Poet Admits // Mute Cypher: Beam Search to find Mutually Enciphering Poetic Texts

Cole Peterson and **Alona Fyshe**
University of Victoria
cpeterso@uvic.ca, afyshe@uvic.ca

Abstract

The Xenotext Experiment implants poetry into an extremophile's DNA, and uses that DNA to generate new poetry in a protein form. The molecular machinery of life requires that these two poems encipher each other under a symmetric substitution cipher. We search for ciphers which permit writing under the Xenotext constraints, incorporating ideas from cipher-cracking algorithms, and using n-gram data to assess a cipher's "writability". Our algorithm, Beam Verse, is a beam search which uses new heuristics to navigate the cipher-space. We find thousands of ciphers which score higher than successful ciphers used to write Xenotext constrained texts.

1 Introduction

For over a decade, poet Christian Bök has been working on The Xenotext (Bök, 2008), a literary experiment which aims to insert poetry into the DNA of an extremophile organism. In contrast to popular modern data storage mediums like paper and hard disks, DNA is more robust to accidents, and requires minimal maintenance to last hundreds or even thousands of years (Cox, 2001). Many groups are actively pursuing efficient and stable ways to use DNA to encode information (Shimanovsky et al., 2002; Goldman et al., 2013). With his project, Bök aims to leave a lasting cultural contribution inside of an organism, which, as a result of DNA's durability and self-replicating properties, could conceivably survive longer than all other existing works of art.

Furthermore, Bök aims to craft his poem so the protein it instructs the cell to produce is yet another English poem. In a sense, Bök not only turns a microbe into a genetic book, he also engineers the microbe to be, in a sense, a poet. The organism's molecular machinery powers this translation between the two texts, which, at a high level, is a symmetric substitution cipher between the letters, and is described in more detail in Section 2. The two poems (the poet's and the organism's) must both play by the rules we refer to as the Xenotext Constraints:

- Each text is valid natural language.
- The substitution cipher function applied to one text, results in the other text, and vice versa. In other words, the cipher function must be symmetric.
- Whitespace characters (space, new line) encipher themselves, and are the only characters allowed identity mappings.

After four years of work, Bök successfully wrote two English poems which satisfied the Xenotext constraints[1], becoming the first person to do so. The first challenge was finding a cipher which allows for two valid English texts to be written. Finding "writable" ciphers is difficult and is the focus of this paper.

We present Beam Verse, a language-agnostic algorithm driven by the target language's n-gram data, that searches for "writable" ciphers, and suggests words and phrases which can be written under them.

[1]The two poems used the cipher $\binom{\texttt{abcdefghijlmq}}{\texttt{tvukyspnoxrwz}}$

We do not concern ourselves with the full biochemical constraints of The Xenotext (eg. the actual impact of the protein and the cell's reaction to it, or its viability after folding) and instead only consider the Xenotext constraints listed above. This problem sits at the intersection of natural language processing and cryptography, and is a prerequisite to the genetic engineering required to fully realize a living xenotext. Our algorithm uncovers new ciphers which make satisfying the Xenotext constraints in English possible, and makes it easier for a poet of any language to investigate the feasibility of writing two mutually enciphering texts in their own tongue.

2 Genetic Background

DNA provides instructions to a living cell to produce a protein. DNA has a double helix structure (reminiscent of a twisted ladder) and is made up of four nucleotides: adenine, cytosine, guanine, and thymine, commonly abbreviated as A, T, C, and G. Each nucleotide always appears paired with another across the "rung" of the ladder, A with T, C with G, and vice versa. To transfer the data in the DNA to the protein-producing ribosome, the double helix is "unzipped", separating the ladder at the rungs, and a copy of the exposed DNA sequence called an *mRNA transcript* is synthesized, pairing in the same way the DNA did, with the exception that adenine in the DNA strand pairs with uracil (U) in the mRNA. The ribosome reads the mRNA as instructions to produce a specific protein. A protein is a sequence of amino acids and each triplet of nucleotides in the mRNA (called a *codon*) represents one of the twenty amino acids (see Table 2) (Campbell and Reece, 2008).

