An Exact A* Method for Solving Letter Substitution Ciphers

by

Eric Corlett

A research paper submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2011 by Eric Corlett

Abstract

An Exact A* Method for Solving Letter Substitution Ciphers

Eric Corlett

Master of Science

Graduate Department of Computer Science

University of Toronto

2011

Letter-substitution ciphers encode a document from a known or hypothesized language into an unknown writing system or an unknown encoding of a known writing system. The process of solving these ciphers is a problem that can occur in a number of practical applications, such as in the problem of determining the encodings of electronic documents in which the language is known, but the encoding standard is not. It has also been used in OCR applications.

In this paper, we introduce a novel method for deciphering letter-substitution ciphers. We do this by formulating a variant of the Viterbi algorithm for use as an A* heuristic over partial solutions of a given cipher. This heuristic can then be used as a guide in an A* search for the correct solution. We give an overview of the classical Viterbi and A* search algorithms, go on to describe of our proposed algorithm, and prove its correctness.

We then test our algorithm on a selection of ciphers formed from Wikipedia articles, and show that our algorithm has the potential to be a viable, practical method for efficiently solving decipherment problems. We also find, however, that it does have a number of shortcomings, most notably a high variation in running time between similar ciphers. In response to this, we describe potential sources of information to offset this variability and use this information to improve our original algorithm.

We test this improved algorithm on both the original ciphers and a selection of newly collected ciphers and find an average improvement in time and an across-the-board im-

provement in variability. We conclude that we have successfully addressed the issue of high variability found in our original algorithm. The improved algorithm proves to be highly effective in the task of solving letter-substitution decipherment problems.

Contents

1 Introduction **1**

2 Terminology **3**

3 Literature Review **5**

4 The Algorithm **10**

4.1 A Review of the A* Search Algorithm 12

4.2 A Review of the Viterbi Algorithm 15

4.3 A Search over the Partial Solutions 19

5 Experiments, Part 1 **29**

5.1 The Test Sets and Language Model 29

5.2 Measurements 31

5.3 Results, Part 1 31

6 An Improved Algorithm **36**

7 Experiments, Part 2 **43**

7.1 Results, Part 2 43

8 Conclusions **51**

Bibliography **53**

List of Algorithms

4.1 The A* Search Algorithm 14
4.2 The Viterbi Algorithm 18
4.3 A* Section of the Search Algorithm 27
4.4 Specialized Viterbi Algorithm 28
6.1 New A* Section of the Search Algorithm 40
6.2 The New Specialized Viterbi Algorithm 41

List of Figures

4.1 A search for a route from Toronto to Vancouver. 15

4.2 A graph for the cipher *"ifmmp_xpsme"*. 21

4.3 Filling the Greenhouse Table. 25

List of Tables

5.1 Time consumption and accuracy on a sample of 10 6000-character texts. 32

5.2 Time consumption and accuracy on prefixes of a single 13500-character ciphertext. 33

7.1 Time consumption and accuracy of the improved algorithm on the original set of 10 6000-character ciphertexts. 44

7.2 Time consumption and accuracy of the original ("Old") and improved ("**New**") algorithm on prefixes of the original 13500-character ciphertext. 45

7.3 Time consumption and accuracy of the improved algorithm on the held-out set of 10 6000-character ciphertexts. 46

7.4 Time consumption and accuracy of the original ("Old") and improved ("**New**") algorithm on prefixes of a held-out 13500-character ciphertext. 47

Chapter 1

Introduction

Letter-substitution ciphers encode a document from a known language into an unknown writing system or an unknown encoding of a known writing system. By a writing system, we mean a symbolic representation of a human language, and we think of it as being distinct from the language itself. For example, a human language like Hindi can be written in the English alphabet, in the Devanagari script, or encoded electronically using a standard like Unicode. In each of these cases, the underlying language is the same, but the representation, or the writing system, changes.

The problem that we would like to address is that of automatically learning to convert from one writing system to another, even when the relationship between the encodings is not known. More specifically, we will approach this problem with the assumption that the relationship between encodings is one to one: there is a mapping from every character type in one encoding to a unique character type in the other.

This problem has practical significance in a number of areas, such as in reading electronic documents that may use one of many different standards to encode text. While this is not a problem in languages like English and Chinese, which have a small set of well-known standard encodings such as ASCII, Big5, and Unicode, there are languages such as Hindi in which nonstandard encodings are common. In these languages, we

would like to be able to automatically retrieve and display the information in electronic documents which use unknown encodings when we find them. We also want to use these documents in information retrieval and data mining, in which case it is important to be able to read through them automatically, without resorting to a human annotator. The holy grail in this area would be an application to archaeological decipherment, in which the underlying language's identity is only hypothesized, and must be tested.

It should be noted that this problem is cosmetically related to the "L2P" (letter-to-phoneme) mapping problem of text-to-speech synthesis. Both problems feature a prominent constraint-based approach, but the constraints in L2P are very different: two different instances of the same written letter may legitimately map to two different phonemes. This is not the case in letter-substitution maps.

The purpose of this paper, then, is to simplify the problem of reading documents in unknown encodings by presenting a new algorithm to be used in their decipherment. Our algorithm operates by running a search over the n-gram probabilities of possible solutions to the cipher, using a generalization of the Viterbi algorithm that is wrapped in an A* search, which determines at each step which partial solutions to expand. It is guaranteed to converge on the language-model-optimal solution, and does not require restarts or risk falling into local optima. We specifically consider the problem of finding decodings of electronic documents drawn from the Internet, and we test our algorithm on ciphers drawn from randomly selected pages of Wikipedia. Our testing indicates that our algorithm is effective in this domain.

Chapter 2

Terminology

Substitution ciphers are ciphers that are defined by some permutation of a plaintext alphabet. Every character of a plaintext string is consistently mapped to another character in the output string using this permutation. For example, if we took the string *"hello_world"* to be the plaintext, then the string *"ifmmp_xpsme"* would be a cipher that maps e to f, l to m, and so on. It is easy to extend this kind of cipher so that the plaintext alphabet is different from the ciphertext alphabet, but still stands in a one-to-one correspondence to it. Given a ciphertext C, we say that the set of characters used in C is the ciphertext alphabet Σ_C, and that its size is n_C. Similarly, the entire possible plaintext alphabet is Σ_P, and its size is n_P. Since n_C is the number of letters actually used in the cipher, rather than the entire alphabet it is sampled from, we may find that $n_C < n_P$ even when the two alphabets are the same. We refer to the length of the cipher string C as $\text{len}(C)$. In the above example, Σ_P is $\{_, a, \ldots, z\}$ and $n_P = 27$, while $\Sigma_C = \{_, e, f, i, m, p, s, x\}$, $\text{len}(C) = 11$, and $n_C = 8$.

Given the ciphertext C, we say that a *partial solution* of size k is a map $\sigma = \{p_1 : c_1, \ldots, p_k : c_k\}$, where $c_1, \ldots, c_k \in \Sigma_C$ and are distinct, and $p_1, \ldots, p_k \in \Sigma_P$ and are distinct, and where $k \leq n_C$. If for a partial solution σ', we have that $\sigma \subset \sigma'$, then we say that σ' *extends* σ. If the size of σ' is $k+1$ and σ is size k, we say that σ' is an

immediate extension of σ. A *full solution* is a partial solution of size n_C. In the above example, $\sigma_1 = \{_:_, d{:}e\}$ would be a partial solution of size 2, and $\sigma_2 = \{_:_, d{:}e, g{:}m\}$ would be a partial solution of size 3 that immediately extends σ_1. A partial solution $\sigma_T\{_:_, d{:}e, e{:}f, h{:}i, l{:}m, o{:}p, r{:}s, w{:}x\}$ would be both a full solution and the correct one. The full solution σ_T extends σ_1 but not σ_2.

The plaintext for a cipher C can be found by searching over all of the length len(C) strings, which are treated as the outputs of different character mappings from C. A string S that results from such a mapping is *consistent* with a partial solution σ iff, for every $p_i{:}c_i \in \sigma$, the character positions of C that map to p_i are exactly the character positions with c_i in C.

In our above example, we had $C =$ *"ifmmp_xpsme"*, in which case we had len$(C) = 11$. So mappings from C to *"hhhhh_hhhhh"* or *"_hhhhhhhhhh"* would be consistent with a partial solution of size 0, while *"hhhhh_hhhhn"* would be consistent with the size 2 partial solution $\sigma = \{_:_, n{:}e\}$.

In this model, every possible full solution to a cipher C will produce a plaintext string with some associated language model probability, and we will consider the best possible solution to be the one that gives the highest probability. For the sake of concreteness, we will assume here that the language model is a character-level trigram model, and that it is found by counting the letters in a corpus such as the Penn Treebank. When referring to the actual probabilities of plaintext sequences, we will assume the probabilities of unigram, bigram, and trigram sequences are known. That is, for all plaintext letters p_1, p_2, p_3, we assume the probabilities $P(p_1)$, $P(p_2|p_1)$, and $P(p_3|p_1p_2)$ are known.

Chapter 3

Literature Review

It may seem at first that automatically decoding (as opposed to deciphering) a document is a simple matter, since a manual approach has been known for a long time. That is, problems of this sort can be manually solved by recording the unigram frequencies of the characters in the cipher and comparing them to the frequencies given in a language model. However, studies have shown that simple algorithms of this sort do not always produce optimal solutions (Bauer, 2007). The reason for this is that the unigram frequencies taken from the cipher message often are different from those of the language model. If the text from which a language model is trained is taken from a different genre than the plaintext of a cipher, for example, then frequency counts may be misleading. People who solve ciphers manually using unigram frequency counting can succeed because they have access to detailed linguistic knowledge that can be used to compensate for differences in frequency. The perceived simplicity of the problem, however, has meant that efforts to solve it have not been particularly common.

The first major effort to understand its computational properties was presented by Peleg and Rosenfeld (1979). In this paper, the solution to a cipher is found by using a graph relaxation method. That is, the algorithm operates by creating a hypergraph for the cipher (a hypergraph is a graph in which the edges may connect more than two

vertexes). The nodes of the hypergraph are the ciphertext letters, and edges are the 3-tuples of the trigrams in the cipher. Each node representing a ciphertext letter is given a probability distribution determining which plaintext letters it can map to, and these probability distributions are tightened to a full solution through an iterative Bayesian method that relates the probability distribution on the nodes to the trigram probabilities in the cipher. Unfortunately, no guarantees of convergence are given for this method.

Since then, several different approaches to this problem have been suggested. Hart (1994) gives an approach that uses word count frequencies to approximate the solution. This is done by assigning a score of 10^{-2} to the 135 most commonly occurring words in English, and a score of 10^{-6} to all other words. Solutions to the cipher are grown by attempting to map the ciphertext words to the high-probability words, subject to the constraint that every cipher symbol type maps to a unique plaintext symbol type. A solution is considered optimal if it maximizes the number of high-probability words in the plain text. This approach can have a large search space, and the presence of out-of-vocabulary words can limit the applicability of pruning. Moreover, the ciphertext must be broken into words in order for it to be used in this method. Clearly, if the encoding of the language does not make word endings clear, this approach will fail.

