# Why Letter Substitution Puzzles are Not Hard to Solve: A Case Study in Entropy and Probabilistic Search-Complexity

**Eric Corlett**
University of Toronto
10 King's College Rd., Room 3302
Toronto, ON, Canada M5S 3G4
ecorlett@cs.toronto.edu

**Gerald Penn**
University of Toronto
10 King's College Rd., Room 3302
Toronto, ON, Canada M5S 3G4
gpenn@cs.toronto.edu

## Abstract

In this paper we investigate the theoretical causes of the disparity between the theoretical and practical running times for the $A^*$ algorithm proposed in Corlett and Penn (2010) for deciphering letter-substitution ciphers. We argue that the difference seen is due to the relatively low entropies of the probability distributions of character transitions seen in natural language, and we develop a principled way of incorporating entropy into our complexity analysis. Specifically, we find that the low entropy of natural languages can allow us, with high probability, to bound the depth of the heuristic values expanded in the search. This leads to a novel probabilistic bound on search depth in these tasks.

## 1 Introduction

When working in NLP, we can find ourselves using algorithms whose worst-case running time bounds do not accurately describe their empirically determined running times. Specifically, we can often find that the algorithms that we are using can be made to run efficiently on real-world instances of their problems despite having theoretically high running times. Thus, we have an apparent disparity between the theoretical and practical running times of these algorithms, and so we must ask why these algorithms can provide results in a reasonable time frame. We must also ask to what extent we can expect our algorithms to remain practical as we change the downstream domains from which we draw problem instances.

At a high level, the reason such algorithms can work well in the real world is that the real world applications from which we draw our inputs do not tend to include the high complexity inputs. In other words, our problem space either does not cover all possible inputs to the algorithm, or it does, but with a probability distribution that gives a vanishingly small likelihood to the "hard" inputs. Thus, it would be beneficial to incorporate into our running time analysis the fact that our possible inputs are restricted, even if only restricted in relative frequency rather than in absolute terms.

This means that any running time that we observe must be considered to be dependent on the distribution of inputs that we expect to sample from. It probably does not come as a surprise that any empirical analysis of running time carries with it the assumption that the data on which the tests were run are typical of the data which we expect to see in practice. Yet the received wisdom on the asymptotic complexity of algorithms in computational linguistics (generally what one might see in an advanced undergraduate algorithms curriculum) has been content to consider input only in terms of its size or length, and not the distribution from which it was sampled. Indeed, many algorithms in NLP actually take entire distributions as input, such as language models. Without a more mature theoretical understanding of time complexity, it is not clear exactly what any empirical running time results would mean. A worst-case complexity result gives a guarantee that an algorithm will take no more than a certain number of steps to complete. An average-case result gives the expected number of steps to complete. But an empirical running time found by sampling from a distribution that is potentially different from what the algorithm was designed for is only a lesson in how truly different the distribution is.

It is also common for the theoretical study of asymptotic time complexity in NLP to focus on the worst-case complexity of a problem or algorithm rather than an expected complexity, in spite of the existence for now over 20 years of methods for average-case analysis of an algorithm. Even these, however, often assume a uniform distribution over input, when in fact the true expectation must consider the probability distribution that we will draw the inputs from. Uniform distributions are only common because we may not know what the distribution is beforehand.

Ideally, we should want to characterize the running time of an algorithm using some known properties of its input distribution, even if the precise distribution is not known. Previous work that attempts this does exist. In particular, there is a variant of analysis referred to as *smoothed analysis* which gives a bound on the average-case running time of an algorithm under the assumption that all inputs are sampled with Gaussian measurement error. As we will argue in Section 2, however, this approach is of limited use to us.

We instead approach the disparity of theoretical and practical running time by making use of statistics such as entropy, which are taken from the input probability distributions, as eligible factors in our analysis of the running time complexity. This is a reasonable approach to the problem, in view of the numerous entropic studies of word and character distributions dating back to Shannon.

Specifically, we analyze the running time of the $A^*$ search algorithm described in Corlett and Penn (2010). This algorithm deciphers text that has been enciphered using a consistent letter substitution, and its running time is linear in the length of the text being deciphered, but theoretically exponential in the size of the input and output alphabets. This naïve theoretical analysis assumes that characters are uniformly distributed, however. A far more informative bound is attainable by making reference to the entropy of the input. Because the algorithm takes a language model as one of its inputs (the algorithm is guaranteed to find the model-optimal letter substitution over a given text), there are actually two input distributions: the distribution assumed by the input language model, and the distribution from which the text to be deciphered was sampled. Another way to view this problem is as a search for a permutation of letters as the outcomes of one distribution such that the two distributions are maximally similar. So our informative bound is attained through reference to the cross-entropy of these two distributions.

We first formalize our innate assumption that these two distributions are similar, and build an upper bound for the algorithm's complexity that incorporates the cross-entropy between the two distributions. The analysis concludes that, rather than being exponential in the length of the input or in the size of the alphabets, it is merely exponential in the cross-entropy of these two distributions, thus exposing the importance of their similarity. Essentially, our bound acts as a probability distribution over the necessary search depth.

