# 1 A Dictionary-based Approach to Solving a Substitution Cipher

## 1.1 Problem and Importance

Our project focuses on the substitution cipher, which simply substitutes every character in the alphabet with another character or symbol, resulting in a key that is merely a permutation of the alphabet. Simple substitution ciphers are not often used in communication when security is important, but are more commonly solved as a hobby.

Solutions for solving substitution ciphers, however, have applications outside of security. Our proposed algorithm could operate on any number of symbol types, needing only a reasonably small dictionary, and can therefore be used in optical character recognition (OCR) applications. Because this solution works for any set of symbols, it has been used to label scanned text characters after clusters of similar symbols have been defined via unsupervised classification. While many OCR implementations require a large amount of training data, $n$-gram data for the language, and must even take different fonts into account, this algorithm to solve substitution ciphers would only need the information contained in a small dictionary. [1][2]

## 1.2 Related work

Solving cryptographic problems has increasingly become a computing task, and algorithms to decode them have come to use more advanced techniques over time. Ideas like threshold relaxation have been used to increase the likelihood of reaching the absolute maximum probability of the correct cipher. [5] Genetic algorithms have also been used, though they can suffer from a rapid convergence to a suboptimal solution. [6]

Our proposed algorithm is based on the work of Hart, which uses word frequency data to determine likely cipher solutions. [3] We will compare the performance of this algorithm with Hasinoff's Quipster system, which uses a stochastic local search algorithm with a scoring function based on an $n$-gram model of English letters. [4] Although Quipster has a relatively high success rate and does not need a dictionary, it requires the expansion of a large number of nodes in order to perform well.

## 1.3 Approach

### 1.3.1 Building Dictionary

To implement our algorithm for solving simple substitution ciphers, we took several novels available at Gutenberg.org, including *A Tale of Two Cities*, *Pride and Prejudice*, and *Anne of Green Gables*, and parsed them for word content, dropping punctuation and normalizing capitalization. We used the 500 most frequently appearing words from these texts to construct a pattern dictionary. The dictionary was implemented as a hash table with word patterns as keys, and a list of corresponding words as values (for example, the pattern "123445" corresponded to the words "really" and "pretty"). All of the code was written in the Python language (v. 2.7).

Quotes for testing were taken from a wide variety of figures, such as Shakespeare, Aristotle and Charles Darwin, which were listed on Wikiquote.org. This provided us with 94 sample messages that gave a substantial variety of different phrasing, tone and word content from different eras.

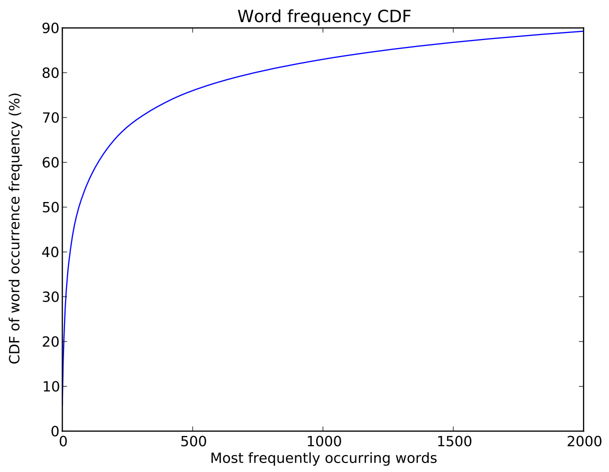

Figure 1: This figure depicts the cumulative density function plot of the 2000 most commonly-occurring words used to construct the dictionary (there were more than 9600 different words in total). This plot shows that half of the word occurrences in the texts came from only 63 words, and that the 500 most commonly used words (the size of our dictionary) accounted for approximately 76% of the word occurrences in the source texts. The fact that so few words account for such a high percentage of English usage is the basis for using this small dictionary-based approach.