Another more recent decipherment algorithm is given by Olson (2007). This approach also starts by breaking the cipher up into words, although it is admitted in the paper that the locations of the word endings may have to be guessed. The algorithm works by performing a depth-first search over the words in the cipher. A ciphertext word is picked at random, and it is matched to all dictionary words that could be the related plaintext. In the example given in Chapter 2, the ciphertext *"ifmmp"* might map to *"hello"*, but it could also map to *"broom"*. Every such possibility becomes a different branch of the search tree. Every possible mapping fixes a number of cipher letters (here, *i, f, m* and *p*), and these constraints are sent to the remaining words. Another word is then chosen at random and the process is repeated, with the extra constraints added. Not every word

in the cipher is expected to be in the vocabulary. This approach, like the one described by Hart (1994), has the limitation that the word endings in the cipher must be explicitly known in order for the algorithm to succeed. Moreover, the presence of out-of-vocabulary words can have a strong influence on the efficiency of the search. If the first word chosen is not in the dictionary, for example, any attempt to match it to a dictionary word will lead to a dead end. The search will spend much of its time going through these dead ends before it starts down the correct search branch.

A more character-driven approach is discussed by Jakobsen (1995), who gives an algorithm that attempts to find an optimal solution to a cipher in terms of its trigram frequencies. It starts by guessing at a solution and determining its trigram probability. The solution is then iteratively improved by swapping letters in the mapping until a locally maximum probability is reached. The solution at this point is returned as the optimum. It can be seen that this algorithm is capable of getting trapped in local maxima, will require random restarts, and even then cannot guarantee a globally optimal solution.

Genetic programming methods have also been applied to this problem. In Gester (2003), a number of different methods for combining full solutions to a cipher are experimentally tested, and are found to be ineffective. The reason for this likely stems from the fact that a search space made up of only the full solutions to a cipher does not readily admit a structure that is smooth enough to ensure that a single global minimum can be found through descent methods — as in Jakobsen's algorithm, there are many local maxima and minima for the problem. Moreover, it is difficult to determine a method for combining two "good" solutions (i.e., two solutions that have many ciphertext letters guessed correctly) into a third solution in such a way that the good letter choices are preserved.

There has been a series of papers that have treated this task as an Expectation and Maximization problem (Knight et al., 2006; Knight, 1999). That is, the algorithms proposed by these papers do not treat the solutions of ciphers as hard assignments, but

as probability distributions over the different plaintext assignments for each ciphertext symbol type. An initial uniform distribution is assumed, and is iteratively updated in expectation and maximization phases. In the expectation phase, the sum of the probabilities that different plaintext solutions could generate the cipher is calculated using a forward and backward Viterbi pass. In the maximization step, the posterior of solution probabilities is found using the plaintext probabilities from the expectation phase. Should the algorithm converge, the probabilistic solution is taken as the optimal one, and the quality of the solution is taken as the number of errors induced in the resulting plaintext (as opposed to the number of incorrectly assigned ciphertext symbol types). Unfortunately, Knight's algorithms are highly dependent on their initial states, and require a number of restarts in order to find the globally optimal solution.

A further contribution was made by Ravi and Knight (2008), who treat the decipherment of letter-substitution ciphers as an integer programming problem. Clever though this constraint-based encoding is, their paper does not quantify the massive running times required to decode even very short documents with this sort of approach. Such inefficiency indicates that integer programming may simply be the wrong tool for the job, possibly because language model probabilities computed from empirical data are not smoothly distributed enough over the space in which a cutting-plane method would attempt to compute a linear relaxation of this problem. Ravi and Knight (2008) also seem to believe that short cipher texts are somehow inherently more difficult to solve than long cipher texts. This difference in difficulty, while real, is not inherent, but rather an artifact of the character-level n-gram language models that they (and we) use, in which preponderant evidence of differences in short character sequences is necessary for the model to clearly favour one letter-substitution mapping over another. Uniform character models equivocate regardless of the length of the cipher, and sharp character models with many zeros can quickly converge even on short ciphers of only a few characters.

Applications of decipherment are also explored by Nagy et al. (1987), who use it in

the context of optical character recognition (OCR), as well as by Snyder et al. (2010), who use it in the context of lost language decipherment.

Chapter 4

The Algorithm

We have approached the problem of solving monoalphabetic ciphers with the transliteration of electronic documents in mind. Specifically, we have been interested in the transliteration of websites that use unknown encodings, and we have constructed our algorithm around this application. Naturally, it would be helpful to be able to use a known encoding for a document if one exists, without having to resort to the longer process of decipherment. We must therefore determine if a document with no known encoding specified actually uses an unknown encoding, or simply mislabels a known one. This is not problematic, however, since the number of known encodings is generally fairly small, and so the possible known encodings can be individually tested. We can reasonably assume that the trigram probability for a document will be very low if the encoding is not correct, and so we can determine if a known encoding for a document is likely by comparing the probability of the document under that encoding with the probability of a random text of a similar length from the target language. The problem description is as follows:

Problem: We are given a website that we have split up into characters to give us a ciphertext C. It is assumed that the language of the text from which the cipher is derived is known, but the encoding is unknown. Given an appropriate language model, we would like to infer the correct encoding. We will assume this to produce the maximum trigram probability given the document and language model.

According to our algorithm, the best possible solution for the ciphertext C is found by conducting a search over its partial solutions. The search is conducted over an implicit directed graph structure whose nodes are a subset of the partial solutions. For any two partial solutions σ and σ' in the graph, the graph has the edge (σ, σ') if and only if σ' is an immediate extension of σ. The root node of the graph is searched first, and at every step, the unsearched neighbors of the searched nodes are listed. For all of these unsearched partial solutions, an estimate is kept of the best probability of any full solution extending it. This estimate is used to order the partial solutions, and the most promising one, as guessed from the estimates, is then searched. We will find that our search is a valid A* search, and so it is both complete and correct over the full solutions. With this in mind, we will give an overview of the A* algorithm in Section 4.1. Our estimate for the probability of the best solution extending a given partial solution will be found using a specialization of the Viterbi algorithm, and so we will follow our discussion of the A* search with an overview of the Viterbi algorithm in Section 4.2. Finally, the details of how we modify these algorithms in order to search the partial solutions will be given in Section 4.3.

4.1 A Review of the A* Search Algorithm

The A* search, which was introduced by Hart et al. (1968), is a search algorithm for graphs. Dechter and Pearl (1985) show that this search algorithm allows us, given a directed graph with distinguished source and target nodes and an edge cost function, to find the least-cost path from the source to the target. In this regard, the A* search algorithm is similar in nature to the Breadth-First Search algorithm, but it allows the search order of the nodes to be modified according to some guess, or heuristic, as to the remaining cost from any node to the sink. The specific problem definition is as follows.

Problem: We are given a (directed) graph $\Gamma = (V, E)$ and two distinguished nodes $s, t \in V$. For every edge $e \in E$ we are given a nonnegative cost $c(e)$, and we are given a nonnegative function $h : V \to \mathbb{R}_{\geq 0}$ such that for any vertexes $u, v \in V$:

- $h(u)$ never overestimates the cost of the best path from u to t.
- $h(u)$ is monotonic. That is, if uv is an edge in E, then $h(u) \leq c(uv) + h(v)$.

The function h is called a *consistent heuristic* if it satisfies these two criteria. We would like to find a path P from s to t such that the sum $\sum_{v \in P} c(v)$ over the edges v in P is minimal. If no path from s to t exists, the algorithm should return *"Solution Infeasible"*.

If we were to run a Breadth-First Search on the graph Γ, we would iteratively calculate the value $g(v)$ for every node v reachable from s, where $g(v)$ is the value of the lowest-cost path from s to v. That is, if P is the set of edges on this lowest cost path, then $g(v) = \sum_{p \in P} c(p)$. The A* algorithm works in a similar manner, except the value $h(v)$ is included in the calculation. Whereas the Breadth-First Search algorithm orders the nodes by $g(v)$, the A* search algorithm orders the nodes by the value of the function $f(v) = g(v) + h(v)$. Thus, $f(v)$ gives the cost of the best path to v plus an estimate of the remaining cost for the best path from v to t. The exact form of the search is given

in Algorithm 4.1. It has been shown by Dechter and Pearl (1985) that so long as the heuristic h is consistent, the A* search is both optimal and complete.

As in the Breadth-First Search algorithm, we keep track of both the nodes that have been searched and the neighbors of nodes that have been searched when running the A* search. Because we use a priority queue to order the nodes that have yet to be searched, we store the unsearched neighbors of the searched nodes in the set "ENQUEUED", and because we expand the list of a node's neighbors when we search it, we store the searched nodes in the set "EXPANDED".

To show why the A* search can be more useful than a Breadth-First Search, consider this example: We have a map showing several cities and the roads between them. We might be in one city on the map (say, Toronto), and desire to travel by road to another (say, Vancouver). In this case, the graph Γ could be similar to that shown in Figure 4.1. Its nodes would be the cities, and its edges would be the roads. Now, we can find the shortest path from Toronto to Vancouver in this graph using the Breadth-First Search algorithm, but the search itself may not be as efficient as we would like. For example, when expanding the immediate neighbors of Toronto in the graph, we may find them to be Montreal and Winnipeg. If we use the Breadth-First Search, we may explore Montreal first, even though it is clear that Winnipeg is closer to the target, and therefore likely to give a better overall path. On the other hand, it can be shown that the distance to the target city is an admissible heuristic. If we use this heuristic as described in the A* algorithm, we would immediately choose to search Winnipeg as opposed to Montreal, and would likely manage to avoid altogether the burden of searching the longer paths to Vancouver that go through Montreal.

Finally, we would like to draw attention to the fact that the efficiency of the A* algorithm can vary greatly according to the heuristic used. It can be seen that the trivial heuristic ($h(v) = 0$ for all $v \in V$) is admissible, and if used, will mimic a normal Breadth-First Search algorithm. If the heuristic can somehow tell us the actual cost of the best

Algorithm 4.1 The A* Search Algorithm

Input: A (directed) graph $\mathit{\Gamma} = (V, E)$, a pair of distinguished nodes s and t, a nonnegative edge cost function $c : E \to \mathbb{R}_{\geq 0}$, and an admissible heuristic $h : V \to \mathbb{R}_{\geq 0}$.

The function call is of the form: A*Search$(\mathit{\Gamma}, s, t, c, h)$.

Output: An optimal path from s to t, and its cost.