We can write in the DNA of an organism by having a codon represent a letter. When this sequence of codons (the full poem) is read by the organism, it then writes a sequence of amino acids, each of which represent a letter, in reply[2]. The letters in each poem have a bijective relationship determined by the biochemical processes that link them. For example, as shown in Table 2, the letters E and T are mutually linked, as wherever the poet uses the letter E, the cell uses the letter T, and vice versa. If the poet was to write "mute" instead of "poet", the cell would write "poet" instead of "mute".

Poet's Letter	P	O	E	T
DNA Codon	AAC	GAG	CCG	GGC
mRNA Codon	UUG	CUC	GGC	CCG
Amino Acid	phenylalanine	leucine	glycine	proline
Cell's Letter	M	U	T	E

Table 1: Sample translation from text through DNA to new text

This view of DNA is extremely simplistic, and serves only to motivate and provide context for the rest of this paper. When using a codon to represent a poet's letter and an amino acid to represent one of the organism's letters, many complexities arise which add further constraints to the text, which we ignore in the remainder of the paper. When actually inserting his poetry into a cell, Bök struggled to get the organism to express the correct protein, because he failed to account for the full complexity of the cell and caused the cell to "censor" itself (Wershler, 2012). However, we consider these additional constraints to be future work.

3 Substitution Ciphers

Substitution ciphers are among the earliest known forms of cryptography, having existed since at least the days of Caesar (Sinkov, 1966). They work by replacing one letter in the plaintext with another to form the ciphertext. However, they are never used in modern cryptography because, despite a large keyspace, substitution ciphers do not change the letter frequency between plaintext and ciphertext. When properties of the plaintext are known (like the language it is written in), letter or word frequency data from that language can be used to quickly crack the cipher and uncover the key.

Every word has a deterministic encryption under a cipher. The encrypted word could be nonsense, or it could be another word. The word "poet", for example, can encrypt to many other words, including "mute". The "poet↔mute" word-pair forms what we call a *partial cipher*, and notate as $\binom{\texttt{poe}}{\texttt{mut}}$. We

[2]This makes the writing lipogrammatic, as there are only 20 amino acids, and one of them (Serine) must be used as the space character. Serine is used for the space because it is the only amino acid to encipher itself, as both codons AGT and TCA produce it, mirroring the constraint that the space is mapped to itself in our ciphers.

say this partial cipher has a *cardinality* of three, as it defines three letter pairings. A *full cipher* in a 26-letter language has a cardinality of 13. We also refer to $\binom{\texttt{poe}}{\texttt{mut}}$ as a *primary* cipher, because it is the lowest cardinality cipher to contain the word-pairing "poet↔mute".

As no characters except whitespace are allowed identity mappings a word-pair like "eat↔cat" is not valid, as both *a* and *t* would have to map to them selves. The symmetric Xenotext constraint prohibits "admits" from pairing with "cipher", as the letter *i* would require a mapping to both *d* and *h*. However, "admits" can pair with the alternative spelling "cypher", forming the primary cipher $\binom{\texttt{admits}}{\texttt{cypher}}$. We can combine this cipher with $\binom{\texttt{poe}}{\texttt{mut}}$, as none of the letter pairs conflict with each other – they are *compatible* with each other. Together, they form $\binom{\texttt{poeadis}}{\texttt{mutcyhr}}$. As the letter-pairs in $\binom{\texttt{poe}}{\texttt{mut}}$ and $\binom{\texttt{admits}}{\texttt{cypher}}$ are subsets of the letter-pairs in $\binom{\texttt{poeadis}}{\texttt{mutcyhr}}$, we call $\binom{\texttt{poe}}{\texttt{mut}}$ and $\binom{\texttt{admits}}{\texttt{cypher}}$ *subciphers* of $\binom{\texttt{poeadis}}{\texttt{mutcyhr}}$, and say that $\binom{\texttt{poeadis}}{\texttt{mutcyhr}}$ *extends* $\binom{\texttt{poe}}{\texttt{mut}}$ and $\binom{\texttt{admits}}{\texttt{cypher}}$. For any two ciphers ϕ_1 and ϕ_2, we use the notation $\phi_1 \subset \phi_2$ to denote that ϕ_1 is a subcipher of ϕ_2.