## 2 Related Work

The closest previous work to the analysis presented here is the use of smoothed analysis to explain the tractable real-world running time of a number of algorithms with an exponential worst-case complexity. These algorithms include the simplex algorithm, as described by Spielman and Teng (2004), the k-means clustering algorithm, as described by Arthur et al. (2009) and others. As in our current approach, smoothed analysis works by running a general average-case analysis of the algorithms without direct knowledge of the distribution from which the problem inputs have been drawn. The assumption made in smoothed analysis is that every input has been read with some Gaussian measurement error. That is, in a typical worst-case analysis, we may have an adversary choose any input for our algorithm, after which we must calculate how bad the resulting running time might be, but in a smoothed analysis, the adversary gives us input by placing it into the real world so that we may measure it, and this measurement adds a small error drawn from a Gaussian distribution to the problem instance. The point of smoothed analysis is to find the worst average-case running time, under these conditions, that the adversary can subject us to. Thus the analysis is an average case, subject to this error, of worst cases. In the papers cited above, this method of analysis was able to drop running times from exponential to polynomial.

It is unfortunate that this approach does not readily apply to many of the algorithms that we use in NLP. To see why this is, simply note that we can only add a small Gaussian error to our inputs if our inputs themselves are numerical. If the inputs to our algorithms are discrete, say, in the form of strings, then Gaussian errors are not meaningful. Rather, we must ask what sort of error we can expect to see in our inputs, and to what extent these errors contribute to the running time of our algorithms. In the case of decipherment, "error" is committed by substituting one character for another consistently.

The strongest known result on the search complexity of $A^*$ is given in Pearl (1984). This work found that, under certain assumptions, a bound on the absolute error between the heuristic used and the true best cost to reach the goal yields a polynomial worst-case depth for the search. This happens when the bound is constant across search instances of different sizes. On the other hand, if the relative error does not have this constant bound, the search complexity can still be exponential. This analysis assumes that the relative errors in the heuristic are independent between nodes of the search tree. It is also often very difficult even to calculate the value of a heuristic that possesses such a bound, as it might involve calculating the true best cost, which can be as difficult as completely solving a search problem instance (Korf et al., 2001). Thus, most practical heuristics still give rise to theoretically exponential search complexities in this view.

In Korf and Reid (1998) and Korf et al. (2001), on the other hand, several practical problems are treated, such as random $k$-SAT, Rubik's cubes, or sliding tile puzzles, which are not wholly unlike deciphering letter substitution puzzles in that they calculate permutations, and therefore can assume, as we do, that overall time complexity directly corresponds to the number of nodes visited at different depths in the search tree that have a heuristic low enough to guarantee node expansion. But their analysis assumes that it is possible to both estimate and use a probability distribution of heuristic values on different nodes of the search graph, whereas in our task, this distribution is very difficult to sample because almost every node in the search graph has a worse heuristic score than the goal does, and would therefore never be expanded. Without an accurate idea of what the distribution of the heuristic is, we cannot accurately estimate the complexity of the algorithm. On the other hand, their analysis makes no use of any estimates of the cost of reaching the goal, because the practical problems that they consider do not allow for particularly accurate estimates. In our treatment, we find that the cost to reach the goal can be estimated with high probability, and that this estimate is much less than the cost of most nodes in the search graph. These different characteristics allow us to formulate a different sort of bound on the search complexity for the decipherment problem.

## 3 The Algorithm

We now turn to the algorithm given in Corlett and Penn (2010) which we will investigate, and we explain the model we use to find our bound.

The purpose of the algorithm is to allow us to read a given ciphertext $C$ which is assumed to be generated by putting an unknown plaintext $P$ through an unknown monoalphabetic cipher.

We will denote the ciphertext alphabet as $\Sigma_c$ and the plaintext alphabet as $\Sigma_p$. Given any string $T$, we will denote $n(T)$ as the length of $T$. Furthermore, we assume that the plaintext $P$ is drawn from some string distribution $q$. We do not assume $q$ to be a trigram distribution, but we do require it to be a distribution from which trigrams can be calculated (e.g, a 5-gram corpus will in general have probabilities that cannot be predicted using the associated trigrams, but the associated trigram corpus can be recovered from the 5-grams).

It is important to realize in the algorithm description and analysis that $q$ may also not be known exactly, but we only assume that it exists, and that we can approximate it with a known trigram distribution $p$. In Corlett and Penn (2010), for example, $p$ is the trigram distribution found using the Penn treebank. It is assumed that this is a good approximation for the distribution $q$, which in Corlett and Penn (2010) is the text in Wikipedia from which ciphers are drawn. As is common when dealing with probability distributions over natural languages, we assume that both $p$ and $q$ are stationary and ergodic, and we furthermore assume that $p$ is smooth enough that any trigram that can be found in any string generated by $q$ occurs in $p$ (i.e., we assume that the cross entropy $H(p, q)$ is finite).

The algorithm works in a model in which, for any run of the algorithm, the plaintext string $P$ is drawn according to the distribution $q$. We do not directly observe $P$, but instead its encoding using the cipher key, which we will call $\pi_T$. We observe the ciphertext $C = \pi_T^{-1}(P)$. We note that $\pi_T$ is unknown, but that it does not change as new ciphertexts are drawn.

Now, the way that the algorithm in Corlett and Penn (2010) works is by searching over the possible keys to the cipher to find the one that maximizes the probability of the plaintext according to the distribution $p$. It does so as follows.

In addition to the possible keys to the cipher, weakened cipher keys called *partial solutions* are added to the search space. A partial solution of size $k$ (denoted as $\pi_k$) is a section of a possible full cipher key which is only defined on $k$ character types in the cipher. We consider the character types to be fixed according to some preset order, and so the $k$ fixed letters in $\pi_k$ do not change between different partial solutions of size $k$.

