Solving Anagrams with Integer Linear Programming

Pavol Zajac **Tomáš Selep** **Eugen Antal**
Slovak University of Technology in Bratislava
Ilkovičova 3, 841 04 Bratislava
Slovakia
pavol.zajac@stuba.sk xselep@stuba.sk eugen.antal@stuba.sk

Abstract

Given some sequence of letters, an anagram is formed by changing their order to create a different text. In a historical context, anagrams were popular mainly as puzzles, but they are also connected to classical transposition ciphers. To solve an anagram means to rearrange the letter sequence to a form that is acceptable as a word or sentence in some language.

In this article, we formalize the anagram solving problem. We focus on anagrams based on a simplified language model based on fixed dictionaries. We study the applicability of known methods for this problem. We propose a method of anagram solving based on integer linear programming. The new method is not strictly superior to existing methods but provides new tools to tackle the problem. The new representation shows potential for integration with Word2Vec representation of words for finding potentially meaningful anagrams in natural languages.

1 Introduction

One of the main categories of historical ciphers is transpositions, the basic principle of which consists in the rearrangement of letters. Transposition leaves the characters of the plain text invariant (Bauer, 2002). These ciphers are closely related to the term anagram — in a traditional sense of rearranging one (meaningful) text to spell another (meaningful) one (Kahn, 1996). Anagrams have a rich history. They were popular in the past, used in puzzles by amateurs, but also used by scientists to establish a priority of inventions (Bauer, 2002). Anagrams were used in the old inscriptions, and many of them are still unsolved, such as the mysterious anagram from the church of the Poor Clares in Bratislava (Antal and Zajac, 2024).

Anagrams are also related to the cryptanalysis of transposition ciphers: the anagram problem is a generalization of the problem of reconstructing the plain text from the letters of the cipher text (Bauer, 2002). However, it has to be noted that from a specific group of letters, multiple meaningful messages can be constructed (Bauer, 2002). On the other hand, if two or more cipher texts of the same length (encrypted with the same transposition using the same key) are available, the *multiple anagramming* method (Gaines, 1956; Barker, 1994; Kahn, 1996; Dunin and Schmeh, 2020) can be used to solve[1] the cipher texts. This method is considered as a general solution for all transposition ciphers (Kahn, 1996). It can also be used to solve complex transposition ciphers such as double or triple columnar transposition ciphers, for which either there are no efficient methods of solving (Antal et al., 2020), or they only work with some limitations (Barker, 1994; Lasry et al., 2014).

Anagram solving is essentially a process that tries to reconstruct meaningful sentences out of the input letters. Note that all rearrangements of a set of words are anagrams of each other. Thus, a sufficient goal in anagram solving is to arrange the input letters into words. For a given input set of letters, there are typically many possible anagram solutions. For example:

Input: `aadehooortuwyy`

Output:

- `who,are,you,today`
- `the,door,you,away`
- `you,heat,door,way`
- `how,are,you,today`

[1] Solving does not necessarily mean obtaining the original cipher key of the transposition cipher, rather obtaining the plain text and the global permutation applied to the whole plain text.

Proceedings of the 8th International Conference on Historical Cryptology, pages 149-154
Poznań, Poland, 16-18 June, 2025

- `you,year,wood,hat`
- `way,outdoor,yeah`
- ...

Various methods of anagram solving were studied in the past, from testing simple rearrangements of letters, to more advanced combinatorial and dictionary based methods (such as presented by Knuth (2011)). Modern approaches, such as Nishino et al. (2019) proposed, try to incorporate the methods from artificial intelligence and large language models to generate anagrams that have a meaning in some specified language. In this article, we try to formalize anagram solving (over some specific dictionary) as a difficult computational problem. Furthermore, we provide a new mathematical formalization of the problem that presents the anagram solving as a specific instance of the integer linear programming (ILP) problem. We experimentally explore whether this representation is suitable for anagram solving and provide some new ideas on how this new representation can be extended to account for solving anagrams in natural language using modern large language models.

2 Preliminaries

Let A be an alphabet. We will call $a \in A$ a letter, and a sequence of letters $s \in A^*$ a string. Let D be a finite set of strings. We will call D a dictionary. Each $w \in D$ will be called a word. A finite sequence of words $s \in D^*$ will be called a sentence. Each sentence can also be considered a string (of letters), but some strings are not sentences. Note that some strings can represent multiple sequences, depending on the dictionary D, because without separators, we can not guarantee the unique parsing of strings into words.

A language that consists of all strings over A will be denoted by $\mathscr{A} = A^*$. Language that contains all sentences (concatenation of strings from dictionary D) will be denoted by $\mathscr{D} = D^*$. A language that is a proper subset of $\mathscr{D}$ will be denoted by $\mathscr{L} \subset \mathscr{D}$, $\mathscr{L} \neq \mathscr{D}$. In practice, we will suppose that $D \subset \mathscr{L}$, that is, every word in the dictionary D taken separately belongs to the language $\mathscr{L}$, but the techniques can be adapted for any subset of $\mathscr{D}$.