If we applied $\binom{\texttt{poeadis}}{\texttt{mutcyhr}}$ to "Poet Admits" (the first part of this paper's title), it would result "Mute Cypher" (the second part of the paper's title). The title refers to the difficulty of writing under the Xenotext constraint, as it is hard to find a cipher where writing is possible, most of the ciphers are mute. Once all of the possible word pairs of a target language have been discovered (Section 7) the challenge becomes navigating the tradeoffs of including a letter pair, as each letter pair eliminates the possibility of using some word-pairs, while including other word-pairs.

If a language has an odd number of characters a symmetric substitution cipher is not possible using every letter. We must decide which letter to leave out of our texts. This is accomplished by inserting a null letter (which appears nowhere in the language) into our letter set, thus giving the language an even number of characters. At the conclusion of Beam Verse the letter paired with null is the character to leave out.

4 Scoring a Cipher's "Writability"

When scoring a cipher, an important consideration is what makes one cipher more "writable" than another. We might score a cipher on the number of valid words under it, as having more words at your disposal makes it easier to write, but this is not necessarily so if all the words are all rare and useless. To combat this, we weight words based upon their frequency in language, so that better, more frequent words contribute more to a ciphers overall score. This values highly frequent and syntactically important words, like "the" or "and", while also allowing a large number of infrequent words to also contribute significantly to the score. However, a word's usefulness when writing mutually enciphering texts is explicitly tied to its sister word under the cipher. "The" looses its usefulness if every time it is used in one poem, a less frequent word like "nag' must be used in the other. We propose that since a word pair is only as good as its weakest word, that ciphers be scored by taking the sum of all available word pairs, counting the minimum frequency of the two words. This means that the word pair "the↔nag" would count the frequency of "nag".

Multiple different word pairings can form the same primary cipher. For example, $\binom{\texttt{th}}{\texttt{ea}}$ is formed by both "the↔eat" and "he↔at", and would count the score of both word-pairs. As there are always less or equal primary ciphers than word-pairs, it is more memory efficient to store the score of all the primary ciphers than to store the scores of all the word-pairs. We count the score of a primary cipher ϕ_p towards a full cipher ϕ_f if it is a subcipher of ϕ_f. Formally, if P is the set of all primary ciphers and $\phi_p \in P$, the score of ϕ_f is $\sum_{\phi_p \subset \phi_f} score(\phi_p)$.

Alternatively, this could be expressed as a dot product between a vector where every element is the score of a primary (the score vector, s), and a vector indicating whether a primary cipher is a subcipher (the heuristic vector, h), as seen in equation 1. In section 8 we show how h can be calculated efficiently, and also how it can be use to provide an upper and lower bound the score of a full cipher extended from a partial cipher.

$$score = s \cdot h \tag{1}$$

The concept of a word-pair can easily be extended

to include longer word-level n-grams. Like words, every n-gram either enciphers to nonsense or has a sister n-gram it is locked to under a cipher. All n-gram pairs also have an associated frequency in the language, and so can contribute to the score of a cipher in the same way as single words do: by the minimum of the two n-gram's frequency counting as the weight of the pair. Using n-grams also indirectly captures some syntactic structure, and allows for generation of sample phrase and sentence generation from the cipher by chaining together n-grams. These small phrases can be used to quickly prototype poetry. For our word list and frequency data, we use Google's n-grams (Michel et al., 2011), but any dataset could be used, and would give different ciphers depending on the data's source.

5 Graphical and Combinatoric Representation

There are 7,905,853,580,625 possible symmetric substitution ciphers in a 26 letter language like English. Even with the efficient means of scoring ciphers shown in section 8 (which can calculate a full cipher's score in $\sim$ 300 microseconds) the brute force solution would take over 75 years of computing time. To avoid this expensive full calculation, we formulate the problem as a graph of partial ciphers and use beam search to navigate the graph to high valued full solutions. We regret that the smallest non-trivial graph (of a 6 letter language) is too large to be included here; it requires 75 nodes arranged in three layers which takes up an entire page, but it can be found on our website[3]. An incomplete graph is shown in Figure 1. As we search the cipher space we trace edges up through the graph to a full cipher solution.

