# An Exact A* Method for Deciphering Letter-Substitution Ciphers

**Eric Corlett** and **Gerald Penn**
Department of Computer Science
University of Toronto
{ecorlett,gpenn}@cs.toronto.edu

## Abstract

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. It 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 relation to OCR applications. In this paper, we introduce an exact method for deciphering messages using a generalization of the Viterbi algorithm. We test this model on a set of ciphers developed from various web sites, and find that our algorithm has the potential to be a viable, practical method for efficiently solving decipherment problems.

## 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. 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 other languages such as Hindi in which there is no dominant encoding standard for the writing system. 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 for 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. 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 will be effective in this domain.

It may seem at first that automatically decoding (as opposed to deciphering) a document is a simple matter, but studies have shown that simple algorithms such as letter frequency counting do not always produce optimal solutions (Bauer, 2007). If the text from which a language model is trained is of a different genre than the plaintext of a cipher, the unigraph letter frequencies may differ substantially from those of the language model, and so frequency counting will be misleading. Because of the perceived simplicity of the problem, however, little work was performed to understand its computational properties until Peleg and Rosenfeld (1979), who developed a method that repeatedly swaps letters in a cipher to find a maximum probability solution. Since then, several different approaches to this problem have been suggested, some of which use word counts in the language to arrive at a solution (Hart, 1994), and some of



*Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1040–1047,
Uppsala, Sweden, 11-16 July 2010. 

which treat the problem as an expectation maximization problem (Knight et al., 2006; Knight, 1999). These later algorithms are, however, 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), which, though published earlier, was inspired in part by the method presented here, first discovered in 2007. Unlike the present method, however, Ravi and Knight (2008) 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. In any case, an exact method is available with a much more efficient A* search that is linear-time in the length of the cipher (though still horribly exponential in the size of the cipher and plain text alphabets), and has the additional advantage of being massively parallelizable. (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 artefact 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 zeroes can quickly converge even on short ciphers of only a few characters. In the present method, the role of the language model can be acutely perceived; both the time complexity of the algorithm and the accuracy of the results depend crucially on this characteristic of the language model. In fact, we must use add-one smoothing to decipher texts of even modest lengths because even one unseen plain-text letter sequence is enough to knock out the correct solution. It is likely that the method of (Ravi and Knight, 2008) is sensitive to this as well, but their experiments were apparently fixed on a single, well-trained model.

Applications of decipherment are also explored by (Nagy et al., 1987), who uses it in the context of optical character recognition (OCR). The problem we consider here is cosmetically related to the "L2P" (letter-to-phoneme) mapping problem of text-to-speech synthesis, which also features a prominent constraint-based approach (van den Bosch and Canisius, 2006), 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.

## 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 a single character of an 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 $\Sigma_C$, and that its size is $n_C$. Similarly, the entire possible plaintext alphabet is $\Sigma_P$, and its size is 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 $c_{len}$. In the above example, $\Sigma_P$ is $\{\_, a, \ldots z\}$ and $n_P = 27$, while $\Sigma_C = \{\_, e, f, i, m, p, s, x\}$, $c_{len} = 11$ 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 $\sigma'$, we have that $\sigma \subset \sigma'$, then we say that $\sigma'$ *extends* $\sigma$. If the size of $\sigma'$ is $k+1$ and $\sigma$ is size $k$, we say that $\sigma'$ is an *immediate extension* of $\sigma$. 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 $\sigma_1$. 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 $\sigma_T$ extends $\sigma_1$ but not $\sigma_2$.

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. This plaintext can be found by treating all of the length $c_{len}$ strings $S$ as being the output of different character mappings from $C$. A string $S$ that results from such a mapping is *consistent* with a partial solution $\sigma$ 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 $c_{len} = 11$. 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 $\sigma = \{\_ : \_, n : e\}$.