Given a partial solution $\pi_k$, a string $\pi_k^{n(C)}(C)$ is defined whose probability we use as an upper bound for the probability of the plaintext whenever the true solution to the cipher contains $\pi_k$ as a subset. The string $\pi_k^{n(C)}(C)$ is the most likely string that we can find that is consistent with $C$ on the letters fixed by $\pi_k$. That is, we define the set $\Pi_k$ so that $S \in \Pi_k$ iff whenever $s_i$ and $c_i$ are the characters at index $i$ in $S$ and $C$, then $s_i = \pi_k(c_i)$ if $c_i$ is fixed in $\pi_k$. Note that if $c_k$ is not fixed in $\pi_k$, we let $s_i$ take any value. We extend the partial character function to the full string function $\pi_k^{n(C)}$ on $\Sigma_c^{n(C)}$ so that $\pi_k^{n(C)}(C) = argmax_{(S\in\Pi_k)} prob_p(S)$.

In Corlett and Penn (2010), the value $\pi_k^{n(C)}(C)$ is efficiently computed by running it through the Viterbi algorithm. That is, given $C$, $p$ and $\pi_k$, a run of the Viterbi algorithm is set up in which the letter transition probabilities are those that are given in $p$. In order to describe the emission probabilities, suppose that we partition the ciphertext alphabet $\Sigma_c$ into two sets $\Sigma_1$ and $\Sigma_2$, where $\Sigma_1$ is the set of ciphertext letters fixed by $\pi_k$. For any plaintext letter $y \in \Sigma_p$, if there is a ciphertext letter $x \in \Sigma_1$ such that $y \to x$ is a rule in $\pi_k$, then the emission probability that $y$ will be seen as $x$ is set to 1, and the probability that $y$ will be seen as any other letter is set to 0. On the other hand, if there is no rule $y \to x$ in $\pi_k$ for any ciphertext letter $x$, then the emission probability associated with $y$ is uniform over the letters $x \in \Sigma_2$ and 0 for the letters $x \in \Sigma_1$.

The search algorithm described in Corlett and Penn (2010) uses the probability of the string $\pi_k^{n(C)}(C)$, or more precisely, the log probability $-logprob_p(\pi_k^{n(C)}(C))$, as an $A^*$ heuristic over the partial solutions $\pi_k$. In this search, an edge is added from a size $k$ partial solution $\pi_k$ to a size $k+1$ partial solution $\pi_{k+1}$ if $\pi_k$ agrees with $\pi_{k+1}$ wherever it is defined. The score of a node $\pi_k$ is the log probability of its associated string: $-logprob_p(\pi_k^{n(C)}(C))$. We can see that if $\pi_k$ has an edge leading to $\pi_{k+1}$, then $\Pi_{k+1} \subset \Pi_k$, so that $-logprob_p(\pi_{k+1}^{n(C)}(C)) \geq -logprob_p(\pi_k^{n(C)}(C))$. Thus, the heuristic is nondecreasing. Moreover, by applying the same statement inductively we can see that any full solution to the cipher that has $\pi_k$ as a subset must have a score at least as great as that of $\pi_k$. This means that the score never overestimates the cost of completing a solution, and therefore that the heuristic is admissible.

## 4 Analysis

The bound that we will prove is that for any $k > 0$ and for any $\delta, \varepsilon > 0$, there exists an $n \in \mathbb{N}$ such that if the length $n(C)$ of the cipher $C$ is at least $n$, then with probability at least $1-\delta$, the search for the key to the cipher $C$ requires no more than $2^{n\cdot(H(p,q)+\varepsilon)}$ expansions of any partial solution of size $k$ to complete. Applying the same bound over every size $k$ of partial solution will then give us that for any $\delta, \varepsilon > 0$, there exists a $n_0 > 0$ such that if the length $n(C)$ of the cipher $C$ is at least $n$, then with probability at least $1-\delta$, the search for the key to the cipher $C$ requires no more than $2^{n(H(p,q)+\varepsilon)}$ expansions of any partial solution of size greater than 0 to complete (note that there is only one partial solution of size 0).

Let $\pi^*$ be the solution that is found by the search. This solution has the property that it is the full solution that induces the most probable plaintext from the cipher, and so it produces a plaintext that is at least as likely as that of the true solution $P$. Thus, we have that $-logprob_p(\pi^{*n(C)}(C)) \leq -logprob_p(\pi_T^{n(C)}(C)) = -logprob_p(P)$.

We find our bound by making use of the fact that an $A^*$ search never expands a node whose score is greater than that of the goal node $\pi^*$. Thus, a partial solution $\pi_k$ is expanded only if

$$-logprob_p(\pi_k^{n(C)}(C)) \leq -logprob_p(\pi^{*n(C)}(C)).$$

Since

$$-logprob_p(\pi^{*n(C)}(C)) \leq -logprob_p(P),$$

we have that $\pi_k$ is expanded only if

$$-logprob_p(\pi_k^{n(C)}(C)) \leq -logprob_p(P).$$

So we would like to count the number of solutions satisfying this inequality.

We would first like to approximate the value of $-logprob_p(P)$, then. But, since $P$ is drawn from an ergodic stationary distribution $q$, this value will approach the cross entropy $H(p,q)$ with high probability: for any $\delta_1, \varepsilon_1 > 0$, there exists an $n_1 > 0$ such that if $n(C) = n(P) > N_1$, then

$$|-logprob_p(P)/n(C) - H(p,q)| < \varepsilon_1$$

with probability at least $1 - \delta_1$. In this case, we have that $-logprob_p(P) < n(C)(H(p,q) + \varepsilon_1)$.