The size of the m^{th} layer of a n letter language is defined by equations 2-4. The counts for a 26-letter language and a derivation of this equation can be seen on our website.

$$f(m,0) = 1 \quad (2)$$

$$f(1,n) = n \times (n-1)/2 \quad (3)$$

$$f(m,n) = f(m-1,n) \times f(1, n-2\times(m-1))/m \quad (4)$$

[3] http://www.langlearnlab.cs.uvic.ca/beamverse

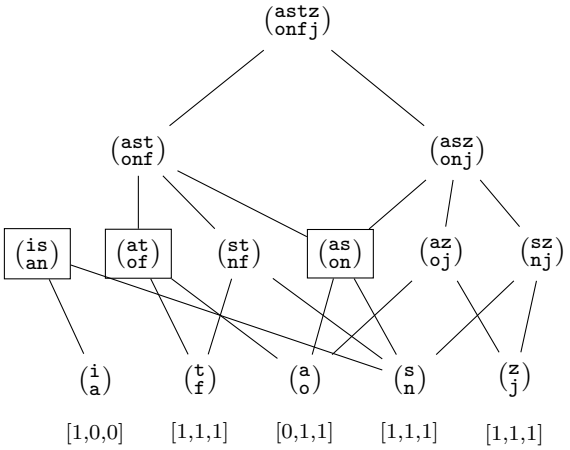

Figure 1: An incomplete cipher-graph, showing some of the partial ciphers which descend from $\binom{\texttt{astz}}{\texttt{onfj}}$. Three primary ciphers are shown in boxes, and are the same example primary ciphers used in Section 8. Compatibility vectors (discussed in Section 8.1) are shown for every letter-pair. Edges in this graph represent a difference of a letter-pair between ciphers. Each cardinality of partial cipher has a layer in the graph, which is our beam in the search.

6 Beam Search

A beam search algorithm does not require us to store all of this graph in memory, as we only examine a subset of the ciphers anticipated to be the best. Beam search works by enumerating all possibilities for one step of a solution (one layer of the cipher graph), sorting those options by a heuristic, keeping the best n partial solutions, and pruning the rest (Edelkamp and Schroedl, 2011). We fully expand one step further on the best n partial solutions, repeating the pruning and expanding process until a set of full solutions are reached. Beam search can effectively crack substitution ciphers, as shown by Nuhn et al. (2013).

A static beam size could be used (keeping the best thousand partial ciphers at each step, for example), however, the lower cardinality the partial, the more possibilities it generates. Every cardinality-1 partial cipher generates 276 possible cardinality-2 partial ciphers, whereas a cardinality-12 partial cipher only generates one possible full cipher (as there are only two unpaired letters remaining, therefore they must pair together). A constant beam size will limit the algorithm's performance in later stages of the search

if this is not accounted for.

We can rearrange our recursive definition in Equations 2 to 4 to determine the beam size which will generate exactly as many partial ciphers as we can hold in memory. If we want to generate B ciphers in an n letter language, and are at layer m, we should prune the beam to $b(m, n)$. This can be found by replacing replacing $f(m, n)$ for B, and $f(m-1, n)$ for $b(m, n)$ in equation 4 and rearranging to produce equation 5, removing m from equation 4 because we cannot assume duplicates will be generated.

$$b(m, n) = \frac{B}{f(1, n - 2 \times (m-1))} \tag{5}$$

7 Generating Primary Ciphers

In order to generate the primary ciphers, we must find all words which can encipher with each other, and record their score. Rather than checking every word against every other word, many useless checks can be avoided by hashing each word or n-gram according to the pattern of its letters, a concept which Hauer et al. (2014) called "pattern-equivalence" and used to crack substitution ciphers. We use a key which represents each letter in the n-gram with a number or symbol, ordered by first occurrence of the letter, while maintaining whitespace (eg. "and we are" $\rightarrow$ "123 45 165"). Another trigram like "his or her" would also generate the same key, and so the two trigrams would be checked against each other to see if they validly encipher, which they do, forming a primary partial cipher $\binom{\texttt{andwr}}{\texttt{hisoe}}$.

A match on the hash key does not guarantee that the words form a valid pairing. Many words which share a key form invalid word-pairings due to the symmetric or identity restrictions of the Xenotext constraint (eg. "cat↔eat", which share the key "123", or "admits↔cipher", which share the key "123456" are both invalid word-pairings). The score of a primary cipher is the sum of the score of all word-pairs which generate the primary. The algorithm is shown in Algorithm 1.