1: Set EXPANDED $= \varnothing$.
2: Set ENQUEUED $= \{s\}$.
3: **for** $v \in V$ **do**
4: Set $g(v) = \infty$.
5: Set $\pi(v) = \varnothing$
6: Set $g(s) = 0$.
7: **while** ENQUEUED $\neq \varnothing$ **do**
8: Find $v \in$ ENQUEUED such that $f(v) = g(v) + h(v)$ is minimal.
9: Set ENQUEUED = ENQUEUED $\setminus\{v\}$.
10: Set EXPANDED = EXPANDED $\cup\{v\}$.
11: **if** $v = t$ **then**
12: Return $g(t)$, $P = \{t,\ \pi(t),\ \pi(\pi(t)),\ \ldots,\ s\}$.
13: **else**
14: **for all** $w \in V$ such that $vw \in E$ and $w \notin$ EXPANDED **do**
15: **if** $g(w) > g(v) + c(vw)$ **then**
16: Set $g(w) = g(v) + c(vw)$.
17: Set $\pi(w) = v$.
18: Return "No Solution" .

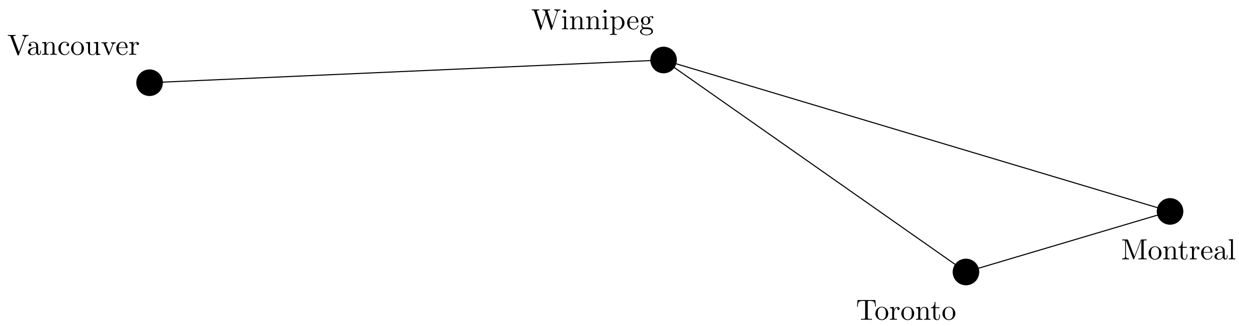


Figure 4.1: When searching for a route from Toronto to Vancouver, it is helpful to search Winnipeg before Montreal, even though Montreal is closer.

path from any node v to t, then the A* algorithm will search only the nodes that are actually on a best path from s to t. In a similar sense, a heuristic that is close to the actual best cost will generally be more efficient than one that is not, and so tightening a heuristic to make it closer to the actual best cost can improve the overall performance of a search. This will be relevant in Chapter 6 when we show how our algorithm is improved.

4.2 A Review of the Viterbi Algorithm

The Viterbi algorithm, which was introduced by Viterbi (1967), is designed to allow a source signal, and its probability to be efficiently inferred after it has passed through a noisy channel. That is, we assume a source signal to be represented as a string generated by a known probability distribution, and that this signal has passed through a channel that changes its characters according to another known probability distribution. If we know the signal after it has passed through this channel, then the Viterbi algorithm allows us to find the most probable signal to have passed into it, and its associated probability. The algorithm is generally run using bigrams to model the source signal, but since our specialization uses trigrams instead, the version we present here will also model the source signal with trigrams. Moreover, we never use the actual source signal in our work, so we only given the sections of the algorithm that give the associated probability.

Problem: We are given a source alphabet Σ_S. A string $S \in \Sigma_S^*$, which represents a source signal, has been generated by a given trigram probability model over Σ_S. That is, for all characters s_1, s_2, $s_3 \in \Sigma_S$, the unigram probability $P(s_1)$, the bigram probability $P(s_2|s_1)$, and the trigram probability $P(s_3|s_1 s_2)$ are known. The string probability of the whole string S is denoted by $P(S)$. We are also given an output probability distribution: every $s \in \Sigma_S$ will be read as a letter c in an alphabet Σ_C. This letter c is not fixed, but is chosen at random for every instance of s in S according to a probability distribution $\beta(c|s)$. Every letter in the string S has been transformed in this manner, resulting in a string $C \in \Sigma_C^*$. We assume that the output transformations are independent, and so we will write $\beta(c_0 c_1 \dots c_n | s_0 s_1 \dots s_n) = \beta(c_0|s_0)\beta(c_1|s_1) \dots \beta(c_n|s_n)$.

Given that we know the output string C and the probability distributions P and β, we would like to infer the probability $P(S) \times \beta(S)$ of the most likely source string S.

Clearly, the brute force method of finding this probability will require far too much time for it to be feasible for all but the shortest strings. This problem can be solved, however, with dynamic programming through the use of the Viterbi algorithm.

The idea behind this algorithm is that we find the probability of the most likely string S by filling in two tables G and B whose columns correspond to the indices of the string C and whose rows are indexed by the source alphabet Σ_S. For every index (x, y), we consider all strings of length $x+1$ in Σ_S that end in y. The probability of the string that gives the highest probability of producing $C[0]C[1] \dots C[x-1]$ will be stored in the cell at $G(x, y)$. For the sake of our calculations, we will also make use of a backpointer in our table: the identity of the second-last letter in the above most likely string will be stored in $B(x, y)$. We note that these back pointers are actually needed for the calculations, and are not used to reconstruct the string S.

Filling in these tables is a very straightforward matter. The first column can be filled in by recording the unigram frequency of each source letter times its output probability, since there is only one string to consider. The backpointers in this column are all null.

To fill in the second column, the bigram frequencies are considered. For the index $(1, s)$, we look at every possible bigram $s's$. We find the s' such that $G(0, s') \times P(s|s') \times \beta(C[1]|s)$ is maximized, and we put the resulting probability into $G(1, s)$. We set $B(1, s) = s'$.

The third column is filled in a similar manner, but using trigrams. At the index $(2, s)$, we consider every possible trigram $s_1 s_2 s$. We find the s_1, s_2 such that the probability $G(0, s_1) \times P(s_2|s_1) \times \beta(C[1]|s_2) \times P(s|s_1 s_2) \times \beta(C[2]|s)$ is maximized. We place the resulting probability into $G(2, s)$, and s_2 into $B(2, s)$. Note that we do not yet make use of the value at $G(1, s_2)$. This is because the optimal value for s_1 might not be equal to the letter at $B(1, s_2)$. If we tried to use the value of $G(1, s_2)$ in this location, however, s_1 would have to be fixed given s_2 for the calculation to be correct.

Finally, every column after the third can be filled using the previous two columns. At the index (i, s), $i \geq 3$, we consider every possible trigram $s_1 s_2 s$. We find the s_1, s_2 such that the probability $G(i-2, s_1) \times P(s_2|B(i-2, s_1)s_1) \times \beta(C[1]|s_2) \times P(s|s_1 s_2) \times \beta(C[2]|s)$ is maximized. We place the resulting probability into $G(i, s)$, and s_2 into $B(i, s)$. This is the point at which the backpointers are really necessary. If we were to use bigrams in the calculation in the way that we do in column three, the probability that we would store in the cell $G(i, s)$ would no longer be an accurate trigram probability.

Finally, we need to find the probability $P(S)\beta(S)$ of the most likely string S. Once we are finished filling the table, we simply find the highest probability in the last column.

The Viterbi algorithm using trigrams is given in Algorithm 4.2. This algorithm is in some ways very close to what we need in order to solve our ciphers, since we can treat the encipherment process as a transformation of a source string in which the output probability $\beta(c|s)$ is 1 if s is the plaintext of c, and 0 otherwise. Unfortunately, the distribution β is unknown, and so we will have modify the algorithm for it to work.

Algorithm 4.2 The Viterbi Algorithm

Input: The string C.

The function call is of the form: $\text{Vit}(C)$.

Output: Probability $P(S)\beta(S)$ of the most likely string S to have generated C.

1: Create two $\text{len}(C) \times |\Sigma_S|$ sized tables G and B.

2: Initialize G, B to 0.

3: **for** $s \in \Sigma_S$ **do**

4: $G(0, s) = P(s) \times \beta(C[0]|s)$.

5: **for** $s \in \Sigma_S$ **do**

6: $G(1, s) = \max_{s' \in \Sigma_S}(G(0, s') \times P(s|s') \times \beta(C[1]|s))$.

7: $B(1, s) = \text{argmax}_{s' \in \Sigma_S}(G(0, s') \times P(s|s') \times \beta(C[1]|s))$.

8: **for** $s \in \Sigma_S$ **do**

9: $G(2, s) = \max_{s_1, s_2 \in \Sigma_S}(G(0, s_1) \times P(s_2|s_1) \times \beta(C[1]|s_2) \times P(s|s_1 s_2) \times \beta(C[2]|s))$.

10: $B(2, s) = \text{argmax}_{s_1, s_2 \in \Sigma_S}(G(0, s_1) \times P(s_2|s_1) \times \beta(C[1]|s_2) \times P(s|s_1 s_2) \times \beta(C[2]|s))$.

11: **for** $i = 3$ to $\text{len}(C) - 1$ **do**

12: **for** $s \in \Sigma_S$ **do**

13: $G(i, s) = \max_{s_1, s_2 \in \Sigma_S}(G(i-2, s_1) \times P(s_2|B(i-2, s_1)s_1) \times \beta(C[1]|s_2) \times P(s|s_1 s_2) \times \beta(C[2]|s))$.

14: $B(i, s) = \text{argmax}_{s_1, s_2 \in \Sigma_S}(G(i-2, s_1) \times P(s_2|B(i-2, s_1)s_1) \times \beta(C[1]|s_2) \times P(s|s_1 s_2) \times \beta(C[2]|s))$.

15: Return $\max_{s \in \Sigma_S} G(\text{len}(C) - 1, s)$.

4.3 A Search over the Partial Solutions

Having reviewed the A* search and Viterbi algorithms, we can return to the problem of efficiently solving letter substitution ciphers. We assume that we are given a cipher C and a language model. We conduct our search by running an A* search over a subset of the partial solutions of C. The subset of the partial solutions that we use is determined as follows: We fix a total order for the different letters in the ciphertext alphabet Σ_C. We will consider the solutions that fix these letters in order. Formally, a partial solution σ of size n will be in the search graph iff the image of σ is precisely the first n letters of Σ_C according to this order. For example, if the order of Σ_C were $_ \leq a \leq g \leq y \leq \ldots \leq b$, then $\sigma = \{_:_, d:a, e:g\}$ would be on the search graph, since the image of σ in Σ_C (i.e., $\{_, a, g\}$) is exactly the set of the first three letters in the order. On the other hand, $\sigma' = \{_:_, d:r\}$ would not be on the search graph, since the image of σ', i.e., $\{_, r\}$, is not the set of the first two letters in the order.

