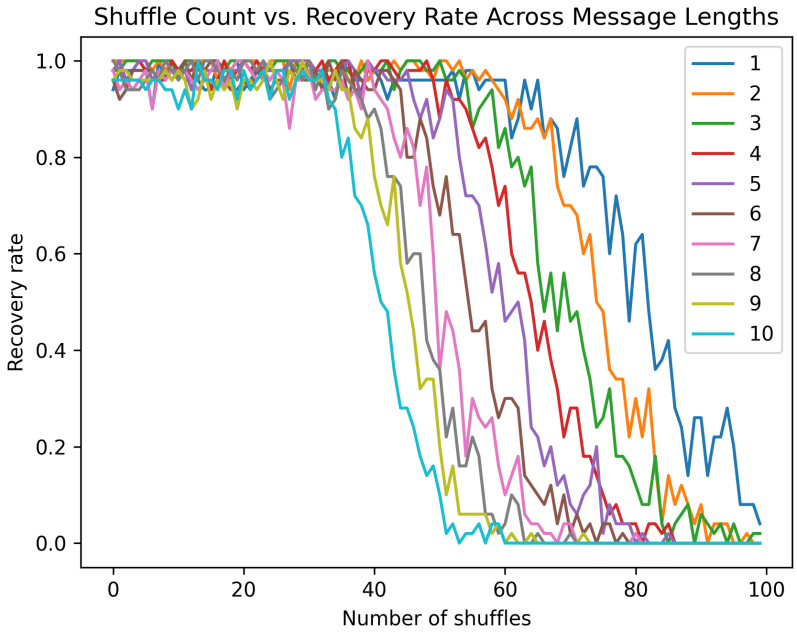


Figure 1: Our initial agent's performance on sequences of lowercase letters drawn uniformly randomly. The vertical axis shows recovery rate, i.e. the proportion of messages that are exactly recovered. Number of shuffles increases along the horizontal axis. Each color encodes the length of the message: longer messages tend to be harder to recover, while shorter messages survive more shuffles.

As such, we elected to add a length byte to the message metadata. A better solution to this problem is discussed in section 3.4.

Our early performance was already quite promising, shown in Figure 1.

3 Improvements

While our solution 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 Better 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. Having a configurable checksum length also proved to be a boon later in the project as it allowed us to experiment freely with checksum lengths to tweak messsage recovery performance.

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 characters 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 Dictionary-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.

3.4 Removing the Length Byte

In our initial scheme, we stored a length byte in the message metadata. For messages with leading 0's (i.e. the message has a prefix consisting of some number of 0's), it is impossible to recover the leading 0's using our bits-to-cards indexed encoding scheme, as the reverse cards-to-bits function returns an integer value. As such, any leading 0's would be lost, so we added message length to

the metadata so that we could manually pad the recovered message with 0's to decode.

Group 3 suggested that we could prepend a 1 bit to the message to preserve leading 0's, and remove the 1 bit before decoding. This allowed us to remove the length byte entirely and save 7 bits of metadata, which improved our agent's performance slightly.

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 consists of messages containing names of people and places separated by spaces, as we thought it would be a reasonable proposition that could realistically be used by a spy agency.

To achieve the best possible message recovery rates given our bit-to-card encoding scheme, we wrote custom message-to-bit encoders for each proposed tournament domain by reverse engineering the corresponding generator code. Below, we detail the message domains and our custom encoders.

We use a few generic techniques, referred to below, to encode messages in each of the following domains. In general, we strip away any constant, predictable syntax (field delimeters, message prefix/suffix, e.g. the leading @ sign in Group 3's password domain) and only encode the variable part.

1. **Dictionary indexing** For message domains that draw from a word list, we load the word list and assign numbers to each word in the list. This allows words to be referred to efficiently by their index instead of encoding the actual letter sequence.

2. **Constant widths** For messages with highly regular structure but variable length, we can break the message into tokens and encode each token in a constant number of bits b. This way, we can decode variable-length messages with no additional information by reading each b bits from the start to the end.

3. **Named known-width fields** For messages with highly regular structure that can be broken into separately encoded parts (henceforth, fields), we can choose a well-known ordering of fields for the domain and encode the message as the concatenation of the fields in the well-known ordering. To decode, we read the number of bits corresponding to the width of each successive field and reassemble the message.

4. **Prefix identification** For messages consisting of concatenations of fixed-width parts, we use the leading bits to identify the token type, which gives the corresponding type length. Thus, we decode the message as a stream,

first reading the type t from the encoding prefix, then the next b_t bits, then the next encoding prefix.

Group 1: Random Characters Group 1's generator produces messages of arbitrary length consisting of lowercase letters, digits, space, and period drawn uniformly randomly. As such, we use **constant width** sequences mapping bits to indexes in an ordered list of all possible characters (6 bits per character). When decoding, we read each 6 bits and output the corresponding character whose index in the domain is given by the corresponding numeric value.

Group 2: Flights Group 2's generator produces messages consisting of 3-letter airport codes, 4-character reservation IDs, and 8-digit dates, representing a particular flight that an agent should fly on. Our agent encodes this via 5 **named known-width fields** fields: airport, reservation, month, day, year.

We have the list of all airport names, so we use **dictionary indexing** to encode them. Reservation codes are length 4 with a known alphabet of uppercase letters and digits, so we use $\lceil log_2(36) \rceil = 6$ bits to encode each character.

We encode month, day, and year separately as the number of possible dates is much smaller than the corresponding range of numbers, i.e. 01312023 is a valid date, but 14999999 is not. The highest expressible date, $\lceil log_2(12282025) \rceil = 24$, whereas $\lceil log_2(12) \rceil + \lceil log_2(28) \rceil + \lceil log_2(4) \rceil = 11$, so we save 13 bits this way.