Algorithm 1 Generating Primary Ciphers

```
function GENERATE PRIMARIES(ngrams)
  for word_1 ← ngrams do
    key ← pattern(word_1)
    for word_2 ← patternDict[key] do
      if mutually-encipher(word, word_2) then
        primaries[encipher(word_1, word_2)] += minScore(word_1, word_2)
      end if
    end for
    patternDict[key].add(word_1)
  end for
  return primaries
end function
```

8 Beam Search Heuristics

Recall from Section 3 that, in a full cipher, all letters have a defined mapping (the cipher has a cardinality of 13), while in a partial cipher some letters have undefined defined mappings, and that a primary cipher is the minimal cardinality partial cipher to contain a particular word-pair and is the building block of a cipher's score. We explore three different heuristics which calculate the true score for full ciphers, and estimate the score of full ciphers extended from a partial cipher by forming an upper and lower bound. All heuristics produce a vector h, which forms the score for a cipher when dotted with the score vector s (Equation 1). For full ciphers this vector will be the same regardless of the heuristic used, and the score from Equation 1 will be the true score of the full cipher, whereas different heuristics will give different values for a partial cipher, and thus guide Beam Verse in different directions. We implement these heuristics using bitwise operations, making them both memory and CPU efficient.

To demonstrate the calculation of the heuristics, we use the following primary ciphers as a running example, $\binom{\texttt{is}}{\texttt{an}}$ (score: 100), $\binom{\texttt{at}}{\texttt{of}}$ (score: 82), and $\binom{\texttt{on}}{\texttt{as}}$ (score: 76), which are the same primary ciphers as are shown in Figure 1. These three primaries would form the score vector, which is shared amongst all heuristics, is $s = [100, 82, 76]$. Thus $P = \{\binom{\texttt{is}}{\texttt{an}}, \binom{\texttt{at}}{\texttt{of}}, \binom{\texttt{on}}{\texttt{as}}\}$, and $|P| = 3$. We show the heuristic calculation for all three heuristics on the partial cipher $\binom{\texttt{asz}}{\texttt{onj}}$.

8.1 Upper Bound: Compatibility Vector

Recall that two ciphers are compatible if they have no conflicting letter-pairs. If two ciphers are compatible, it is possible to combine them. Every cipher ϕ has a vector representing compatibility with the

primary ciphers P. This vector is $|P|$ long, and contains a 1 in the i^{th} element if ϕ is compatible with the i^{th} primary cipher, and a 0 if it is not.

We use a superscript c on a cipher to notate its compatibility vector. Here are the compatibility vectors for four letter-pairs, using the primary ciphers outlined above, and are shown in Figure 1:

$(\substack{\texttt{i}\\\texttt{a}})^c = [1,0,0]$, $(\substack{\texttt{s}\\\texttt{n}})^c = [1,1,1]$,
$(\substack{\texttt{a}\\\texttt{o}})^c = [0,1,1]$, $(\substack{\texttt{z}\\\texttt{j}})^c = [1,1,1]$.

This is an upper-bound because the primary ciphers compatible with ϕ may not be compatible with each other. For example, the null cipher, which has no letter pairings, is compatible with all primary ciphers, but no full cipher contains all primaries. When we combine two ciphers ϕ_1 and ϕ_2, which have compatibility vectors ϕ_1^c and ϕ_2^c, the resulting cipher ϕ_3 has a compatibility vector $\phi_3^c = \phi_1^c \wedge \phi_2^c$, where $\wedge$ is the bitwise AND operation. We calculate the compatibility vector for every letter-pair, and can combine those vectors to determine the compatibility vector for any cipher. The heuristic's score for $(\substack{\texttt{asz}\\\texttt{onj}})$ follows.

$h = (\substack{\texttt{asz}\\\texttt{onj}})^c = (\substack{\texttt{a}\\\texttt{o}})^c \wedge (\substack{\texttt{s}\\\texttt{n}})^c \wedge (\substack{\texttt{z}\\\texttt{j}})^c = [0,1,1]$
$score(\substack{\texttt{asz}\\\texttt{onj}}) = 100 \cdot 0 + 82 \cdot 1 + 76 \cdot 1 = 158$

8.2 Lower Bound: Guarantee Vector

We can calculate another vector, g for every cipher ϕ which represents whether each primary cipher is a subcipher of ϕ. This forms a lower bound guarantee because any cipher which extends from ϕ will also contain the primary ciphers in g, plus potentially more. The null cipher in this case would have a heuristic vector g of all zeros, as it does not contain any of the primary ciphers. Likewise, in this P, all of the individual letter pairs $((\substack{\texttt{a}\\\texttt{o}}), (\substack{\texttt{s}\\\texttt{n}}), (\substack{\texttt{z}\\\texttt{j}}))$ would have a heuristic vector of all zeros, as all of the primaries require at least two letter-pairs.