### 1.3.2 Algorithm

We implemented the recursive dictionary-based depth first search algorithm presented by Hart. [3] Using a dictionary to drastically narrow the search space cuts the computation time significantly compared to searching through every permutation of the alphabet, and the use of such a small dictionary is enabled by the very compressed nature of word frequencies in English communication (see Fig. 1). For each word in the cipher set (with duplicate words eliminated), the algorithm looked for words in the dictionary that matched the character pattern of the encrypted word, and created a search tree based on these words with the same pattern, with a backup method in case the encrypted word was not in the dictionary.

The algorithm would start with a default key where every letter maps to the special character '*'. For each word in the cipher, the possible decoded words from the dictionary represent a level in the tree, and at each level where a dictionary word pattern matched the given cipher word pattern, the score was incremented by 1, and information from the word was added to the key.

Concretely, in terms of the previous example, if the $n$-th cipher word has a pattern "123445", then the corresponding dictionary words "really" and "pretty" will form the $n$-th level of the search tree, as well as the third possibility that the given word is not in the dictionary. When the algorithm explores the branches for the words "really" and "pretty," the score will be incremented by 1, but when it explores the branch for the third possibility, the score will remain the same, and no information will be added to the key. The branches are explored by recursively calling the function on the next word in the message.

The score was used to keep track of the best solution, and if a solution for a branch would not be able to reach the highest recorded score, then the branch would be pruned. Any branches would also be pruned if they were inconsistent with the key (if one character mapped to multiple characters, for instance). A heuristic was used in ordering the words of the set; words with smaller pattern sets were given priority, as well as words with more characters. This is essentially the MRV heuristic.

To test the algorithm, we had a script go through each quote and generate a random substitution key. The quote would then be encoded with the random key, and the algorithm would be called to solve it. The results for character accuracy, time solved, and best solution would then be stored in a spreadsheet for each quote.

## 1.4 Evaluation

We compared our method to the Quipster algorithm, which randomly swaps letters several thousand times and keeps the key with the best score, which is based on an $n$-gram database. The primary measure we used was the character accuracy of our solutions (the number of correct characters in a message divided by total characters, including duplicates). The overall average character accuracy of our data set was 70.31%, and the average time to solve a cryptogram was 81.48 seconds. The accuracy and time versus character length information can be seen in Figures 2 and 3. Consistent with Hart's results, the maximum time to calculate a solution was rarely over 200 seconds.

As seen in the graphs, the character accuracy was not very highly correlated with the length of the quote. This is very different from the Quipster results, where there is a very evident upward trend in accuracy as the word length increases. The Quipster algorithm approaches 100% character accuracy for quotes with word lengths of about 200

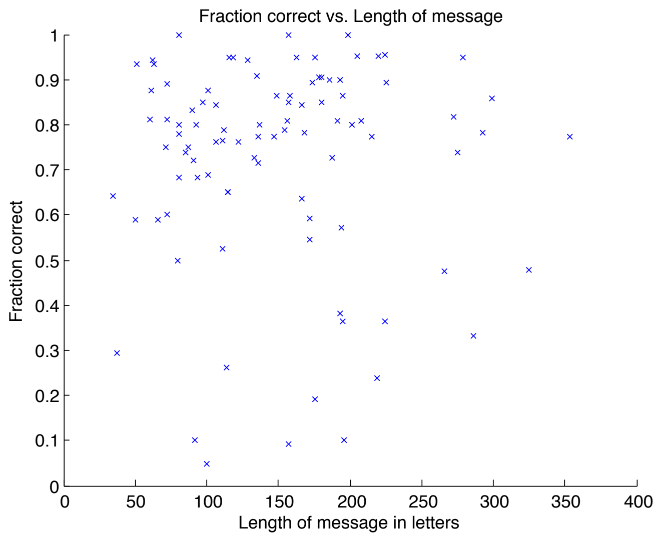

Figure 2: This plot shows character accuracy vs. character length of the quote. There is very little change in accuracy as the character length increases. The average overall accuracy of the quotes was 70.31%.