We create a graph out of this subset of vertexes by placing, for any partial solutions σ, σ', an edge from σ to σ' iff σ' is an immediate extension of σ. Each of these edges is given a cost of 0. We add to the graph a target node t, and run an edge from every full solution to it. If σ is a full solution, the cost of the edge σt is the negative log probability of the plain text of that solution given the language model. In the most general case, we take the source node s to be the empty solution ($\{\}$), but if we have previous knowledge of the system we can "move up" the starting point of the search to the best partial solution known. For example, in Chapter 5 we will find that spaces can be reliably fixed in our ciphers, and so we will start our search from the partial solution fixing the space. An example of the sort of graph that we consider is shown in Figure 4.2. If we were to use the cipher from Chapter 2, that is, $C =$ "*ifmmp_xpsme*", and if we were to fix the letters from the rightmost occurrence first, then the first level of solutions that we would consider would fix the "*e*". Thus, we would search partial solutions of the form $\{d:e\}$ and $\{v:e\}$. The second level of solutions would all fix the "*e*" and the "*m*", and so we would

search through solutions of the form $\{d{:}e,\ l{:}m\}$ and $\{v{:}e,\ d{:}m\}$. This would continue until the full solutions were listed.

Now, if Σ_C is even moderately large, this graph will clearly be too large to define explicitly in a real-world program. We therefore only work with it implicitly by keeping track of the partial solutions that we have searched and the edges linking out from them.

Given a suitable heuristic for this graph, then, we can run an A* search in order to find the optimal solution. Such a heuristic can be found by firstly defining $m(\sigma)$ as the maximum over the trigram probabilities of all strings of length $\text{len}(C)$ consistent with a partial solution σ. This means, in particular, that we take the maximum over the set of all strings in which the ciphertext letters not in the range of σ can be mapped to any plaintext letter not in σ and do not even need to be consistently mapped to the same plaintext letter in every instance. If $v(\sigma)$ is this maximum, then our heuristic will be $G_{max}(\sigma) = -\log(v(\sigma))$

Recalling the constraints given in Section 4.1, we can see that this heuristic is consistent in the A* sense: $v(\sigma)$ is an upper bound for the trigram probability of the plaintext generated by any full solution extending σ, since each such plaintext is clearly a string consistent with σ. Therefore, $G_{max}(\sigma)$ is a lower bound for the negative log of these probabilities, which are precisely the possible costs needed to reach t from σ on our graph. We can conclude that $G_{max}(\sigma)$ never overestimates the full cost needed to reach t. Furthermore, if σ' is any immediate extension of σ, it is also clear that the set of strings consistent with σ' is a subset of the set of strings consistent with σ. The value of $v(\sigma')$ is then not greater than $v(\sigma)$, and so G_{max}, in turn, is a nondecreasing heuristic.

Given this heuristic, then, the search portion of our algorithm is an A* search over our graph, and is depicted in Algorithm 4.3. In the algorithm shown, we ignore zero-probability partial solutions rather than place them onto the queue. The order by which we add the letters of Σ_C to partial solutions is the order of the distinct ciphertext letters in right-to-left order of their final occurrence in C. We denote the order of the letters

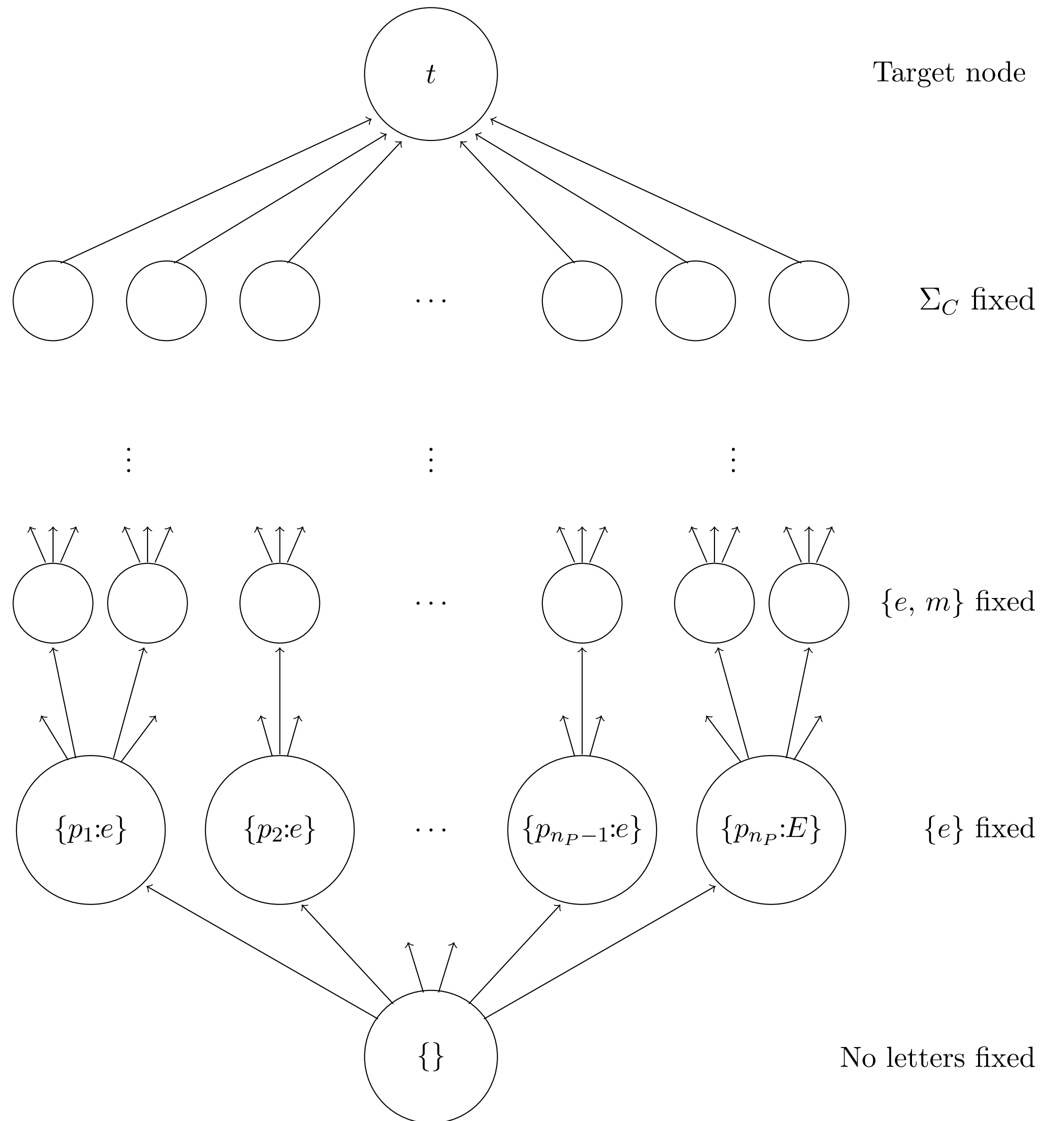


Figure 4.2: We form a graph over a subset of the partial solutions to the cipher *"ifmmp_xpsme"* by fixing each ciphertext letter according to the rightmost first order. The edges of the graph connect any partial solution to any of its immediate extensions.

with subscripts so that $c_1 \leq c_2 \leq \ldots c_{n_C}$. Other orderings for the alphabet Σ_C, such as most frequent first, are also possible, and are discussed later.

Due to the fact that an A* search with a consistent heuristic gives an optimal solution, it can be seen that our search will give us the full solution to the cipher that has the highest trigram probability, which is what we desire. However, we must still be able to find, for any partial solution σ, the value $G_{max}(\sigma)$. Naturally, a straightforward approach to this task will be inefficient. In general, the number of strings that we must consider to find $G_{max}(\sigma)$ will be exponential in the length of the cipher, and so we cannot consider every one separately. However, the probability of each string of length $\text{len}(C)$ is a straightforward function of its prefixes, and these prefixes are shared across many strings. In fact, it can be seen that, aside from the issue of ensuring that the strings considered are consistent with σ, this problem is essentially the same as that which is solved by the Viterbi algorithm. It is therefore reasonable to modify the Viterbi algorithm in a way that will allow us to solve this more constrained problem.

We do this by allowing the Viterbi algorithm to 'pinch' the probabilities around the letters in σ to ensure consistency. That is, we allow the Viterbi algorithm to run normally, except for the fact that if $p{:}c$ is in our partial solution σ, then we will force the output probability $\beta(c|p) = 1$ and force $\beta(c|p') = \beta(c'|c) = 0$ whenever $c' \neq c$, $p' \neq p$. All other output probabilities are assumed to be uniform, and in order to give the desired trigram probability at the end of the calculation, are set to 1. Due to the fact that the output score $\beta(c|p)$ is no longer technically a probability distribution under these calculations, and due to the fact that it takes the value of either 0 or 1, we do not explicitly refer to β in our specialized Viterbi algorithm. It can be seen that this specialization does not render the ability of the algorithm to return the highest probability string invalid, and so it will return the maximum probability $v(\sigma)$ over all strings of length $\text{len}(C)$ which are consistent with a partial solution σ. From that point, the value $G_{max}(\sigma) = -\log(v(\sigma))$ can easily be calculated. In fact, in the practical algorithm, the negative log domain

values are actually used throughout the calculation so as to maximize the numerical stability of the operations, although the actual steps used are the negative log domain counterparts of the regular steps.

Our calculation of $G_{max}(\sigma)$ has another difference from its Viterbi predecessor. As can be seen in Algorithm 4.3, Line 15, we do not, in general, directly calculate $G_{max}(\sigma)$ when we are actually searching the node associated with σ. Rather, we calculate G_{max} for the neighbors immediately reachable from it, which correspond to a subset of the immediate extensions of σ. We do this because, when conducting an A* search over a graph, we expand a node by finding the cost of its neighbors, not of the node itself. While the ciphertext letter to be fixed in this calculation (denoted c_{s+1} in Algorithm 4.3) is given, the calculation must be performed for each unfixed plaintext letter p. Rather than run each calculation separately, we find them in parallel in a single function call. This function call is denoted $SVit(\sigma, c_{s+1})$. A call to this function will return a $\text{len}(C) \times n_P \times n_P$ table G_{big}. The value $G_{max}(\sigma \cup \{p : c_{s+1}\})$ referenced in Algorithm 4.3 is found by taking $\max_{l \in \Sigma_P}(G_{big}(\text{len}(C) - 1,\ l,\ p))$.