Efficiently implementing this heuristic is slightly more challenging than the compatibility heuristic. Our method, which uses bitwise operations and is cacheable like the compatibility vector. Using this heuristic, g of $(\substack{\texttt{asz}\\\texttt{onj}})$ is $[0,0,1]$, as $(\substack{\texttt{is}\\\texttt{an}}) \not\subset (\substack{\texttt{asz}\\\texttt{onj}})$, $(\substack{\texttt{at}\\\texttt{of}}) \not\subset (\substack{\texttt{asz}\\\texttt{onj}})$, and $(\substack{\texttt{as}\\\texttt{on}}) \subset (\substack{\texttt{asz}\\\texttt{onj}})$.

This heuristic therefore scores $(\substack{\texttt{asz}\\\texttt{onj}})$ as follows:

$score(\substack{\texttt{asz}\\\texttt{onj}}) = 100 \cdot 0 + 82 \cdot 0 + 76 \cdot 1 = 76$

8.3 Middle Ground: Medium Vector

Both of the two aforementioned heuristics have weaknesses. The optimistic, max heuristic does not differentiate between a primary cipher it already has and one that it potentially has, and the conservative min heuristic is greedy and short-sighted. Our third heuristic incorporates elements from the first two, to ideally form a measure that is neither overly optimistic, or overly short-sighted. Unlike the lower bound heuristic in Section 8.2, which requires all letter pairings to count a primary cipher, this medium heuristic counts some of the primary cipher's score if some of the letter-pairs are present. For example, if a partial cipher has $3/4$ of the required letter pairings for a primary, it would count 75% of the score.

For example, $(\substack{\texttt{asz}\\\texttt{onj}})$ has one of the two letter pairings of the first primary, $(\substack{\texttt{is}\\\texttt{an}})$; one of the two letter pairings of the second primary, $(\substack{\texttt{at}\\\texttt{of}})$; and two of the two letter pairings of the third primary, $(\substack{\texttt{on}\\\texttt{as}})$. We represent this as [0.5,0.5,1]. However, we know from Section 8.2 that the first primary is incompatible with $(\substack{\texttt{asz}\\\texttt{onj}})$, and so we do not count its score. That makes the heuristic vector $h = [0, 0.5, 1]$, and $score(\substack{\texttt{asz}\\\texttt{onj}}) = 100 \cdot 0 + 82 \cdot .5 + 76 \cdot 1 = 117$.

We have now evaluated the same cipher using three different heuristics, all which produce a different score. These scores are guaranteed to converge to the same value at the full cipher.

8.4 Speed improvements

Table 8.4 shows the massive performance gains of the heuristics, which are over 3000 times faster than the simplistic means of scoring by iterating over every word and checking if it enciphers to anything useful.

9 Related Work

Nuhn et al. (2013) use a beam search algorithm to crack substitution ciphers. Our work differs from their's in several key ways: in Nuhn et al. (2013) there are two distinct symbol spaces, that of the ciphertext and that of the plaintext and so there is no concept of symmetry. Each step of Nuhn et al.'s beam search explores pairing a ciphertext character with a plaintext character, and decides upon the "extension-order" to pair the ciphertext letters, whereas each step of our search pairs two characters

Heuristic	Time	Memory
word	$1 \times 10^6 \mu s$	all words +1int/word
med 8.3	$3 \times 10^3 \mu s$	n bits/primary + 1 int/primary
min 8.2	$2 \times 10^3 \mu s$	n bits/primary + 1 int/primary
max 8.1	$3 \times 10^2 \mu s$	1 bit/primary + 1 int/primary

Table 2: Time to score a cipher using different means, and each mean's memory requirements. The word method stores the strings of all words and enciphers them and checks if they are valid words. It will produce the same value as the min heuristic.

together. As such, we make 13 decisions, not 26.