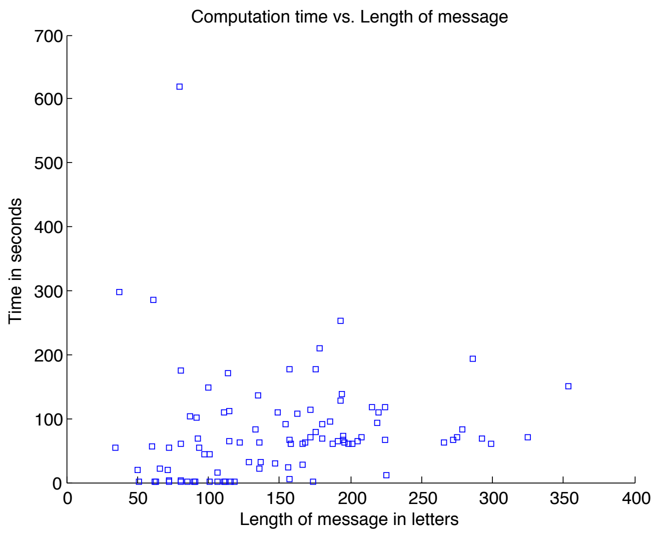

Figure 3: This figure shows time taken to solve the cryptogram vs. length of the quote. This shows a somewhat clearer relationship between the quote length and the time required to solve it. The outliers likely represent examples of short quotes that were lacking a distinctive word that could be found in the dictionary.

characters. Our algorithm never consistently achieved this level of accuracy for any character length.

## 1.5 Discussion

Despite the less than perfect accuracy, most of the solutions could, however, be inferred by a human reader. Take, for example, the following solution from our algorithm with only 80% accuracy: “e*e*ybody has to die, but i always belie*ed an e*ce*tion would be made in my case. now what?” It is fairly obvious to a human reader that the three words missing characters could be solved with a ‘v’, ‘x’ and ‘p.’ Our algorithm could thus be useful when supplemented with a minimal level of user oversight.

The relatively poor and inconsistent performance of our algorithm compared to Quipster is likely due to the small size of our dictionary. While this is beneficial in that it requires less data storage and creates a smaller search space, it makes the algorithm much more prone to suffer from sampling noise. If a message happens to contain few or no words from the dictionary, then the algorithm will search for an excessively long time and will be unable to reach a good solution. The Quipster algorithm, on the other hand, relies on the $n$-gram model of language, and thus benefits from the greater number of $n$-grams in a message, and an $n$-gram database that is much larger than our 500-word dictionary. In terms of time, Quipster also had an advantage in that it used a computationally cheaper algorithm that consisted of simply swapping random characters in the key and applying a scoring function.

Thus, while our algorithm may solve cryptograms well under ideal conditions (and can give a very usable solution in most cases), it is far too dependent on the encrypted words being in the dictionary. The Quipster algorithm is based on a more sophisticated language model, and, with modern computers, can reliably and more quickly produce superior results, despite relying on essentially random guesses. In summary, while the idea behind the small dictionary solver is an interesting one, it is neither the most effective nor efficient method for solving substitution ciphers. It could, however, benefit from improvements that would incorporate methods of the Quipster algorithm, such as the $n$-gram scoring method or some randomized method to better deal with words not in the dictionary.

## References

[1] G. Nagy, S. Seth, and K. Einspahr, “Decoding Substitution Ciphers by Means of Word Matching with Application to OCR,” Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. PAMI-9, pp. 710-715, 1987.

[2] Gary Huang, Andrew McCallum, and Erik Learned-Miller. Cryptogram decoding for optical character recognition. Technical Report 06-45, University of Massachusetts Amherst, June 2006.

[3] G. W. Hart, “To decode short cryptograms,” Commun. ACM, vol. 37, pp. 102-108, 1994.

[4] S. W. Hasinoff, “Solving Substitution Ciphers,” A Technical Report. University of Toronto, 2003.

[5] S. Peleg and A. Rosenfeld, “Breaking substitution ciphers using a relaxation algorithm,” *Commun. ACM*, vol. 22, pp. 598-605, 1979.

[6] R. Spillman, “Solving large knapsack problems with a genetic algorithm,” in *Systems, Man and Cybernetics*, 1995. *Intelligent Systems for the 21st Century., IEEE International Conference on*, 1995, pp. 632-637 vol.1.