Now, if $k$ is fixed, and if $\pi_k$ and $\pi'_k$ are two different size $k$ partial solutions, then $\pi_k$ and $\pi'_k$ must disagree on at least one letter assignment. Thus, the sets $\Pi_k$ and $\Pi'_k$ must be disjoint. But then we also have that $\pi_k^{n(C)}(C) \neq \pi_k^{n(C)\prime}(C)$. Therefore, if we can find an upper bound for the size of the set

$$\{S \in \Sigma_p^{n(C)} | S = \pi_k^{n(C)}(C) \text{ for some } \pi_k\},$$

we will have an upper bound on the number of times the search will expand any partial solution of size $k$. We note that under the previous assumptions, and with probability at least $1 - \delta_1$, none of these strings can have a log probability larger than $n(C)(H(p,q) + \varepsilon_1)$.

For any plaintext string $C$ drawn from $q$, we let ${}_aP_b$ be the substring of $P$ between the indices $a$ and $b$. Similarly, we let ${}_aS_b$ be the substring of $S = \pi_k^{n(C)}(C)$ between the indices $a$ and $b$.

We now turn to the proof of our bound: Let $\delta, \varepsilon > 0$ be given. We give the following three bounds on $n$:

(a) As stated above, we can choose $n_1$ so that for any string $P$ drawn from $q$ with length at least $n_1$,

$$|-logprob_p(P)/n(P) - H(p,q)| < \varepsilon_1/2$$

with probability at least $1 - \delta/3$.

(b) We have noted that if $k$ is fixed then any two size $k$ partial solutions must disagree on at least one of the letters that they fix. So if we have a substring ${}_aP_b$ of $P$ with an instance of every letter type fixed by the partial solutions of size $k$, then the substrings ${}_aS_b$ of $S$ must be distinct for every $S \in \{S \in \Sigma_p^{n(C)} | S = \pi_k^{n(C)}(C)$ for some $\pi_k\}$. Since $q$ is ergodic, we can find an $n_2$ such that for any string $P$ drawn from $q$ with length at least $n_2$, every letter fixed in $\pi_k$ can be found in some length $n_2$ substring $P_2$ of $P$, with probability at least $1 - \delta/3$.

(c) By the Lemma below, there exists an $n' > 0$ such that for all partial solutions $\pi_k$, there exists a trigram distribution $r_k$ on the alphabet $\Sigma_p$ such that if $S = \pi_k^{n(C)}(C)$ and $b - a = n > n'$, then

$$\left|\frac{-logprob({}_aS_b)}{n} - H(p, r_k)\right| < \varepsilon/4$$

with a probability of at least $1 - \delta/3$.

Let $n = \max(n_1, n_2, n')$. Then, the probability of any single one of the properties in (a), (b) or (c) failing in a string of length at least $n$ is at most $\delta/3$, and so the probability of any of them failing is at most $\delta$. Thus, with a probability of at least $1-\delta$, all three of the properties hold for any string $P$ drawn from $q$ with length at least $n$. Let $P$ be drawn from $q$, and suppose $n(P) > n$. Let ${}_aP_b$ be a length $n$ substring of $P$ containing a token of every letter type fixed by the size $k$ partial solutions.

Suppose that $\pi_k$ is a partial solution such that $-logprob_p(\pi_k^{n(C)}(C)) \leq n(P)(H(p,q) + \varepsilon/2)$. Then, letting $S = \pi_k^{n(C)}(C)$, we have that if

$$\left|\frac{-logprob(S)}{n(P)} - H(p, r_k)\right| < \varepsilon/4$$

and

$$\left|\frac{-logprob({}_aS_b)}{n} - H(p, r_k)\right| < \varepsilon/4$$

it follows that

$$\begin{aligned} &\left|\frac{-logprob(S)}{n(P)} + \frac{logprob({}_aS_b)}{n}\right| \\ &\leq \left|\frac{-logprob(S)}{n(P)} - H(p, r_k)\right| \\ &+ \left|-H(p, r_k) - \frac{logprob({}_aS_b)}{n}\right| \\ &\leq \varepsilon/4 + \varepsilon/4 = \varepsilon/2 \end{aligned}$$

But then,

$$\begin{aligned} &-\frac{logprob({}_aS_b)}{n} < \frac{-logprob(S)}{n(P)} + \varepsilon/2 \\ &\leq \frac{n(P)(H(p,q) + \varepsilon/2)}{n(P)} + \varepsilon/2 \\ &= H(p,q) + \varepsilon. \end{aligned}$$

So, for our bound we will simply need to find the number of substrings ${}_aS_b$ such that

$$-\log prob_p({}_aS_b) < n(H(p,q)+\varepsilon).$$

Letting $I_H({}_aS_b) = 1$ if $-logprob_p({}_aS_b) < n(H(p,q)+\varepsilon)$ and 0 otherwise, the number of strings we need becomes

$$\begin{aligned}\sum_{{}_aS_b\in\Sigma_p^{n(C)}} I_H({}_aS_b) &= 2^{n\cdot(H(p,q)+\varepsilon)}\sum_{{}_aS_b\in\Sigma_p^{n(C)}} I_H({}_aS_b)2^{-n\cdot(H(p,q)+\varepsilon)}\\ &< 2^{n\cdot(H(p,q)+\varepsilon)}\sum_{{}_aS_b\in\Sigma_p^{n(C)}} I_H({}_aS_b)prob_p({}_aS_b)\\ &\text{(since } -\log prob_p({}_aS_b) < n(H(p,q)+\varepsilon)\\ &\text{implies } prob_p({}_aS_b) > 2^{-n\cdot(H(p,q)+\varepsilon)})\\ &\le 2^{n\cdot(H(p,q)+\varepsilon)}\sum_{{}_aS_b\in\Sigma_p^{n(C)}} prob_p({}_aS_b)\\ &= 2^{n\cdot(H(p,q)+\varepsilon)}\end{aligned}$$