Additionally, the search space of the non-symmetric and symmetric ciphers are characteristically different. If the "extension-order" is predetermined as in Nuhn et al.'s work, there is only one path from the bottom of the graph to a full solution. In contrast, our graph has 13! different paths to any full solution, as all the permutations of the 13 different letter pairs are valid paths. On the one hand, this highly connected property of the graph means that we can prune our beam to a smaller size, as failing to expand a partial cipher does not eliminate it from appearing as a subcipher in a future solution like it does for Nuhn et al..

However, the connectedness of our cipher graph does present new challenges. As the Xenotext constraints are not satisfied by one cipher, we want to maximize the number of different solutions presented to the writer which each allow for unique expressive potential. We observe, however, that the connectedness property results in a final beam which is smaller and less diverse than would be anticipated. This is caused by multiple partial ciphers in a beam sharing the same "propensity" to become an identical full cipher. We solve this by enforcing that every partial cipher in the beam be incompatible with every other, thereby guaranteeing that no two partial ciphers can share the same "propensity", and that all possibilities generated from them in all future layers will be unique. As there are many ($\mathcal{O}(n^2)$) compatibility comparisons to be made at every beam, we limit only enforce compatibility for the first thousand ciphers in the beam.

Our scoring function is also entirely different from what would be used to crack a substitution cipher. Unlike Beam Verse, cracking software is tied to a ciphertext, and typically uses character-level n-gram data to calculate the probability that a decipherment is valid. Beam Verse, on the other hand, uses word-level n-grams as the basis of scoring, and is not tied to any given text, but suggests fragments of text which satisfy the Xenotext constraint.

10 Results

The raw score of a cipher changes according to the dataset, and so we report the score divided by highest scored cipher across all of the heuristics. Table 10 shows results using unigrams, while Table 10 shows results for primary ciphers generated from bigrams.

Heuristic	High	Low	End Beam Size
max	0.74	0.53	12160
min	0.98	0.97	4223
med	0.93	0.73	13043
max incomp	0.71	0.59	12160
min incomp	1.00	0.97	4181
med incomp	0.85	0.74	13043
Bök	0.39		

Table 3: Normalized scores for three different heuristics on highest 2^{16} **unigram** primary ciphers, and a variable beam aiming for 2^{15} ciphers. "Incomp" means that we enforce that all partial ciphers in the beam be incompatible with each other. The low value is the normalized score of the index of the shortest end beam. This is a better comparison that the last cipher in the beam, as the length of the beams is variable across heuristics.

Heuristic	High	Low	End Beam Size
max	0.81	0.73	9631
min	1.00	0.94	5777
med	0.97	0.88	13291
max incomp	0.81	0.73	9631
min incomp	0.97	0.88	13301
med incomp	0.97	0.84	13291
Bök	0.23		

Table 4: Normalized scores for different heuristics on highest 2^{20} **bigram** primary ciphers, and a variable beam aiming for 2^{15} ciphers.

We note that all ciphers we generate, regardless

of heuristic, score higher than the cipher Bök used to write his poems. This suggests that there are many ciphers other than Bök's which can be used to write Xenotext constrained poetry. Poems written using ciphers generated from Beam Verse can be found on our website. However, attempting to write with some high-scoring ciphers has revealed that our scoring metric may be only loosely correlated with the true "writability", as some ciphers which score higher that Bök's we find more difficult to write with.

Bök's cipher also scores relatively worse than the top ciphers using a bigram model (Table 10). Many of the bigrams Bök uses in his poems are not frequent enough to be in the Google bigrams. Anecdotally, we find ciphers generated using bigram data to be more writable, as bigram models begin to capture syntactic structure of the language.

Enforcing that each cipher in the beam be incompatible with every other showed minimal gains with some heuristics and minimal losses in others. It does, however, guarantee that more ciphers will be generated. Enforcing incompatibility is probably not worth the processing time if memory permits increasing the size of the beam instead.

The top scoring cipher[4] according to the unigram model performs similarly to the Bök cipher when scored against the bigram model, accumulating only 24% of the points the highest scoring bigram cipher does. The top bigram cipher[5] scores 68% of the top unigram cipher's score when using unigram scoring, not as drastic of a difference, but still low enough to be pruned by Beam Verse and not be discovered. The discrepancy in scores between models suggests that "writable" ciphers are excluded from our final results, and also encourages running Beam Verse on additional datasets to find new ciphers. A score which incorporates elements from multiple models of language might be explored, searching for ciphers which perform well across all datasets.