Let $s \in \mathscr{A}$ be a string of length n, and let $\{\{s\}\}$ denote a multiset of letters of string s. We say that s_1 is an anagram of s_2 if and only if $\{\{s_1\}\} = \{\{s_2\}\}$. In other words, s_2 contains all letters from s_1 in any (different) order. Let n_i denote the frequency of letter i in $\{\{s\}\}$, and $|s| = n = \sum_{i \in A} n_i$. Then there exists

$$C(s) = \frac{n!}{\prod_{i \in A}(n_i!)}$$

different anagrams of s in $\mathscr{A}$. As $\mathscr{L}$ is a (proper) subset of $\mathscr{A}$, we can naturally ask how many anagrams of s are in $\mathscr{L}$. This question is difficult in general, and the decision version of this problem is NP-complete (it can be formulated as a subset sum problem, which is NP complete (Karp, 2009)).

3 Anagram solving

Under *solving an anagram* for the given string s, we mean rearranging the letters of s to form a string s' that belongs to language $\mathscr{L}$. For natural languages, this is a difficult task due to complex grammar rules, thus, we have to simplify this process slightly in computer assisted anagram solving.

Let D be a dictionary of individual words of the language $\mathscr{L}$, and let $\mathscr{D} = D^* \supset \mathscr{L}$. We can solve anagrams in $\mathscr{L}$ with a two-step process:

1. Enumerate anagrams of s that belongs to (dictionary based) language $\mathscr{D}$. That is, we only construct a concatenation of words from dictionary D, ignoring grammar rules of $\mathscr{L}$.
2. Filter potential candidates (anagrams in $\mathscr{D}$) by checking the grammar rules of $\mathscr{L}$.

An alternative approach is to combine these two steps by using statistical hints from a language model, such as the method presented by Nishino et al. (2019). However, in this article, we only want to focus on step 1 in the process, the construction of anagrams in $\mathscr{D}$.

Furthermore, observe that language $\mathscr{D}$ contains all possible sequences of words from dictionary D, regardless of their order. Therefore, if sentence $w_1|w_2|\cdots w_n \in \mathscr{D}$ is an anagram of s, any sentence that is a permutation of these words will also be an anagram of s. Thus, for the given input string s, the anagram solving method based on dictionary D will output a multiset of dictionary words $W = \{\{w_1, w_2, \ldots, w_n\}\}$, $w_i \in D$, such that $\{\{s\}\} = \{\{w_1|w_2|\ldots|w_n\}\}$.

3.1 Anagram solving based on letter permutations

A naïve method of anagram solving involves a generate and test approach: for the given input

string s, create all letter permutations, and test whether the result can be parsed into dictionary words.

The complexity of this method is given by $C(s) \cdot T_1$, where $C(s)$ is the number of letter-anagrams of s (or in a simple implementation by $n!$, where $n = |s|$). Time T_1 is the time required to check whether the letter sequence can be parsed into dictionary words[2].

We expect that this method can be useful in scenarios where the dictionary size is large and the input size is small.

3.2 Anagram solving based on dictionary search

Let D be a dictionary of size d. Given input s, we can compute the multiset $S = \{\{s\}\}$. For each word in dictionary $w_i \in D$, we can also compute multiset $W_i = \{\{w_i\}\}$. To find anagrams in $\mathscr{D}$, we can use the following steps:

1. Filter dictionary: remove words from D, that cannot be formed from letters of s.
2. Search: for each remaining word w_i in D, remove letters of w_i from s, and recursively try to construct an anagram of the remaining letters.

Note that our goal is to enumerate all multisets of words that are anagrams to s. To avoid different word permutations, we need to use a sorted dictionary (in any order). In the recursive step, we only search for sub-anagrams in a dictionary that consists of word w_i, and words higher in the dictionary order (because words lower in the order were already explored).

During the algorithm, we need to check whether some dictionary word w can be a sub-anagram of string s. This means that we need to check if $\{\{w\}\} \subset \{\{s\}\}$. This multiset comparison can be done using vectors with letter frequencies, which requires $|A|$ comparisons. A well-known optimization[3] is to associate the letters of the alphabet with primes. Strings over the alphabet are then associated with products of the letter primes (with multiplicities), as their numerical fingerprints. That is, we use some function $\pi : A^* \to \mathbb{Z}$, with $\pi(x) = p_x$ with $x \in A$, and p_x a prime number associated with x. For longer strings, $\pi(x_1|x_2|\ldots|x_n) = \prod p_{x_i}$. It is easy to see that $\{\{w\}\} \subset \{\{s\}\}$ holds if and only if $\pi(s) = 0 \mod \pi(w)$ (numerical fingerprint of w is a divisor of the fingerprint of s).