Thus, we have a bound of $2^{n\cdot(H(p,q)+\varepsilon)}$ on the number of substrings of length $n$ satisfying $-\log prob_p({}_aS_b) < n(H(p,q)+\varepsilon)$. Since we know that with probability at least $1-\delta$, these are the only strings that need be considered, we have proven our bound. $\square$

## 4.1 Lemma:

We now show that for any fixed $k > 0$ and $\delta', \varepsilon' > 0$, there exists some $n' > 0$ such that for all partial solutions $\pi_k$, there exists a trigram distribution $r_k$ on the alphabet $\Sigma_p$ such that if $S = \pi_k^{n(C)}(C)$ and $b-a = n > n'$, $|\frac{-logprob({}_aS_b)}{n} - H(p,r_k)| < \varepsilon'$ with a probability of at least $1-\delta'$.

**Proof of Lemma:** Given any partial solution $\pi_k$, it will be useful in this section to consider the strings $S = \pi_k^{n(C)}(C)$ as functions of the plaintext $P$ rather than the ciphertext $C$. Since $C = \pi_T^{-1}(P)$, then, we will compose $\pi_k^{n(C)}$ and $\pi_T^{-1}$ to get $\pi_k^{n(C)\prime}(P) = \pi_k^{n(C)}(\pi_T^{-1}(P))$. Now, since $\pi_T$ is derived from a character bijection between $\Sigma_c$ and $\Sigma_p$, and since $\pi_k^{n(C)}$ fixes the $k$ character types in $\Sigma_c$ that are defined in $\pi_k$, we have that $\pi_k^{n(C)\prime}$ fixes $k$ character types in $\Sigma_p$. Let $\Sigma_{P_1}$ be the set of $k$ character types in $\Sigma_p$ that are fixed by $\pi_k^{n(C)\prime}$, and let $\Sigma_{P_2} = \Sigma_p \setminus \Sigma_{P_1}$. We note that $\Sigma_{P_1}$ and $\Sigma_{P_2}$ do not depend on which $\pi_k$ we use, but only on $k$.

Now, any string $P$ which is drawn from $q$ can be decomposed into overlapping substrings by splitting it whenever it has see two adjacent characters from $\Sigma_{P_1}$. When we see a bigram in $P$ of this form, say, $y_1y_2$, we split $P$ so that both the end of the initial string and the beginning of the new string are $y_1y_2$. Note that when we have more than two adjacent characters from $\Sigma_{P_1}$ we will split the string more than once, so that we get a series of three-character substrings of $P$ in our decomposition. As a matter of bookkeeping we will consider the initial segment to begin with two start characters **s** with indices corresponding to 0 and $-1$ in $P$. As an example, consider the string

$P$ = *friends, romans, countrymen, lend me your ears*

Where $\Sigma_{P_1} = \{`\ ', `,', `a', `y'\}$. In this case, we would decompose $P$ into the strings *'**ss**friends, ', ', romans, ', ', countrymen, ', ', lend me ', 'e y', ' your e'* and *' ears'*.

Let $M$ be the set of all substrings that can be generated in this way by decomposing strings $P$ which are drawn from $q$. Since the end of any string $m \in M$ contains two adjacent characters in $\Sigma_{P_1}$ and since the presence of two adjacent characters in $\Sigma_{P_1}$ signals a position at which a string will be decomposed into segments, we have that the set $M$ is prefix-free. Every string $m \in M$ is a string in $\Sigma_p$, and so they will have probabilities $prob_q(m)$ in $q$. It should be noted that for any $m \in M$ the probability $prob_q(m)$ may be different from the trigram probabilities predicted by $q$, but will instead be the overall probability in $q$ of seeing the string $m$.

For any pair $T, P$ of strings, let $\#(T,P)$ be the number of times $T$ occurs in $P$. Since we assume that the strings drawn from $q$ converge to the distribution $q$, we have that for any $\delta_3, \varepsilon_3 > 0$ and any $n_4 > 0$, there exists an $n_3 > 0$ such that for any substring $P_3$ of $P$ of length at least $n_3$, where $P$ is drawn from $q$, and for any $m \in M$ of length at most $n_4$, the number $|\#(m,P)/len(P_3) - prob_q(m)| < \varepsilon_3$ with probability greater than $1-\delta_3$.

Now suppose that for some $P$ drawn from $q$ we have a substring ${}_aP_b$ of $P$ such that ${}_aP_b = m, m \in M$. If $S = \pi_k^{n(C)\prime}(P)$, consider the substring ${}_aS_b$ of $S$. Recall that the string function $\pi_k^{n(C)\prime}$ can map the characters in $P$ to $S$ in one of two ways: if a character $x_i \in \Sigma_{P_1}$ is found at index $i$ in $P$, then the corresponding character in $S$