The function SVit(σ, c), depicted in Algorithm 4.4, uses dynamic programming to score every immediate extension of a given partial solution in tandem, by finding, in a manner consistent with the real Viterbi algorithm, the most probable input string given a set of output symbols, which in this case is the cipher C. A call to SVit(σ, c) manages the task of finding our heuristic by filling in a table G_{big} such that for all $1 \leq i \leq \text{len}(C)$, and l, $k \in \Sigma_P$, $G_{big}(i,\ l,\ k)$ is the maximum probability over every plaintext string S for which:

- $\text{len}(S) = i$,
- $S[i] = l$,
- for every p in the domain of σ, every $1 \leq j \leq i$, $S[j] = p$ iff $C[j] = \sigma(p)$, and
- for every position $1 \leq j \leq i$, $S[j] = k$ iff $C[j] = c$.

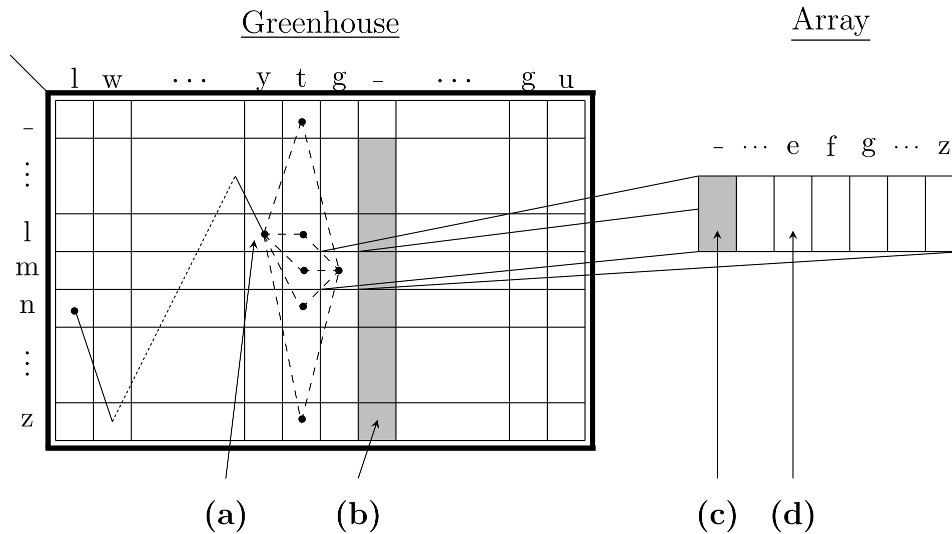
The real Viterbi algorithm lacks these final two constraints, and would only store a single cell at $G(i, l)$. There, G is called a trellis. Our G_{big} is a collection of many parallel trellises, which we refer to as a *greenhouse*.

The table is completed by filling in the columns from $i = 0$ to len(C) in order. For every column i, we let $C[i]$ represent the value of C at i. We will iterate over the values of l and over the values of k such that $k\!:\!c$ and $l\!:\!C[i]$ are consistent with σ. Because we are using a trigram character model, the cells in the first and second columns must be primed with unigram and bigram probabilities. The remaining probabilities are calculated by searching through the cells from the previous two columns, using the entry at the earlier column to indicate the probability of the best string up to that point, and searching through the trigram probabilities over two additional letters. Cells that would produce inconsistencies are left at zero, and these as well as cells that the language model assigns zero to can only produce zero entries in later columns. This process is depicted in Figure 4.3.

As in the real Viterbi algorithm, backpointers are necessary to reference one of the two language model probabilities. With this in mind, we also keep reference to the second-last letter in the optimal string ending at $G_{big}(i, l, k)$ for every l and k and for every $i \geq 1$. We refer to this letter as $B_{big}(i, l, k)$.

In order to decrease the search space, we add the further restriction that the solutions of every three-character sequence must be consistent: if the ciphertext indicates that two adjacent letters are the same, then only the plaintext strings that map the same letter to each will be considered. The number of letters that are forced to be consistent is three because consistency is enforced by removing inconsistent strings from consideration during trigram model evaluation.

Because every partial solution is only obtained by extending a solution of size one less, and extensions are only made in a predetermined order of cipher alphabet letters, every partial solution is only enqueued and extended once.



Each cell in the greenhouse is indexed by a position in the cipher (denoted as i) and a plaintext letter (denoted as l). Each cell consists of a smaller array which is indexed by the plaintext letter k. The cells in the array give the best probabilities of any path passing through the greenhouse cell, given that the index character of the array maps to the character in column c, where c is the next ciphertext character to be fixed in the solution. The probability is set to zero if no path can pass through the cell.
This is the case, for example, in (b) and (c), where the knowledge that "_" maps to "_" would tell us that the cells indicated in gray are unreachable. The cell at (d) is filled using the trigram probabilities and the probability of the path at starting at (a).

Figure 4.3: Filling the Greenhouse Table.

SVit is highly parallelizable. The $n_P \times n_P$ cells of every column i do not depend on each other — only on the cells of the previous two columns $i-1$ and $i-2$, as well as the language model. In our implementation of the algorithm, we have written the underlying program in C/C++, and we have used the CUDA library developed for NVIDIA graphics cards in order to implement the parallel sections of the code.

Algorithm 4.3 A* Section of the Search Algorithm

Input: The ciphertext C and (optionally) the starting partial solution σ_0.

The function call is of the form: Decipher(C, $[\sigma_0]$)

Output: The most likely full solution σ to C.

1: **if** σ_0 is not given as an argument **then**
2: $\sigma_0 = \{\}$.
3: Order the letters $c_1 \dots c_{n_C}$ by rightmost occurrence in C.
4: Create a priority queue Q for partial solutions, ordered by highest probability.
5: Push the starting σ_0 solution onto the queue.
6: **while** Q is not empty **do**
7: Pop the best partial solution σ from Q.
8: $s = |\sigma|$.
9: **if** $s = n_C$ **then**
10: Return σ
11: **else**
12: Set $SV = SVit(\sigma, c_{s+1})$
13: **for** all p not in the range of σ **do**
14: Define $G_{max}(\sigma \cup \{p{:}c_{s+1}\})$ as the heuristic plus the known cost at $(\sigma \cup \{p{:}c_{s+1}\})$
15: Set $G_{max}(\sigma \cup \{p{:}c_{s+1}\}) = \max_{l \in \Sigma_P}(SV(\text{len}(C) - 1, l, p))$.
16: **if** $G_{max}(\sigma \cup \{p{:}c_{s+1}\}) < \infty$ **then**
17: Push $\sigma \cup \{p{:}c_{s+1}\}$ onto Q with the score $G_{max}(\sigma \cup \{p{:}c_{s+1}\})$.
18: Return "Solution Infeasible".

Algorithm 4.4 Specialized Viterbi Algorithm

Input: Partial solution σ and ciphertext character c.

The function call is of the form: $SVit(\sigma, c)$.

Output: Greenhouse G_{big}.

1: Initialize G_{big} to 0 and B_{big} to null.

2: **for all** (l, k) such that $\sigma \cup \{k{:}c, l{:}C[0]\}$ is consistent **do**

3: $G_{big}(0, l, k) = P(l)$.

4: **for all** (l, k) such that $\sigma \cup \{k{:}c, l{:}C[1]\}$ is consistent **do**

5: **for all** j such that $\sigma \cup \{k{:}c, l{:}C[1], j{:}C[0]\}$ is consistent **do**

6: $G_{big}(1, l, k) = \max(G_{big}(1, l, k), G_{big}(0, j, k) \times P(l|j))$

7: $B_{big}(1, l, k) =$ the j for which this max was found.

8: **for all** (l, k) such that $\sigma \cup \{k{:}c, l{:}C[2]\}$ is consistent **do**

9: **for all** j_1, j_2 such that $\sigma \cup \{k : c, j_2{:}C[0], j_1{:}C[1], l{:}C[2]\}$ is consistent **do**

10: $G_{big}(2, l, k) = \max(G_{big}(2, l, k), G_{big}(0, j_2, k) \times P(j_1|j_2) \times P(l|j_2 j_1))$.

11: $B_{big}(2, l, k) =$ the j_1 for which this max was found.

12: **for** $i = 3$ to $\text{len}(C) - 1$ **do**

13: **for all** (l, k) such that $\sigma \cup \{k{:}c, l{:}C[i]\}$ is consistent **do**

14: **for all** j_1, j_2 such that $\sigma \cup \{k{:}c, j_2 : C[i-2], j_1{:}C[i-1], l : C[i]\}$ is consistent **do**

15: $G_{big}(i, l, k) = \max(G_{big}(i, l, k), G_{big}(i-2, j_2, k) \times P(j_1|j_2 B(i, j_2, k)) \times P(l|j_2 j_1))$.

16: $B_{big}(i, l, k) =$ the j_1 for which this max was found.

17: Return G_{big}.

Chapter 5

Experiments, Part 1

As we have stated in Chapter 4, our algorithm has been designed with the application of transliterating websites in mind. In order to gain realistic test data for this application, we have operated on the assumption that Wikipedia is a good approximation of the type of language that will be found in most web pages. We sampled a sequence of English-language articles from Wikipedia using their random page selector, and these were used to create a set of reference pages. In order to minimize the common material used in each page, only the text enclosed by the paragraph tags of the main body of the pages was used.

A rough search over web pages has shown that a length of 1000 to 11000 characters is a realistic length for many articles, although this can vary according to the genre of the page. Wikipedia, for example, has entries that are one sentence in length. We have run two groups of tests for our algorithm.

5.1 The Test Sets and Language Model

In the first set of tests, we chose the mean of the above lengths to be our sample size, and we created and decoded 10 ciphers of this size (i.e., different texts, same size). We made these cipher texts by concatenating the contents of randomly chosen Wikipedia pages

until they contained at least 6000 characters, and then using the first 6000 characters of the resulting files as the plaintexts of the cipher. The text length was rounded up to the nearest word where needed.

In the second set of tests, we used a single long ciphertext, and measured the time required for the algorithm to finish a number of prefixes of it (i.e., same text, different sizes). The plaintext for this set of tests was developed in the same way as the first set, and the input ciphertext lengths considered were 1000, 3500, 6000, 8500, 11000, and 13500 characters.

In all of the data considered, the frequency of spaces was far higher than that of any other character, and so in any real application the character corresponding to the space can likely be guessed without difficulty. The ciphers we have considered have therefore been simplified by allowing the knowledge of which character corresponds to the space. It appears that Ravi and Knight (2008) did this as well. We add the knowledge of which character corresponds to a space by starting our search from the partial solution that correctly fixes the space. Our algorithm will still work without this assumption, but would take longer.

Our character-level language model used was developed from the first 1.5 million characters of the Wall Street Journal section of the Penn Treebank corpus. The characters used in the language model were the upper- and lower-case letters, spaces, and full stops; other characters were skipped when counting the frequencies. Furthermore, the number of sequential spaces allowed was limited to one in order to maximize context and to eliminate any long stretches of white space. In the event that a trigram or bigram was found in the plaintext that was not counted in the language model, add-one smoothing was used. As discussed previously, the space character is assumed to be known.