## 3 The Algorithm

In order to efficiently search for the most likely solution for a ciphertext $C$, we conduct a search of the partial solutions using their trigram probabilities as a heuristic, where the trigram probability of a partial solution $\sigma$ of length $k$ is the maximum trigram probability over all strings consistent with it, meaning, in particular, that ciphertext letters not in its range can be mapped to any plaintext letter, and do not even need to be consistently mapped to the same plaintext letter in every instance. Given a partial solution $\sigma$ of length $n$, we can extend $\sigma$ by choosing a ciphertext letter $c$ not in the range of $\sigma$, and then use our generalization of the Viterbi algorithm to find, for each $p$ not in the domain of $\sigma$, a score to rank the choice of $p$ for $c$, namely the trigram probability of the extension $\sigma_p$ of $\sigma$. If we start with an empty solution and iteratively choose the most likely remaining partial solution in this way, storing the extensions obtained in a priority heap as we go, we will eventually reach a solution of size $n_C$. Every extension of $\sigma$ has a probability that is, at best, equal to that of $\sigma$, and every partial solution receives, at worst, a score equal to its best extension, because the score is potentially based on an inconsistent mapping that does not qualify as an extension. These two observations taken together mean that one minus the score assigned by our method constitutes a cost function over which this score is an admissible heuristic in the A* sense. Thus the first solution of size $n_C$ will be the best solution of size $n_C$.

The order by which we add the letters $c$ to partial solutions is the order of the distinct ciphertext letters in right-to-left order of their final occurrence in $C$. Other orderings for the $c$, such as most frequent first, are also possible though less elegant.[1]

**Algorithm 1** Search Algorithm

Order the letters $c_1 \ldots c_{n_C}$ by rightmost occurrence in $C$, $r_{n_C} < \ldots < r_1$.
Create a priority queue $Q$ for partial solutions, ordered by highest probability.
Push the empty solution $\sigma_0 = \{\}$ onto the queue.
**while** $Q$ is not empty **do**
  Pop the best partial solution $\sigma$ from $Q$.
  $s = |\sigma|$.
  **if** $s = n_C$ **then**
    return $\sigma$
  **else**
    For all $p$ not in the range of $\sigma$, push the immediate extension $\sigma_p$ onto $Q$ with the score assigned to table cell $G(r_{s+1}, p, p)$ by GVit$(\sigma, c_{s+1}, r_{s+1})$ if it is non-zero.
  **end if**
**end while**
Return "Solution Infeasible".

Our generalization of the Viterbi algorithm, depicted in Figure 1, 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 $C$. Unlike the real Viterbi algorithm, we must also observe the constraints of the input partial solution's mapping.

[1] We have experimented with the most frequent first regimen as well, and it performs worse than the one reported here. 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.

A typical decipherment involves multiple runs of this algorithm, each of which scores all of the immediate extensions, both tightening and lowering their scores relative to the score of the input partial solution. A call GVit$(\sigma, c, r)$ manages this by filling in a table $G$ such that for all $1 \leq i \leq r$, and $l, k \in \Sigma_P$, $G(i, l, k)$ is the maximum probability over every plaintext string $S$ for which:

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

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. Ours is larger, so so we will refer to $G$ as a *greenhouse*.

The table is completed by filling in the columns from $i = 1$ to $c_{len}$ in order. In every column $i$, we will iterate over the values of $l$ and over the values of $k$ such that $k : c$ and $l :$ are consistent with $\sigma$. 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. Backpointers are necessary to reference one of the two language model probabilities. 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.

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 considered / extended once.

GVit 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 to in order to implement the parallel sections of the code.

## 4 Experiment

The above algorithm is designed for application to the transliteration of electronic documents, specifically, the transliteration of websites, and it has been tested with this in mind. In order to gain realistic test data, we have operated on the assumption that Wikipedia is a good approximation of the type of language that will be found in most internet articles. 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 were used. A rough search over internet articles 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, does have entries that are one sentence in length. We have run two groups of tests for our algorithm. 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 appending 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.