is $\pi_k(x_i)$. Otherwise, $x_i$ is mapped to whichever character $y_i$ in $\Sigma_P$ maximizes the probability in $p$ of $S$ given $\pi_k^{n(C)\prime}(x_{i-2})\pi_k^{n(C)\prime}(x_{i-1})y_i$. Since the values of $\pi_k^{n(C)\prime}(x_{i-2}), \pi_k^{n(C)\prime}(x_{i-1})$ and $y_i$ are interdependent, and since $\pi_k^{n(C)\prime}(x_{i-2})$ is dependent on its previous two neighbors, the value that $y_i$ takes may be dependent on the values taken by $\pi_k^{n(C)\prime}(x_j)$ for indices $j$ quite far from $i$. However, we see that no dependencies can cross over a substring in $P$ containing two adjacent characters in $\Sigma_{P_1}$, since these characters are not transformed by $\pi_k^{n(C)\prime}$ in a way that depends on their neighbors. Thus, if ${}_aP_b = m \in M$, the endpoints of ${}_aP_b$ are made up of two adjacent characters in $\Sigma_{P_1}$, and so the substring ${}_aS_b$ of $S$ depends only on the substring ${}_aP_b$ of $P$. Specifically, we see that ${}_aS_b = \pi_k^{n(C)\prime}({}_aP_b)$.

Since we can decompose any $P$ into overlapping substrings $m_1, m_2, \ldots, m_t$ in $M$, then, we can carry over this decomposition into $S$ to break $S$ into $\pi_k^{n(C)\prime}(m_1), \pi_k^{n(C)\prime}(m_2), \ldots, \pi_k^{n(C)\prime}(m_t)$. Note that the score generated by $S$ in the $A^*$ search algorithm is the sum $\sum_{1\le i\le} logprob_p(y_{i-2}y_{i-1}y_i)$, where $y_i$ is the $i^{th}$ character in $S$. Also note that every three-character sequence $y_{i-2}y_{i-1}y_i$ occurs exactly once in the decomposition $\pi_k^{n(C)\prime}(m_1), \pi_k^{n(C)\prime}(m_2), \ldots, \pi_k^{n(C)\prime}(m_t)$. Since for any $m$ the number of occurrences of $\pi_k^{n(C)\prime}(m)$ in $S$ under this decomposition will be equal to the number of occurrences of $m$ in $P$, we have that

$$\begin{aligned} -logprob_p(S) &= \sum_{1\le i\le n(P)} logprob_p(y_{i-2}y_{i-1}y_i) \\ &= \sum_{m\in M} \#(m,P)\cdot(-logprob_p(\pi_k^{n(C)\prime}(m))). \end{aligned}$$

Having finished these definitions, we can now define the distribution $r_k$. In principle, this distribution should be the limit of the frequency of trigram counts of the strings $S = \pi_k^{n(C)\prime}(P)$, where $n(P)$ approaches infinity. Given a string $S = \pi_k^{n(C)\prime}(P)$, where $P$ is drawn from $q$, and given any trigram $y_1y_2y_3$ of characters in $\Sigma_p$, this frequency count is $\frac{\#(y_1y_2y_3,S)}{n(P)}$. Breaking $S$ into its component substrings $\pi_k^{n(C)\prime}(m_1), \pi_k^{n(C)\prime}(m_2), \ldots, \pi_k^{n(C)\prime}(m_t)$, as we have done above, we see that any instance of the trigram $y_1y_2y_3$ in $S$ occurs in exactly one of the substrings $\pi_k^{n(C)\prime}(m_i), 1 \le i \le t$. Grouping together similar $m_i$s, we find

$$\begin{aligned} \frac{\#(y_1y_2y_3,S)}{n(P)} &= \frac{\sum_{i=1}^{t} \#(y_1y_2y_3, \pi_k^{n(C)\prime}(m_i))}{n(P)} \\ &= \frac{\sum_{m\in M} \#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))\cdot\#(m,P)}{n(P)} \end{aligned}$$

As $n(P)$ approaches infinity, we find that $\frac{\#(m,P)}{n(P)}$ approaches $prob_q(m)$, and so we can write

$$prob_{r_k}(y_1y_2y_3) = \sum_{m\in M} \#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))prob_q(m).$$

Since $0 \le \sum_{m\in M} \#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))prob_q(m)$ when $P$ is sampled from $q$ we have that

$$\begin{aligned} &\sum_{y_1y_2y_3} prob_{r_k}(y_1y_2y_3) \\ &= \sum_{y_1y_2y_3}\sum_{m\in M}\#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))prob_q(m) \\ &= \lim_{n(P)\to\infty}\sum_{y_1y_2y_3}\sum_{m\in M}\#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))\frac{\#(m,P)}{n(P)} \\ &= \lim_{n(P)\to\infty}\sum_{m\in M}\sum_{y_1y_2y_3}\#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))\frac{\#(m,P)}{n(P)} \\ &= \lim_{n(P)\to\infty}\sum_{m\in M}\frac{(n(\pi_k^{n(C)\prime}(m))-2)\#(m,P)}{n(P)} \\ &= \lim_{n(P)\to\infty}\sum_{m\in M}\frac{(n(m)-2)\#(m,P)}{n(P)} \\ &= \lim_{n(P)\to\infty}\frac{n(P)}{n(P)} = 1, \end{aligned}$$

so we have that $prob_{r_k}$ is a valid probability distribution. In the above calculation we can rearrange the terms, so convergence implies absolute convergence. The sum $\sum_{y_1y_2y_3} \#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))$ gives $(n(\pi_k^{n(C)\prime}(m)) - 2)$ because there is one trigram for every character in $\pi_k^{n(C)\prime}(m)$, less two to compensate for the endpoints. However, since the different $m$ overlap by two in a decomposition from $P$, the sum $(n(m) - 2)\#(m,P)$ just gives back the length $n(P)$, allowing for the fact that the initial $m$ has two extra start characters.