Group 3: Passwords Group 3's generator produces passwords consisting of digits and words. We do not know what order digits and words appear due to a shuffle operation in the generator: sometimes, two words will be adjacent, sometimes an unknown number of consecutive digits. As such, we use a **prefix identification** encoding, as described above. A 0 prefix denotes a single digit while a 1 prefix denotes an indexed word. Since the number of consecutive digits is unknown, we encode each digit separately to avoid encoding digit sequence length. Words are encoding with **dictionary indexing**. We strip off the initial @ sign and add it back when decoding.

Group 4: Coordinates Group 4's generator produces latitude/longitude pairs. Degree values always have exactly 4 decimal places, so we can store all degree values using 7-digit decimal numbers (when decoding, leading 0's in the integer part are ignored). To store the full coordinate, we use **named known-width fields** for latitude degree, latitude N/S bit, longitude degree, longitude E/W bit. When decoding, we add back formatting (spaces, separating comma).

Group 5: Addresses Group 5's generator produces random addresses. Street names and suffixes are drawn randomly from dictionaries, while address number can be 1 to 4 digits, possibly with leading 0's. To handle leading 0's, we also store the number length, and pad with 0's if necessary. We use **named known-width fields** to store number, number length, street name, street suffix, and trailing space bit. Street name and suffix are encoded using **dictionary**

indexing. We found that Group 5's generator had a bug causing it to add a trailing space to all generated addresses except for the last one, so we store an extra bit to denote the presence of a trailing space.

Group 6: n-grams Group 6's generator produces phrases of n words from a corpus of n-grams derived from recent news articles. As this generator effectively generates "words" from a set of 10 dictionaries, we use **named known-width fields** to store n and a **dictionary index** into the corresponding n-dictionary. We found a slight bug with the n-gram lists in which they sometimes contained $(n-1)$-grams. To compensate, we also search all dictionaries $n' \geq n$ to try to find the corresponding n and index.

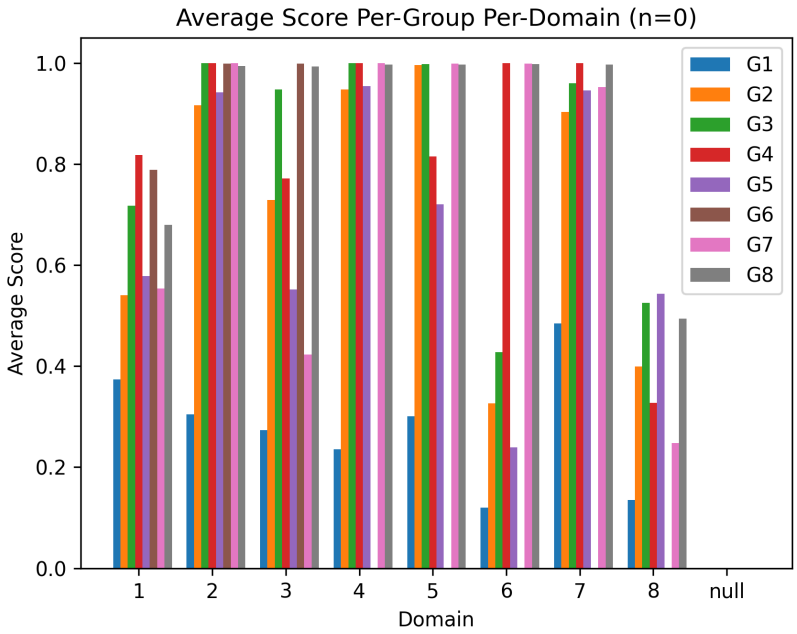
Group 7: Words Group 6's generator produces sequences of space-separated words drawn uniformly randomly from a dictionary. As such, we use **constant width** fields corresponding to **dictionary indexes**. We found that some of the bugged $(n-1)$-grams from Group 6 collided with words in Group 7's dictionary. As such, the encoder for Group 7's generator rejects messages with trailing spaces so that they can instead be encoded by the n-gram encoder.

Group 8: Names & Places As mentioned earlier, we chose to generate messages with words randomly drawn from dictionaries of names and places. Thus, we use a **prefix identification** scheme in which a preceding 0 bit denotes a name with a preceding 1 bit denotes a place. The following bits are extracted as **dictionary indexes**. We realized later that the savings from using this scheme are very marginal: there are 18240 names and 10197 places, which differ by 1 bit in length when encoded. The savings in encoding in this way (vs. combining the dictionaries) would be more substantial for a larger disparity in the dictionary lengths.

5 Tournament Performance

6 Limitations

6.1 Collisions at $n = 0$



The plot above shows recovery rates for $n = 0$, i.e. no shuffles. OUr recovery rate is not perfect for some domains. In these cases, we recover *the wrong message*. This is caused by checksum collisions between encoding domains in our variable-card encoding scheme. Since we do not know c, the number of cards used to encode a message in the deck, we attempt increasing values of c until we find a message that passes the checksum. It is possible that at some incorrect, lower value of c, a different domain's message passes the checksum and is returned. This false positive rate can be lowered by increasing the number of checksum bits, which trades off against message survivability since, as discussed above, longer messages are more likely to be corrupted by fewer shuffles.

6.2 Impact of Partial Decoding

7 Conclusion

7.1 Acknowledgements

We would like to acknowledge Group 3 (Xiaozhou Shi, Rashel Rojas, Noah Silverstein) for the tip to use a single preceding 1 bit instead of a length byte to preserve leading zeros in the message body. This saved 7 bits in our message metadata, which allowed our encoding scheme to survive to higher shuffles than before.

7.2 Summary of Contributions