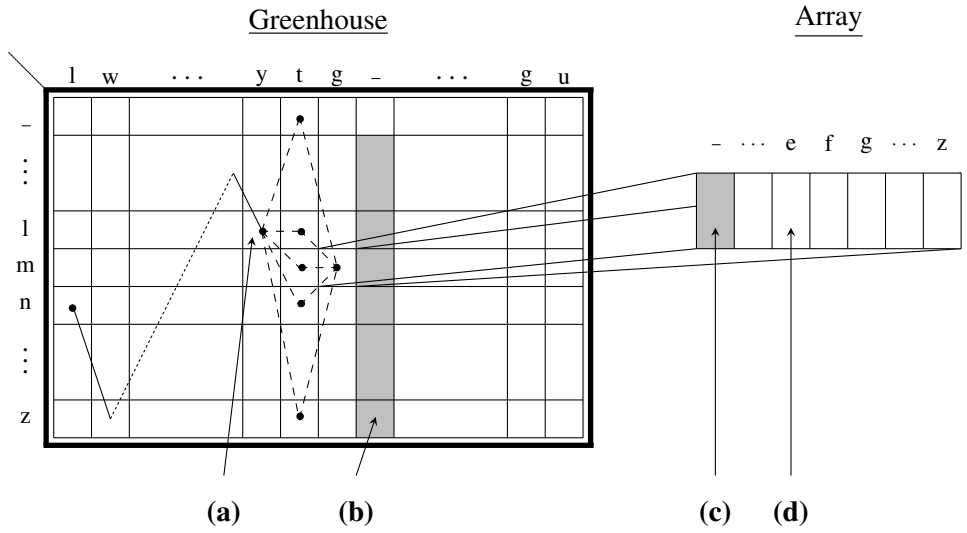


Figure 1: Filling the Greenhouse Table. Each cell in the greenhouse is indexed by a plaintext letter and a character from the cipher. Each cell consists of a smaller array. The cells in the array give the best probabilities of any path passing through the greenhouse cell, given that the index character 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).

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. Our algorithm will still work without this assumption, but would take longer. In the event that a trigram or bigram would be found in the plaintext that was not counted in the language model, add one smoothing was used.

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. As discussed in the previous paragraph, the space character is assumed to be known.

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 a low probability character in the ciphertext may be guessed wrong without changing as much of the actual plaintext. Counting the actual number of word errors is meant as an estimate of how useful or readable the plaintext will be. We did not count the accuracy or word error rate for unfinished ciphers.

We would have liked to compare our results with those of Ravi and Knight (2008), but the method presented there was simply not feasible

**Algorithm 2** Generalized Viterbi Algorithm

GVit($\sigma, c, r$)
Input: partial solution $\sigma$, ciphertext character $c$, and index $r$ into $C$.
Output: greenhouse $G$.
Initialize $G$ to 0.
i = 1
**for** All $(l, k)$ such that $\sigma \cup \{k : c, l : C_i\}$ is consistent **do**
  $G(i, l, k) = P(l)$.
**end for**
i = 2
**for** All $(l, k)$ such that $\sigma \cup \{k : c, l : C_i\}$ is consistent **do**
  **for** $j$ such that $\sigma \cup \{k : c, l : C_i, j : C_{i-1}\}$ is consistent **do**
    $G(i, l, k) = \max(G(i, l, k), G(0, j, k) \times P(l|j))$
  **end for**
**end for**
i = 3
**for** $(l, k)$ such that $\sigma \cup \{k : c, l : C_i\}$ is consistent **do**
  **for** $j_1, j_2$ such that $\sigma \cup \{k : c, j_2 : C[i-2], j_1 : C[i-1], l : C_i\}$ is consistent **do**
    $G(i, l, k) = \max(G(i, l, k), G(i-2, j_2, k) \times P(j_1|j_2) \times P(l|j_2 j_1))$.
  **end for**
**end for**
**for** $i = 4$ to $r$ **do**
  **for** $(l, k)$ such that $\sigma \cup \{k : c, l : C_i\}$ is consistent **do**
    **for** $j_1, j_2$ such that $\sigma \cup \{k : c, j_2 : C[i-2], j_1 : C[i-1], l : C_i\}$ is consistent **do**
      $G(i, l, k) = \max(G(i, l, k), G(i-2, j_2, k) \times P(j_1|j_2 j_{2(back)}) \times P(l|j_2 j_1))$.
    **end for**
  **end for**
**end for**

on texts and (case-sensitive) alphabets of this size with the computing hardware at our disposal.

## 5 Results

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. For this set of tests, in the event that a test took more than 12 hours, we terminated it and listed it as unfinished. This cutoff was set in advance of the runs based upon our armchair speculation about how long one might at most be reasonably expected to wait for a web-page to be transliterated (an overnight run). The results from this run appear in Table 1. All running times reported in this section were obtained on a computer running Ubuntu Linux 8.04 with 4 GB of RAM and $8 \times 2.5$ GHz CPU cores. Column-level subcomputations in the greenhouse were dispatched to an NVIDIA Quadro FX 1700 GPU card that is attached through a 16-lane PCI Express adapter. The card has 512 MB of cache memory, a 460 MHz core processor and 32 shader processors operating in parallel at 920 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 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 large size of the texts relative to the uniticity distance of an English letter-substitution cipher (Bauer, 2007). The results do show, however, that character trigram probabilities 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. 80% of our runs finished before the 12 hour cutoff in the first experiment.