Having defined $r_k$, we can now find the value of $H(p, r_k)$. By definition, this term will be

$$\sum_{y_1y_2y_3} -logprob_p(y_1y_2y_3)prob_{r_k}(y_1y_2y_3)$$
$$= \sum_{y_1y_2y_3} -logprob_p(y_1y_2y_3)\sum_{m\in M}\#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))prob_q(m)$$
$$= \sum_{m\in M}\sum_{y_1y_2y_3} -logprob_p(y_1y_2y_3)\#(y_1y_2y_3, \pi_k^{n(C)\prime}(m))prob_q(m)$$
$$= \sum_{m\in M} -logprob_p(m)prob_q(m).$$

Now, we can finish the proof of the Lemma. Holding $k$ fixed, let $\delta', \varepsilon' > 0$ be given. Since we have assumed that $p$ does not assign a zero probability to any trigram generated by $q$, we can find a trigram $x_1x_2x_3$ generated by $q$ whose probability in $p$ is minimal. Let $X = -logprob_p(x_1x_2x_3)$, and note that $prob_p(x_1x_2x_3) > 0$ implies $X < \infty$. Since we know by the argument above that when $P$ is sampled from $q$, $\lim_{n(P)\to\infty}(\sum_{m\in M} \frac{(n\pi_k^{n(C)\prime}(m)-2)\cdot\#(m,P)}{n(P)}) = 1$, we have that

$$\sum_{m\in M}(n\pi_k^{n(C)\prime}(m) - 2)prob_q(m) = 1.$$

Thus, we can choose $n_4$ so that

$$\sum_{m\in M, n(m)\le n_4}(n\pi_k^{n(C)\prime}(m) - 2)prob_q(m) > 1 - \varepsilon'/4X.$$

Let $Y = |\{m \in M, n(m) \le n_4\}|$, and choose $n'$ such that if $P$ is sampled from $q$ and ${}_aP_b$ is a substring of $P$ with length greater than $n'$, then with probability at least $1 - \delta'$, for every $m \in M$ we will have that

$$\left|\frac{\#(m, {}_aP_b)}{n({}_aP_b)} - prob_q(m)\right| < \varepsilon'/4XY(n_4 - 2).$$

Let $\pi_k$ be any partial solution of length $k$, and let $r_k$ be the trigram probability distribution described above. Then let $P$ be sampled from $q$, and let $S = \pi_k^{n(C)}(C) = \pi_k^{n(C)\prime}(P)$, and let $a, b$ be indices of $S$ such that $b - a = n > n'$. Finally, we will partition the set $M$ as follows: we let $M'$ be the set $\{m \in M | n(n) \le n_4\}$ and $M''$ be the set $\{m \in M | n(m) > n_4\}$. Thus, we have that

$$\left|\frac{-logprob({}_aS_b)}{n} - H(p, r_k)\right| = \left|\frac{\sum_{m\in M}\#(m, {}_aP_b)(-logprob_p(\pi_k^{n(C)\prime}(m))}{n} - \sum_{m\in M} prob_q(m)\cdot(-logprob_p(\pi_k^{n(C)\prime}(m))\right|.$$

Grouping the terms of these sums into the index sets $M'$ and $M''$, we find that this value is at most

$$\left|\sum_{m\in M'}\left(\frac{\#(m, {}_aP_b)}{n} - prob_q(m)\right)(-logprob_p(\pi_k^{n(C)\prime}(m))\right| + \left|\sum_{m\in M''}\left(\frac{\#(m, {}_aP_b)}{n} - prob_q(m)\right)(-logprob_p(\pi_k^{n(C)\prime}(m))\right|$$

Furthermore, we can break up the sum over the index $M''$ to bound this value by

$$\left|\sum_{m\in M'}\left(\frac{\#(m, {}_aP_b)}{n} - prob_q(m)\right)(-logprob_p(\pi_k^{n(C)\prime}(m))\right| + \left|\sum_{m\in M''}\frac{\#(m, {}_aP_b)}{n}(-logprob_p(\pi_k^{n(C)\prime}(m))\right| + \left|\sum_{m\in M''}prob_q(m)(-logprob_p(\pi_k^{n(C)\prime}(m))\right|$$

Now, for any $m \in M$, we have that the score $-logprob_p(\pi_k^{n(C)\prime}(m)$ equals $\sum_{1\le i\le n(m)-2} -logprob_p(y_iy_{i+1}y_{i+2})$, where $y_i$ is the character at the index $i$ in $\pi_k^{n(C)\prime}(m)$. Taking the maximum possible values for $-logprob_p(y_iy_{i+1}y_{i+2})$, we find that this sum is at most $(n(m) - 2)X$. Applying this bound to the previous formula, we find that it is at most

$$\left|\sum_{m\in M'}\left(\frac{\#(m, {}_aP_b)}{n} - prob_q(m)\right)(n(m) - 2)X\right| + \left|\sum_{m\in M''}\frac{\#(m, {}_aP_b)}{n}(n(m) - 2)X\right| + \left|\sum_{m\in M''}prob_q(m)\cdot(n(m) - 2)X\right|.$$