The complexity of the method depends on the size of the dictionary d, on the contents of the dictionary, and on the length of the anagram n. If the average word length in the dictionary is l, we expect that we need n/l words to form an anagram of s. We thus have to search through the space of (approximately) $d^{n/l}$ strings.

4 Integer linear programming representation of the anagram solving

When considering a dictionary based anagram solver, we can notice that the only criteria for some string being a (sub-)anagram of another string are the involved letter frequencies. In this section, we present our new approach, similar to the approach used in algebraic cryptanalysis: we represent the problem in algebraic way and use specialized tools to solve the problem in its algebraic form. Our chosen representation is in the form of an integer linear programming instance.

Let $m = |A|$, and let $d = |D|$. We will represent dictionary word $w_i \in D$ with m-dimensional vector $\mathbf{w}_i \in \mathbb{Z}^m$, with $w_{i,j}$ coordinate equal to multiplicity of letter j in word w_i. Let $x_i \in \mathbb{Z}$ denote multiplicity of word w_i in multiset W, and let $y_j \in \mathbb{Z}$ denote multiplicity of letter j in $\{\{s\}\}$. Then it must hold that $y_j = \sum_{i=1}^{d} x_i w_{i,j}$, for each $j = 1, 2, \ldots, m$. Thus, we can write a system of linear equations

$$\left(\mathbf{w}_1^T \mathbf{w}_2^T \cdots \mathbf{w}_d^T\right) \mathbf{x}^T = \mathbf{y}^T,$$

with d unknowns x_i, and m equations. We are looking for an integer solution ($x_i \in \mathbb{Z}$) of this linear equation system with $x_i \geq 0$. This is known as an integer linear programming problem, ILP in short. Any valid solution $\mathbf{x}$ of the ILP problem represents a single multiset $W = \{\{w_1, w_2, \ldots, w_n\}\}$ that can be used to construct anagrams of s that are in $\mathscr{D}$.

Note that some words w_i in the dictionary might be an anagram of another word w_k. In this case $\mathbf{w}_i = \mathbf{w}_k$. In practice, we can represent both words with a single vector, and construct multiple anagram solutions by substituting $\mathbf{w}_i$ with each anagram word from the original dictionary.

[2] This is known as a Word-break problem, and has complexity $O(n^2)$ (Cormen et al., 2009).

[3] This method might be very old, but it is not clear to us who invented the method. We can refer to `https://stackoverflow.com/questions/13215789/comparing-anagrams-using-prime-numbers`, where the method is discussed.

Note that in terms of complexity, the linear programming approach is, in practical terms, equivalent to a dictionary search. This is because typically there is a significantly larger number of dictionary words (unknowns in the system) than alphabet letters (equations in the system). This means that the ILP solver needs to enumerate a large number of potential solutions word by word in a similar manner to a dictionary search.

5 Comparison of the methods

We have implemented both the dictionary based method and the ILP based method to test their behavior and efficiency on (relatively) small anagrams. The ILP method implementation uses ILP solver Gurobi[4], version 12.0.0. We focused on finding any feasible solutions with a systematic search, without improving the search with heuristics, and optimization through an objective function. The full settings of the solver are summarized in Table 1.

We have tested the methods on a specific dataset of n input sentences of increasing length k. The original input sentences were picked from the OANC corpus[5]. Thus, the sentences can be considered meaningful in the English language. All words in these sentences were present in the custom dictionary[6] of 89000 words.

For anagram solving, we reduced the dictionary size to a set of 10000 most frequent words with a length of at least 3. The dictionary reduction is necessary for performance reasons, and moreover it can simulate the case where we do not always have access to the full original dictionary. The similar reasoning holds for using short and special words, such as 'a', 'an', 'the': they have a detrimental effect on anagram solving, and can be preprocessed separately (guessed and removed from the input sequence).

During the experiment, we took the input sentence and ordered the letters in this sentence alphabetically. Both solvers (dictionary based, and ILP based) tried to find anagrams of this scrambled input.

A specific input sequence can produce multiple anagrams valid in the language given by the reduced test dictionary (see Figure 1) and can fail to produce the original sentence from the larger input dictionary. In the test settings, the probability of finding the original sentence was approximately 81%. To objectivize the comparison of methods, the efficiency of anagram solving was measured by the time required to find the original input sentence as an anagram of the input. The results of the experiments are summarized in Figure 2.