| Cipher | Time | Enqueued | Expanded | Zeros | Accuracy | Word Error Rate |
|---|---|---|---|---|---|---|
| 1 | 2:03:06 | 964 | 964 | 44157 | 100% | 0% |
| 2 | 0:13:00 | 132 | 132 | 5197 | 100% | 0% |
| 3 | 0:05:42 | 91 | 91 | 3080 | 100% | 0% |
| 4 | Unfinished | N/A | N/A | N/A | N/A | N/A |
| 5 | Unfinished | N/A | N/A | N/A | N/A | N/A |
| 6 | 5:33:50 | 2521 | 2521 | 114283 | 100% | 0% |
| 7 | 6:02:41 | 2626 | 2626 | 116392 | 100% | 0% |
| 8 | 3:19:17 | 1483 | 1483 | 66070 | 100% | 0% |
| 9 | 9:22:54 | 4814 | 4814 | 215086 | 100% | 0% |
| 10 | 1:23:21 | 950 | 950 | 42107 | 100% | 0% |

Table 1: Time consumption and accuracy on a sample of 10 6000-character texts.

| Size | Time | Enqueued | Expanded | Zeros | Accuracy | Word Error Rate |
|---|---|---|---|---|---|---|
| 1000 | 40:06:05 | 119759 | 119755 | 5172631 | 92.59% | 1.89% |
| 3500 | 0:38:02 | 615 | 614 | 26865 | 96.30% | 0.17% |
| 6000 | 0:12:34 | 147 | 147 | 5709 | 100% | 0% |
| 8500 | 8:52:25 | 1302 | 1302 | 60978 | 100% | 0% |
| 11000 | 1:03:58 | 210 | 210 | 8868 | 100% | 0% |
| 13500 | 0:54:30 | 219 | 219 | 9277 | 100% | 0% |

Table 2: Time consumption and accuracy on prefixes of a single 13500-character ciphertext.

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, and that there are some that take longer than the threshold we gave in the experiment. Specifically, there is substantial variability in the 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 the second test set, there is a general decrease in both the running time and the number of solutions expanded as the length of the ciphers increases. Running time correlates very well with A* queue size.

Asymptotically, the time required for each sweep of the Viterbi algorithm increases, but this is more than offset by the decrease in the number of required sweeps.

The results, however, do not show that running time monotonically decreases with length. In particular, the length 8500 cipher generates more solutions than the length 3500 or 6000 ones. Recall that the ciphers in this section are all prefixes of the same string. Because 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.

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 really 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 very great difference to the running time of this method.

## 6 Conclusions

In the above paper, we have 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 our algorithm over ciphers drawn from Wikipedia, we found its accuracy to be 100% on the ciphers that it solved within a threshold of 12 hours, this being 80% of the total attempted. We found that the running time of our algorithm is highly variable depending on the order of characters attempted, and, due to the linear-time theoretical complexity of this method, that running times tend to decrease with larger ciphertexts due to our character-level language model's facility at eliminating highly improbable solutions. There is, however, a great deal of room for improvement in the trigram model's ability to rank partial solutions that are not eliminated outright.

Perhaps the most valuable insight gleaned from this study has been on the role of the language model. This 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 corresponds to larger units of plaintext such syllables or words. Extensions in these directions are all very worthwhile to explore.

## References


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

George W. Hart. 1994. To Decode Short Cryptograms. *Communications of the ACM*, 37(9): 102–108.

Kevin Knight. 1999. Decoding Complexity in Word-Replacement Translation Models. *Computational Linguistics*, 25(4):607–615.

Kevin Knight, Anish Nair, Nishit Rathod, Kenji Yamada. Unsupervised Analysis for Decipherment Problems. *Proceedings of the COLING/ACL 2006*, 2006, 499–506.

George Nagy, Sharad Seth, Kent Einspahr. 1987. Decoding Substitution Ciphers by Means of Word Matching with Application to OCR. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(5):710–715.

Shmuel Peleg and Azriel Rosenfeld. 1979. Breaking Substitution Ciphers Using a Relaxation Algorithm. *Communications of the ACM*, 22(11):589–605.

Sujith Ravi, Kevin Knight. 2008. Attacking Decipherment Problems Optimally with Low-Order N-gram Models *Proceedings of the ACL 2008*, 812–819.

Antal van den Bosch, Sander Canisius. 2006. Improved Morpho-phonological Sequence Processing with Constraint Satisfaction Inference *Proceedings of the Eighth Meeting of the ACL Special Interest Group on Computational Phonology at HLT-NAACL 2006*, 41–49.