5.2 Measurements

When testing our algorithm, we judged the time complexity of our algorithm by measuring the actual time taken by the algorithm to complete its runs, as well as the number of partial solutions placed onto the queue ("enqueued"), the number popped off the queue ("expanded"), and the number of zero-probability partial solutions not enqueued ("zeros") during these runs. These latter numbers give us insight into the quality of trigram probabilities as a heuristic for the A* search.

We judged the quality of the decoding by measuring the percentage of characters in the cipher alphabet that were correctly guessed, and also the word error rate of the plaintext generated by our solution. The second metric is useful because the frequencies of the ciphertext characters in the cipher is not close to uniform. If a low-probability character in the ciphertext is guessed wrongly the resulting plaintext would be more accurate than indicated by the number of correctly guessed characters in the cipher alphabet. Counting the actual number of word errors is meant as an estimate of how useful or readable the plaintext will be.

We would have liked to compare our results with those of Ravi and Knight (2008), but the method presented there was simply not feasible on texts and (case-sensitive) alphabets of this size with the computing hardware at our disposal.

5.3 Results, Part 1

In our first set of tests, we measured the time consumption and accuracy of our algorithm over 10 ciphers taken from random texts that were 6000 characters long. The time values in these tables are given in the format of (H)H:MM:SS. The results from this run appear in Table 5.1. All running times reported in this section were obtained on a computer running Ubuntu Linux 8.04 with 50 GB of RAM and 16×1.6 GHz CPU cores. Column-level subcomputations in the greenhouse were dispatched to two NVIDIA Tesla S1070

Cipher	Time	Enqueued	Expanded	Zeros	Accuracy	Word Error Rate
1	2:52:58	964	964	44157	100%	0%
2	0:18:16	132	132	5197	100%	0%
3	0:07:23	91	91	3080	100%	0%
4	22:58:05	6238	6238	272327	100%	0%
5	84:05:33	16678	16678	714002	100%	0%
6	7:32:37	2521	2521	114283	100%	0%
7	8:30:41	2626	2626	116392	100%	0%
8	4:49:46	1483	1482	66070	100%	0%
9	13:03:22	4814	4814	215086	100%	0%
10	1:57:05	950	950	42107	100%	0%

Table 5.1: Time consumption and accuracy on a sample of 10 6000-character texts.

GPU cards that are attached through 16-lane PCI Express adapters. Each card has 16 GB of cache memory, a 602 MHz core processor and 4×240 shader processors operating in parallel at 1440 MHz each.

In our second set of tests, we measured the time consumption and accuracy of our algorithm over several prefixes of different lengths of a single 13500-character ciphertext. The results of this run are given in Table 5.2.

The first thing to note in this data is that the accuracy of this algorithm is above 90% for all of the test data, and 100% on all but the smallest 2 ciphers. We can also observe that even when there are errors (e.g., in the size 1000 cipher), the word error rate is very small. This is a Zipf's Law effect — misclassified characters come from poorly attested character trigrams, which are in turn found only in longer, rarer words. The overall high accuracy is probably due to the relatively large size of the texts: as the ciphertexts get longer, the number of solutions and the number of partial solutions that are reasonable decrease drastically. The results do show, however, that character trigram probabilities

Size	Time	Enqueued	Expanded	Zeros	Accuracy	Word Error Rate
1000	57:56:01	119759	119755	5172631	92.59%	1.89%
3500	0:52:35	615	614	26865	96.30%	0.17%
6000	0:16:31	147	147	5709	100%	0%
8500	12:45:55	1302	1302	60978	100%	0%
11000	1:28:05	210	210	8868	100%	0%
13500	1:15:19	219	219	9277	100%	0%

Table 5.2: Time consumption and accuracy on prefixes of a single 13500-character ciphertext.

are an effective indicator of the most likely solution, even when the language model and test data are from very different genres (here, the Wall Street Journal and Wikipedia, respectively). These results also show that our algorithm is effective as a way of decoding simple ciphers.

As far as the running time of the algorithm goes, we see a substantial variance: from a few minutes to several hours for most of the longer ciphers. Specifically, there is substantial variability in the running times seen.

Desiring to reduce the variance of the running time, we look at the second set of tests for possible causes. In this test set, we see a similar variation in running time. Specifically, the length 8500 cipher generates more solutions, and runs for longer, than the length 6000 or 11000 ones. This indicates that the length of a cipher is not the only factor in determining its complexity. Moreover, since the ciphers are all prefixes of the same string, the plaintext language cannot be the deciding factor either.

There is a major difference between these ciphers, however, that can explain these differences. Recall that the algorithm fixes characters starting from the end of the cipher, these prefixes have very different character orderings, $c_1, \ldots, c_{n_C}$, and thus a very different order of partial solutions. The running time of our algorithm depends very crucially on

these initial conditions.

If we use the data of Table 5.1 as an indicator of the time and search complexity between ciphers of the same length, then we can argue that the time required for a cipher to run correlates very well with the number of nodes that must be searched. This is not the case for the data of Table 5.2. Here, we see that the average time taken per run of $SVit$ (i.e., TIME / EXPENDED) increases with the cipher length for the ciphers that are shorter than 11000 characters. This is naturally to be expected, since the size of the table to be filled is a linear function of the cipher length. The time taken per run of $SVit$ most likely decreases for the ciphers of length 11000 and 13500 because of the order in which the cipher letters are fixed. Fixing the ciphertext letters in an efficient manner will allow more zero probability strings to identified as the table is filled, and this will make the algorithm run more efficiently.

If we take this difference into account, the data also suggests that there is generally a decrease in search complexity as the length of the cipher increases. That is, although all of the times in Table 5.2 are within the expected variation indicated in Table 5.1, the number of nodes searched per cipher are most certainly not. Specifically, the number of nodes searched for the length 1000 cipher is almost an order of magnitude more than the largest measurement in Table 5.1.

An overall decrease in the number of nodes that must be searched in longer ciphers is expected, as well: as a cipher string gets longer, it will likely give more clues as to which partial solutions are correct and which ones are not. This leads to an overall decrease in the complexity of the search space. Overall, the time required for each sweep of the Viterbi algorithm increases with cipher length, but this is more than offset by the decrease in the number of required sweeps.

Perhaps most interestingly, we note that the number of enqueued partial solutions is in every case identical or nearly identical to the number of partial solutions expanded. From a theoretical perspective, we must also remember the zero-probability solutions, which

should in a sense count when judging the effectiveness of our A* heuristic. Naturally, these are ignored by our implementation because they are so badly scored that they could never be considered. Nevertheless, what these numbers show is that scores based on character-level trigrams, while theoretically admissible, are not all that clever when it comes to navigating through the search space of all possible letter substitution ciphers, apart from their very keen ability at assigning zeros to a large number of partial solutions. A more complex heuristic that can additionally rank non-zero probability solutions with more prescience would likely make a great difference in the running time of this method.

Chapter 6

An Improved Algorithm

As we have shown, our algorithm has an undesirably high degree of variability between different ciphers. With this in mind, we have worked to improve our method to better reflect the structure of the problem. We can do this in two ways: we can change the graph that we search or our heuristic over that graph.

Our method for changing the graph is straightforward. We have already observed that the order in which the letters are fixed has a large influence on the complexity of the search. This letter order determines the graph we search, and so one of the changes that we will make will be to change the order in which the letters are fixed so as to reduce the overall search complexity. Unfortunately, the optimal order in which we must fix the cipher letters is difficult to determine. We have experimented with the most-frequent-first regimen, and found that it performs worse than the original. Our hypothesis is that this is due to the fact that the most frequent character tends to appear in many high-frequency trigrams, and so our priority queue becomes very long because of a lack of low-probability trigrams to knock the scores of partial solutions below the scores of the extensions of their better scoring but same-length peers. A least-frequent-first regimen has the opposite problem, in which their rare occurrence in the ciphertext provides too few opportunities to potentially reduce the score of a candidate.

It is possible to find a good approximation to the optimal order, however, and we have implemented such an approximation in our improved algorithm. We find our letter-fixing order by running the first iteration $SVit$ (usually $SVit(\{\}, c)$) in parallel for every possible value of c. Thus, for every such run we will obtain the set of partial solutions that fix a particular ciphertext letter c. Due to the fact that some of the solutions are pruned since they have probability of zero, these sets are not all the same size. We argue that the ciphertext letters whose sets of partial solutions are smaller are inherently more constrained by the cipher, and so are good choices to be fixed first. We therefore fix our letters in an order such that the ones with the smallest number of partial solutions in the first run are fixed first. In the case of a tie, in which two ciphertext letters generate the same number of partial solutions, we revert to the last-first ordering that we used in the original algorithm. In a sense, what we are doing is initially fixing the letters that give us a low branching factor when we are close to the source of our graph. Later on, we will still have to fix the letters that give large branching factors, but these letters will be fixed when our partial solutions are much larger, and so there will be extra constraints that will lower the branching factor at that point, as well.

Ideally, we would like to locally choose which ciphertext letter to fix at every node we search, but we will find this to be impractical. If the order of letter addition is not the same regardless of the branch of the search, we would have to ensure that no single partial solution is enqueued more than once (i.e., we would have to prevent situations in which one branch in our search would hold the solutions $\{z{:}a\}$, $\{z{:}a,\ y{:}b\}$, $\{z{:}a,\ y{:}b,\ x{:}c\}$ while another branch would hold $\{z{:}a\}$, $\{z{:}a,\ x{:}c\}$, $\{z{:}a,\ x{:}c,\ y{:}b\}$). We therefore use our comparison of partial solutions as a way of fixing letter order only once, at the outset of our calculations.

We also would like to tighten our heuristic so as to make it better reflect the actual probability cost on the search space. Such a heuristic would naturally lower the time required to solve the ciphers, and would be less susceptible to the variability that plagues

the initial version of the algorithm. A simple way to incorporate more information into the heuristic is to fix more than one letter in each step. However, fixing more than one letter in a step is likely to simply exchange a reduction in depth with a large increase in the branching factor of the search, and so cannot be expected to give a substantial reduction in search space complexity. We therefore continue to fix only one letter per step, and instead try to make use of more information in every step.

It can be seen that there is a ready source of new information on the cipher, as well. Recall that we are already running $SVit$ in parallel for every unfixed cipher letter c in the initial step of the new algorithm. It would seem to be a waste if we were to run all of these calculations and then throw most of the resulting data away. We will therefore make use of the partial solutions generated by all of the ciphertext letters, although we ultimately only leave one fixed.

Specifically, consider once again the cipher $C =$ *"ifmmp_xpsme"*. In the initial step of our new method, we run $SVit$ once for each letter *"i"*, *"f"*, *"m"*, *"p"*, *"x"*, *"s"*, and *"e"*. After these calculations we might find that the *"m"* is the most constrained letter, and so we will fix *"m"* first. We might also find, however, that the solution $\{q{:}p\}$ has to be pruned because it gives a zero-probability solution. It makes sense to send this information to later steps of the algorithm, even though *"m"* is the letter that is fixed — if there is no nonzero-probability string mapping *"q"* to *"p"* without *"m"* fixed, there will certainly be no such string after *"m"* is fixed, either.