We can bound each of these three terms separately. Looking at the first sum in this series, we find that with probability at least $1 - \delta'$,

$$\left|\sum_{m\in M'}\left(\frac{\#(m,{}_aP_b)}{n}-prob_q(m)\right)(n(m)-2)X\right| \quad (*)$$
$$\leq \sum_{m\in M'}\left|\frac{\#(m,{}_aP_b)}{n}-prob_q(m)\right|(n(m)-2)X$$
$$\leq \sum_{m\in M'}\left|\frac{\varepsilon'}{4(n_4-2)XY}\right|\cdot(n(m)-2)X$$
$$\leq \sum_{m\in M'}\left|\frac{\varepsilon'}{4Y}\right|$$
$$=\frac{\varepsilon'}{4Y}\sum_{m\in M'}1=\frac{\varepsilon'}{4Y}Y=\varepsilon/4.$$

In order to bound the second sum, we make use of the fact that $\sum_{m\in M}\#(m,{}_aP_b)(n(m)-2)=n({}_aP_b)=n$ to find that once again, with probability greater than $1-\delta'$,

$$\left|\sum_{m\in M''}\frac{\#(m,{}_aP_b)}{n}(n(m)-2)X\right|$$
$$\leq \sum_{m\in M''}\left|\frac{\#(m,{}_aP_b)}{n}(n(m)-2)X\right|.$$

Since $M''=M-M'$, this value is

$$\sum_{m\in M}\left|\frac{\#(m,{}_aP_b)}{n}(n(m)-2)X\right|$$
$$-\sum_{m\in M'}\left|\frac{\#(m,{}_aP_b)}{n}(n(m)-2)X\right|$$
$$=X-\sum_{m\in M'}\left|\frac{\#(m,{}_aP_b)}{n}(n(m)-2)X\right|.$$

This value can further be split into

$$=X-\sum_{m\in M'}\left|\left(\frac{\#(m,{}_aP_b)}{n}+(1-1)prob_q(m)\right)(n(m)-2)X\right|$$
$$\leq X-\left(\sum_{m\in M'}|prob_q(m)(n(m)-2)X|\right.$$
$$\left.-\sum_{m\in M'}\left|\frac{\#(m,{}_aP_b)}{n}-prob_q(m)\right|(n(m)-2)X\right)$$

Using our value for the sum in (*), we find that this is

$$=X-\sum_{m\in M'}|prob_q(m)(n(m)-2)X|$$
$$+\sum_{m\in M'}\left|\frac{\#(m,{}_aP_b)}{n}-prob_q(m)\right|(n(m)-2)X$$
$$\leq X-\sum_{m\in M'}|prob_q(m)(n(m)-2)X|+\frac{\varepsilon'}{4},$$

Using our definition of $n_4$, we can further bound this value by

$$=X\left(1-\sum_{m\in M'}prob_q(m)(n(m)-2)\right)+\frac{\varepsilon'}{4}$$
$$<X\left(1-\left(1-\frac{\varepsilon'}{4X}\right)\right)+\frac{\varepsilon'}{4}$$
$$=X\frac{\varepsilon'}{4X}+\frac{\varepsilon'}{4}=\frac{\varepsilon'}{2}.$$

Finally, we once again make use of the definition of $n_4$ to find that the last sum is

$$\left|\sum_{m\in M''}prob_q(m)\cdot(n(m)-2)X\right|$$
$$=\sum_{m\in M''}prob_q(m)\cdot(n(m)-2)X$$
$$=X\sum_{m\in M''}prob_q(m)\cdot(n(m)-2)$$
$$<X\frac{\varepsilon'}{4X}$$
$$=\frac{\varepsilon'}{4}.$$

Adding these three sums together, we get

$$\frac{\varepsilon'}{4}+\frac{\varepsilon'}{2}+\frac{\varepsilon'}{4}=\varepsilon'.$$

Thus, $\left|\frac{-logprob({}_aS_b)}{n}-H(p,r_k)\right|<\varepsilon'$ with probability greater than $1-\delta'$, as required. □

## 5 Conclusion

In this paper, we discussed a discrepancy between the theoretical and practical running times of certain algorithms that are sensitive to the entropies of their input, or the entropies of the distributions from which their inputs are sampled. We then used the algorithm from Corlett and Penn (2010) as a subject to allow us to investigate ways to talk about average-case complexity in light of this discrepancy. Our analysis was sufficient to give us a bound on the search complexity of this algorithm which is exponential in the cross-entropy between the training distribution and the input distribution. Our method in effect yields a probabilistic bound on the depth of the search heuristic used. This leads to an exponentially smaller search space for the overall problem.

We must note, however, that our analysis does not fully reconcile the discrepancy between the theoretical and practical running time for this algorithm. In particular, our bound still does not explain why the number of search nodes expanded by this algorithm tends to converge on one per partial solution size as the length of the string grows very large. As such, we are interested in further studies as to how to explain the running time of this algorithm. It is our opinion that this can be done by refining our description of the sets $\Pi_k$ to exclude strings which cannot be considered by the algorithm. Not only would this allow us to reduce the overall number of strings we would have to count when determining the bound, but we would also have to consider fewer strings when determining the value of $n'$. Both changes would reduce the overall complexity of our bound.

This general strategy may have the potential to illuminate the practical time complexities of approximate search algorithms as well.

## References

David Arthur, Bodo Manthey, and Heiko Röglin. $k$-means has polynomial smoothed complexity. In *The* $50^{th}$ *Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, 2009. URL `http://arxiv.org/abs/0904.1113`.

Eric Corlett and Gerald Penn. An exact A$^*$ method for deciphering letter-substitution ciphers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1040–1047, 2010.

Richard E Korf and Michael Reid. Complexity analysis of admissible heuristic search. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 1998.

Richard E Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening-A$^*$. *Artificial Intelligence*, 129(1–2): 199–218, 2001.

Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004.