11 Further Work

Work in Kao (2011) sets out to quantify good poetic style and techniques. We note that some poetic techniques, like alliteration and anaphora, are preserved through the substitution cipher. We could boost alliterative n-grams to encourage Beam Verse to include alliterative n-grams.

As Beam Verse is language agnostic, all of the work here is applicable to other languages. The Xenotext constraints might be more easily satisfied in a different language than English, perhaps a language with a smaller character set like Hawaiian (which only consists of thirteen letters). Additionally, The Xenotext project as defined here only minimally uses the organism to actively write – the organism does not have any degree of freedom to express itself as its poem is precisely determined by the author's poem. However, DNA possesses computational power (Paun et al., 2005), which could be leveraged to generate natural language. By taking advantage of the complexity of the cell, its output could be more loosely defined, and change according to mutations in the DNA.

Further investigation can also be done into quantifying the "writability" of a limited vocabulary (perhaps using semantic and grammar data), and constrained text generation under constraint. Poetic endeavours with rigid mathematical constraints are not only attempted by Bök. Any work in the traditions of the Oulipo, a primarily French-speaking group of writers who explore the creative potential of mathematical and logical constraints, would stand to benefit immensely from software tools designed to aid constrained writing. Whereas visual artists and musicians have been quick to use computers to produce images and sounds which would have been impossible by traditional means, writers have been slow to use computers to produce works which would have been impossible to create otherwise.

12 Conclusion

In this paper we present a new metric to quantify "writability" of a symmetric substitution cipher. We experiment using three different heuristics in a beam search, an algorithm we call Beam Verse. We find that our score for "writability", which takes the minimum frequency of a word or n-gram pair, is effective at finding candidate ciphers, but is not a perfect metric of "writability" in this constrained environment. "Writability" is highly subjective, and possibly requires more data than just n-gram frequency

[4] $\begin{pmatrix}\texttt{abcdegijkmnqv}\\ \texttt{fhlryutpswozx}\end{pmatrix}$
[5] $\begin{pmatrix}\texttt{abcdefjklnpqx}\\ \texttt{ightomuvrswzy}\end{pmatrix}$

(eg. semantic and grammar information). Luckily, beam search is highly flexible, and any scoring function, perhaps using a more sophisticated model of writability, could be used in place of the one used here.

Source code and highly scoring ciphers are available for download[6].

References

Christian Bök. 2008. The xenotext experiment. *SCRIPTed*, 5:228–231.

Neil A. Campbell and Jane B. Reece. 2008. *Biology*. Pearson, 8th edition.

Jonathan PL Cox. 2001. Long-term data storage in dna. *TRENDS in Biotechnology*, 19(7):247–250.

Stefan Edelkamp and Stefan Schroedl. 2011. *Heuristic search: theory and applications*. Elsevier.

Nick Goldman, Paul Bertone, Siyuan Chen, Christophe Dessimoz, Emily M LeProust, Botond Sipos, and Ewan Birney. 2013. Towards practical, high-capacity, low-maintenance information storage in synthesized dna. *Nature*, 494(7435):77–80.

Bradley Hauer, Ryan Hayward, and Grzegorz Kondrak. 2014. Solving substitution ciphers with combined language models. pages 2314–2325.

Justine T Kao. 2011. A computational analysis of poetic craft in contemporary professional and amateur poetry.

Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Joseph P Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, and Jon Orwant. 2011. Quantitative analysis of culture using millions of digitized books. *science*, 331(6014):176–182.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Malte Nuhn, Julian Schamper, and Hermann Ney. 2013. Beam search for solving substitution ciphers. Citeseer.

Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa. 2005. *DNA computing: new computing paradigms*. Springer Science & Business Media.

Boris Shimanovsky, Jessica Feng, and Miodrag Potkonjak. 2002. Hiding data in dna. pages 373–386. Springer.

Abraham Sinkov. 1966. Elementary cryptanalysis: A mathematical approach, mathematical association of america, 1966. *Additional Reading*.

Darren Wershler. 2012. The xenotext experiment, so far. *Canadian Journal of Communication*, 37(1):43.

[6] http://www.langlearnlab.cs.uvic.ca/beamverse