Adding the information $q{\nmid}p$ to the partial solutions fixing *"m"* allows us to streamline the subsequent iterations of our $SVit$ algorithm. Even if *"p"* is not the next letter to be fixed, we can force the Viterbi algorithm itself to ignore the strings that map *"q"* to *"p"* as it fills the greenhouse. This will give us fewer strings to consider, and so will in turn increase the likelihood that we will run into zero-probability solutions in later steps.

We can go through the same process at every step of the algorithm: regardless of which ciphertext letter we ultimately fix, we run $SVit$ for every unfixed letter. As we

fill the greenhouse, we ignore mappings that have already been shown to produce zero probability solutions in the ancestor nodes of the current solution. In turn, we pass all known zero-probability solutions to the child solutions that will be expanded later. This allows us to aggressively prune dead-end solutions from our search graph, and so will allow us to find our solution much faster.

Of course, we also have to ensure as we do this process that the completeness and correctness of our search is maintained. This presents no difficulty. Since we are only pruning strings that have a zero-probability mapping as we fill in the greenhouse, any feasible string (and, specifically, the correct string) will be considered, and so will produce a solution with a nonzero probability. In fact, the final probability when we fill that section of the greenhouse will still be at least that of the best feasible full solution extending the current partial solution. Moreover, as we pass through the graph we only add constraints to the strings considered, and so the probability at the end of the greenhouse will still be nonincreasing. In the negative log domain, this means that the correct solution is still reachable, and that our score for the nodes that we search will be a lower bound for the true score and will be nondecreasing. The score we compute for each partial solution is therefore still a consistent heuristic, and so our search using it is still complete and correct.

Putting these changes together, we implement our modified algorithm by altering our original function $SVit$ into a new function $SVit'$. This new function is essentially a set of parallel calls to $SVit$, save for the fact that it allows extra constraints, in the form of forbidden mappings, to be used when filling the table. We have also changed the search section of our algorithm to maintain the overhead required for the use of this new function, as well as to determine the new order in which the ciphertext letters will be fixed. The code for the revised A* section of the search is given in Algorithm 6.1, and the new function $SVit'$ is given in Algorithm 6.2.

Algorithm 6.1 New A* Section of the Search Algorithm

Input: The ciphertext C and (optionally) the starting partial solution σ_0.

The function call is of the form: Decipher$'(C, [\sigma_0])$.

Output: The most likely full solution σ to C.

1: **if** σ_0 is not given as an argument **then**
2: $\quad \sigma_0 = \{\}$.
3: Create a priority queue Q for tuples of the partial solutions σ and the sets of forbidden mappings F, ordered by highest probability.
4: Push the starting σ_0 solution onto the queue.
5: **while** Q is not empty **do**
6: $\quad$ Pop the best partial solution and forbidden mapping set (σ, F) from Q.
7: $\quad s = |\sigma|$.
8: $\quad$ **if** $s = n_C$ **then**
9: $\quad\quad$ Return σ
10: $\quad$ **else**
11: $\quad\quad$ Set $SV = SVit'(\sigma,\, F)$
12: $\quad\quad$ **if** This is the first iteration of the while loop **then**
13: $\quad\quad\quad$ Order the letters $c_1 \ldots c_{n_C}$ according to which is the most constrained in SV. Break ties by rightmost first.
14: $\quad\quad$ Let $SV_{c_{s+1}}$ be the greenhouse in SV corresponding to c_{s+1}.
15: $\quad\quad$ Let F' be F updated with all mappings rendered impossible in SV.
16: $\quad\quad$ **for** all p not in the range of σ **do**
17: $\quad\quad\quad$ Set the value $G(\sigma \cup \{p{:}c_{s+1}\})$ to be $\max_{l \in \Sigma_P}(SV_{c_{s+1}}(\text{len}(C) - 1,\, l,\, p))$.
18: $\quad\quad\quad$ **if** $G(\sigma \cup \{p{:}c_{s+1}\}) < \infty$ **then**
19: $\quad\quad\quad\quad$ Push $(\sigma \cup \{p{:}c_{s+1}\}, F')$ onto Q with the score $G(\sigma \cup \{p{:}c_{s+1}\})$.
20: Return "Solution Infeasible".

Algorithm 6.2 The New Specialized Viterbi Algorithm

Input: partial solution σ and a set F of forbidden mappings of the form $p \not: c$.

The function call is of the form: $SVit'(\sigma, F)$.

Output: Set of greenhouses G_c for every $c \notin \sigma$.

1: **for all** c not fixed in σ **do**
2: Create and initialize the tables G_c, B_c.
3: **for all** (l, k) such that $\sigma \cup \{k{:}c, l{:}C[0]\}$ is consistent, $\{k \not: c, l \not: C[0]\} \cap F = \varnothing$ **do**
4: $G_c(0, l, k) = P(l)$.
5: **for all** (l, k) such that $\sigma \cup \{k{:}c, l{:}C[1]\}$ is consistent, $\{k \not: c, l \not: C[1]\} \cap F = \varnothing$ **do**
6: **for all** j such that $\sigma \cup \{k{:}c, l{:}C[1], j{:}C[0]\}$ is consistent and $j \not: C[0] \notin F$ **do**
7: $G_c(1, l, k) = \max(G_c(1, l, k), G_c(0, j, k) \times P(l|j))$
8: $B_c(1, l, k) =$ the j for which this max was found.
9: **for all** (l, k) such that $\sigma \cup \{k{:}c, l{:}C[2]\}$ is consistent, $\{k \not: c, l \not: C[2]\} \cap F = \varnothing$ **do**
10: **for all** j_1, j_2 such that $\sigma \cup \{k : c, j_2 : C[0], j_1 : C[1], l : C[2]\}$ is consistent and $\{j_1 \not: C[1], j_2 \not: C[0]\} \cap F = \varnothing$ **do**
11: $G_c(2, l, k) = \max(G_c(2, l, k), G_c(0, j_2, k) \times P(j_1|j_2) \times P(l|j_2 j_1))$.
12: $B_c(2, l, k) =$ the j_1 for which this max was found.
13: **for** $i = 3$ to $\text{len}(C) - 1$ **do**
14: **for all** (l, k) such that $\sigma \cup \{k{:}c, l{:}C[i]\}$ is consistent, $\{k \not: c, l \not: C[i]\} \cap F = \varnothing$ **do**
15: **for all** j_1, j_2 such that $\sigma \cup \{k{:}c, j_2 : C[i-2], j_1{:}C[i-1], l : C[i]\}$ is consistent and $\{j_1 \not: C[i-1], j_2 \not: C[i-2]\} \cap F = \varnothing$ **do**
16: $G_c(i, l, k) = \max(G_c(i, l, k), G_c(i-2, j_2, k) \times P(j_1|j_2 B_c(i, j_2, k)) \times P(l|j_2 j_1))$.
17: $B_c(i, l, k) =$ the j_1 for which this max was found.
18: Return $\{G_c | c \notin \sigma\}$.

This new version of the algorithm maintains the property that the $n_P \times n_P$ cells of every column in each of the n_C different tables do not depend on each other, and so each can be filled in parallel. We have once again written the underlying algorithm in C/C++, and we have used the CUDA library to implement the parallel sections of the code.

Chapter 7

Experiments, Part 2

In our second round of experiments we have tested the performance of the improved version of our algorithm using SVit$'$ as compared to the performance of the algorithm using SVit. We have therefore tested this modification of our algorithm in the same task as before. The conditions of our tests were also the same as those in Chapter 5. We have, however, added an additional two sets of ciphers to be used in these experiments. The additional test sets have been constructed in the same manner as those used in Chapter 5, in that one is a set of 10 6000 character ciphers and the other is a set of prefixes taken from the same 13500 character long ciphertext. These extra test sets have been added to this round of experiments because our improved algorithm was developed in response to our observations regarding the results in Section 5.3. They serve the purpose of ensuring that our test sets are not so uniform over multiple generations of our algorithm that we have to worry about overfitting.

7.1 Results, Part 2

In our first set of tests, we looked at the performance of the improved version of our algorithm over 10 6000 character ciphertexts. Once again, the time values in these tables are in the format of (H)H:MM:SS. The results of this run are given in Table 7.1.

Cipher	Time	Enqueued	Expanded	Zeros	Accuracy	WER
1	2:32:32	56	56	1476	100%	0%
2	2:16:57	52	52	1325	100%	0%
3	2:15:29	52	52	1325	100%	0%
4	2:32:57	55	55	1410	100%	0%
5	2:34:37	65	65	1851	100%	0%
6	2:36:52	55	55	1426	100%	0%
7	2:46:59	51	51	1778	100%	0%
8	2:39:08	57	57	2576	100%	0%
9	2:31:57	48	48	1465	100%	0%
10	2:46:36	53	53	1362	100%	0%

Table 7.1: Time consumption and accuracy of the improved algorithm on the original set of 10 6000-character ciphertexts.

Version	Size	Time	Enqueued	Expanded	Zeros	Accuracy	WER
New	**1000**	**4:31:17**	**1296**	**1293**	**45892**	**92.59%**	**1.89%**
Old	1000	57:56:01	119759	119755	5172631	92.59%	1.89%
New	**3500**	**1:35:12**	**55**	**54**	**1479**	**96.30%**	**0.17%**
Old	3500	0:52:35	615	614	26865	96.30%	0.17%
New	**6000**	**2:27:33**	**57**	**57**	**1565**	**100%**	**0%**
Old	6000	0:16:31	147	147	5709	100%	0%
New	**8500**	**3:22:33**	**56**	**56**	**1530**	**100%**	**0%**
Old	8500	12:45:55	1302	1302	60978	100%	0%
New	**11000**	**4:18:02**	**56**	**56**	**1530**	**100%**	**0%**
Old	11000	1:28:05	210	210	8868	100%	0%
New	**13500**	**5:10:01**	**56**	**56**	**1530**	**100%**	**0%**
Old	13500	1:15:19	219	219	9277	100%	0%

Table 7.2: Time consumption and accuracy of the original ("Old") and improved ("**New**") algorithm on prefixes of the original 13500-character ciphertext.

In our second set of tests, we compared the time consumption and accuracy of the two versions of our algorithm over several prefixes of different lengths of the original 13500 character ciphertext. The results of this run are given in Table 7.2.

After the first two sets of tests, we swapped out the original test sets for the new ones. These sets were of the same form as the old ones, and so our third set of test measured the time consumption and accuracy of the new version of our algorithm over 10 new ciphers taken from random texts that were 6000 characters long. The results of this run are given in Table 7.3.

Cipher	Time	Enqueued	Expanded	Zeros	Accuracy	WER
1	2:45:40	58	58	1632	100%	0%
2	2:34:26	58	58	1630	100%	0%
3	2:19:29	53	53	1477	100%	0%
4	2:22:56	53	53	1525	100%	0%
5	2:34:31	55	55	1670	100%	0%
6	2:53:32	70	70	2281	100%	0%
7	2:17:50	51	51	1515	100%	0%
8	2:41:21	57	57	1725	100%	0%
9	2:15:54	48	48	1408	100%	0%
10	2:29:37	53	53	1661	100%	0%

Table 7.3: Time consumption and accuracy of the improved algorithm on the held-out set of 10 6000-character ciphertexts.

Version	Size	Time	Enqueued	Expanded	Zeros	Accuracy	WER
New	**1000**	**2:07:12**	**314**	**308**	**13907**	**95.65%**	**2.56%**
Old	1000	442:49:47	1194831	1194817	51738099	95.65%	2.56%
New	**3500**	**1:35:15**	**54**	**54**	**1479**	**100%**	**0%**
Old	3500	0:36:30	334	334	14119	100%	0%
New	**6000**	**2:34:25**	**54**	**54**	**1479**	**100%**	**0%**
Old	6000	10:11:56	4144	4144	198044	100%	0%
New	**8500**	**3:28:59**	**55**	**55**	**1529**	**100%**	**0%**
Old	8500	0:13:54	85	85	2944	100%	0%
New	**11000**	**4:23:07**	**54**	**54**	**1479**	**100%**	**0%**
Old	11000	0:38:30	148	148	6031	100%	0%
New	**13500**	**5:22:54**	**55**	**55**	**1480**	**100%**	**0%**
Old	13500	3:16:55	460	460	20787	100%	0%

Table 7.4: Time consumption and accuracy of the original ("Old") and improved ("**New**") algorithm on prefixes of a held-out 13500-character ciphertext.

Finally, we compared the time consumption and accuracy of the two versions of our algorithm over several prefixes of different lengths of a new 13500 character ciphertext. The results of this run are given in Table 7.4.

The first thing we note in these results is that the accuracy and the word error rate for the prefix ciphers in Table 7.2 and Table 7.4 do not depend on the algorithm used, but only on the cipher that is run. This is to be expected, since both algorithms find the model optimal solution for the cipher. The main difference between the algorithms is in their efficiency and stability. As before, the accuracy of the results is above 90% for all ciphers, and is 100% for all ciphers of a length of at least 6000 characters.

When comparing the the number of nodes searched between the two algorithms, we

see an across-the-board improvement in the improved version. In fact, the efficiency of the search that is made by the improved algorithm is very close to the best that could be run in this particular framework. That is, so long as we fix only one letter per run of the generalized Viterbi algorithm, we will need at least as many runs as there are letters in the ciphertext alphabet. The size of the ciphertext alphabet in the ciphers used in the above experiments varies, but always lies between 46 and 53 letters. Thus, for every cipher of at least 3500 characters, the overall number of steps in the search is close to the minimum number of steps.

This is somewhat offset by the increase in time required for the improved algorithm to run. While the improved algorithm can is faster than the original for a few of the ciphers, we can see that the original version of the cipher is actually the faster version much of the time. The overall slowdown of the improved algorithm is to be expected, however, since a run of SVit$'$ is in a sense made up of many runs of SVit. Although these runs are calculated in parallel, the number of parallel calculations that can be made at one time on our hardware is still limited, and so there is also an overall increase in time. We argue that the improved version of our algorithm is still an improvement over the original version in spite of this slowdown due to the fact that we also find it to be very stable across different ciphers. In our data, we see that for the original algorithm, the difference in the required running time across ciphers is not easily related to the length of the cipher. The times required to run the 3500, 6000, 11000 and 13500 character ciphers in Table 7.2 and the 3500, 8500 and 11000 character ciphers in Table 7.4 are low, while we see a spike in the times required for the 8500 character cipher in Table 7.2 and the 6000 and 13500 character ciphers in Table 7.4. The data from the new test sets also confirm our earlier observation that the short 1000 character cipher can be expected to require many more iterations than the longer ones.

In contrast, the improved version of our algorithm has very little variation in running time or efficiency across ciphers. In Table 7.1, we see that the variation in the time

required to solve the different ciphers is much lower than that which is seen in Table 5.1. The results shown in Table 7.3 are also very stable. The difference in stability is also seen in Table 7.2 and Table 7.4. Where the original version of the algorithm is more dependent on the order in which letters occur than on the length of the cipher, we see a clear relation between cipher length and the running time of the improved algorithm. For the ciphers of at least 3500 characters, the running time of the improved algorithm is roughly linear. This is to be expected, since the number of runs of the generalized Viterbi algorithm is almost constant for these ciphers, and since the main difference between these ciphers in the time required for each run of the generalized Viterbi algorithm is the length of the cipher. This trend does not extend to the 1000 character cipher because it requires more generalized Viterbi runs to solve. A shorter cipher places fewer constraints on the strings in the Viterbi table, and so more strings get passed on to the A* queue. It is likely that the same effect can be exploited for the original version of the algorithm, but since we can see a spike in running time due to the order of letter fixing even for the cipher of 13500 characters, we expect that the cipher length needed to guarantee stability would be higher than is practical.

Finally, our second set of experiments confirms our observation in Section 5.3 that the number of enqueued partial solutions is in every case very nearly identical to the number of partial solutions that are expanded. The difference between the two sets of results lies in the number of partial solutions that are enqueued. The first change in our improved algorithm forces the program to use information from previous runs to aggressively prune the strings that are considered acceptable during the process of filling the Viterbi table, and this in turn decreases the number of partial solutions that make it on to the queue at all. The second change in our improved algorithm is an attempt to minimize the branching factor of the A* tree. In neither case is the value of the trigram probability as a score for the goodness of the solution severely impacted on. The improvement in

our results stems from a greater use of the information gleaned from the value of the presence, as opposed to the actual value, of a trigram probability in our language model.

Chapter 8

Conclusions

In this paper, we have developed and presented an algorithm for solving letter-substitution ciphers, with an eye towards discovering unknown encoding standards in electronic documents on the fly. In a test of the fully developed version of our algorithm over ciphers drawn from Wikipedia, we found its accuracy to be 100% on almost all of the ciphers that it solved. We found that the running time of our algorithm is stable across different ciphers, and that that the linear-time theoretical complexity of this method can be seen for ciphers whose sizes are within our range of interest. For sufficiently short ciphers, the running time tends to decrease with an increase in ciphertext length due to our character-level language model's facility at eliminating highly improbable solutions. This facility is seen, however, to be due to the presence of a trigram in a language model, as opposed to its actual probability. There is therefore room for improvement in the trigram model's ability to rank partial solutions that are not eliminated outright.

Our algorithm is well adapted to our problem in that it is guaranteed to converge to the optimal solution to a cipher given a language model, and will not get stuck in local optima. The algorithms presented by Peleg and Rosenfeld (1979) and Knight et al. (2006) do not give this guarantee, and so may require random restarts. The algorithms presented by Jakobsen (1995) and Gester (2003) can also get stuck in local optima. Our

algorithm demonstrates a fundamental theoretical improvement over these predecessors in this regard.

Moreover, although we assume knowledge of the location of the word endings in our experiments, we do not actually require it for our algorithm to function — decoding a cipher with no previous knowledge will naturally take longer, but the underlying search will only differ in its starting point. This gives us an advantage over the algorithms presented in Hart (1994) and Olson (2007), which require knowledge of the location of the word endings in order to function. This makes our algorithm applicable in a much wider range of circumstances than these predecessors.

The contribution of Ravi and Knight (2008) is, like our solution, guaranteed to converge to the model optimal solution without random restarts, but requires massive running times to do so. This indicates that the algorithm that they use is inefficient in this task. In addition, their algorithm does not readily break into parallel subprograms, and thus requires all of its work to be run on a single CPU. On the other hand, our improved algorithm has not only been shown to be very efficient in this task, but also to be highly parallelizable. In our experiments, our algorithms have been split between the many cores of a NVIDIA graphics card, and have shown very good running time complexity. As we have said in Chapter 5, we would have liked to compare our method empirically with the method presented in Ravi and Knight (2008), but their method was simply not feasible on our hardware for the size of ciphertext and plaintext alphabets that we have used.

Perhaps the most valuable insight gleaned from our study has been on the role of the language model. The algorithm's asymptotic runtime complexity is actually a function of entropic aspects of the character-level language model that it uses — more uniform models provide less prominent separations between candidate partial solutions, and this leads to badly ordered queues, in which extended partial solutions can never compete with partial solutions that have smaller domains, leading to a blind search. We believe that

there is a great deal of promise in characterizing natural language processing algorithms in this way, due to the prevalence of Bayesian methods that use language models as priors.

Our approach makes no explicit attempt to account for noisy ciphers, in which characters are erroneously mapped, nor any attempt to account for more general substitution ciphers in which a single plaintext (resp. ciphertext) letter can map to multiple ciphertext (resp. plaintext) letters, nor for ciphers in which ciphertext units correspond to larger units of plaintext such as syllables or words. All of these problems provide worthwhile avenues for future research.

Bibliography

Friedrich L. Bauer. *Decrypted Secrets.* Springer-Verlag, Berlin Heidelberg, 2007.

Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the Association of Computing Machinery*, 38(3):505–536, 1985.

Joe Gester. Solving substitution ciphers with genetics algorithm [sic]. Technical report, University of Rochester, 2003.

George W. Hart. To decode short cryptograms. *Communications of the ACM*, 37(9): 102–108, 1994.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Thomas Jakobsen. A fast method for the cryptanalysis of substitution ciphers. *Cryptologia*, 19(3):265–274, 1995.

Kevin Knight. Decoding complexity in word-replacement translation models. *Computational Linguistics*, 25(4):607–615, 1999.

Kevin Knight, Anish Nair, Nishit Rathod, and Kenji Yamada. Unsupervised analysis for decipherment problems. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 499–506, 2006.

George Nagy, Sharad Seth, and Kent Einspahr. Decoding substitution ciphers by means of word matching with application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(5):710–715, 1987.

Edwin Olson. Robust dictionary attack of short simple substitution ciphers. *Cryptologia*, 31(4):332–342, 2007.

Shmuel Peleg and Azriel Rosenfeld. Breaking substitution ciphers using a relaxation algorithm. *Communications of the ACM*, 22(11):589–605, 1979.

Sujith Ravi and Kevin Knight. Attacking decipherment problems optimally with low-order n-gram models. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics*, pages 812–819, 2008.

Benjamin Snyder, Regina Barzilay, and Kevin Knight. A statistical model for lost language decipherment. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1048–1057, 2010.

Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13(2):260–269, April 1967.