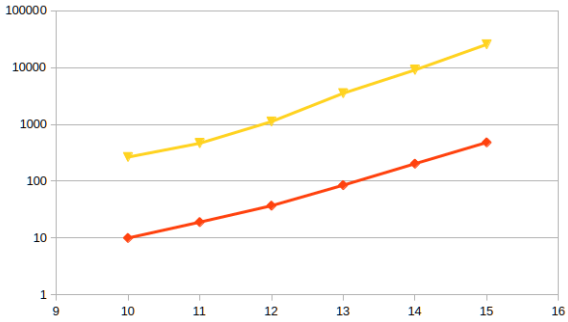

Figure 1: Number of anagrams obtained from an input of length k (x-axis). The bottom line represents the median value, and the top line the maximum value attained (dataset size 10000 inputs).

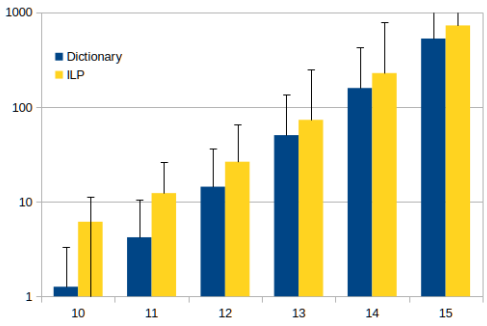

Figure 2: Average runtime of dictionary based vs. ILP based method (as a function of input of length k). Error bars are based on standard deviation.

As expected, we can see that the complexity of the anagram search increases exponentially with the length of the input k. The effect of the search method contributes only to a constant term. The dictionary based method is faster, but its advantage seems to be decreasing with increased input size k (see Figure 3). This is mainly a contribution of the more complex initialization of the ILP method. We have not tested potential advantages that could be obtained by ILP heuristics, and whether they can outperform dictionary based methods (on average).

[4] https://www.gurobi.com/downloads/gurobi-software/

[5] https://anc.org/data/oanc/download/

[6] https://people.sc.fsu.edu/~jburkardt/datasets/words/anagram_dictionary.txt

Parameter	Value	Description
Objective fn.	None	Solver searches for any feasible solution.
Heuristics	0	Heuristics turned off.
MIPFocus	1	Focus on feasible solutions.
PoolSearchMode	2	Systematic search.
PoolSolutions	MAX_INT	(No) Limit on the number of solutions.
Others	Default	

Table 1: The settings of the ILP solver Gurobi 12.0.0 used in the experiments.

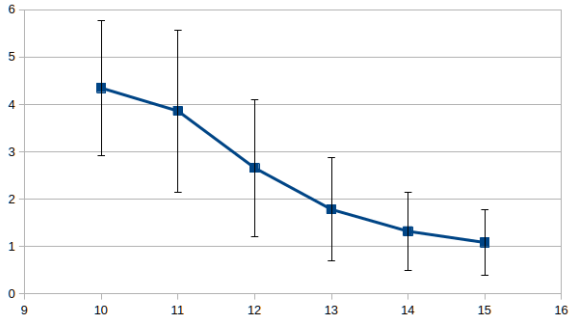

Figure 3: Average ratio of ILP solving time vs. dictionary based solving time (as a function of input of length k). Error bars are based on standard deviation. Values above 1 mean ILP method was slower by this constant factor.

6 Conclusions

In this paper, we have summarized some known methods that can be used for automated anagram solving, focusing on dictionary based anagrams. Anagram solving can be used to study old inscriptions, such as the mysterious anagram from the church of the Poor Clares in Bratislava (Antal and Zajac, 2024). Other potential applications are in the analysis of classical transposition ciphers and various transposition puzzles.

We have also introduced a new method of anagram solving that uses integer linear programming (ILP). Initial experiments show that this method is less suitable for solving short anagrams than dictionary method, due to the higher cost of ILP initialization. The ILP method compares more favorably with the dictionary based method with sufficiently long anagrams. Furthermore, we have only tested a base setting of the ILP solver, and have not used any heuristics or optimizations (that would require a specification of some objective function, and it is not clear what the function should look like).

A main difference between ILP based approach and dictionary based approach is the potential to extend the algebraic model with additional information that can be processed in the same way (using ILP solver). A potential extension can be provided by representing words with higher-dimensional Word2Vec (Mikolov et al., 2013) coefficients. A linear combination of words in a potential anagram would create a point in this vector space that can express a joint information about the whole collection of candidate words. The resulting coefficients (and/or their combinations) can then be targets of ILP constraints, and provide additional information improving the ILP efficiency. This approach, however, requires further research, as it is not clear how exactly these constraints should be constructed and used in anagram solving.

Acknowledgments

This research was supported by the project ”Artificial intelligence for encrypted handwritten document processing”, ”09I05-03-V02-00031” of call ”Support of research projects aimed at digitization of the economy in TRL levels 1-3”, Call. No. 09I05-03-V02, managed by Research Agency and funded by The Recovery and Resilience Facility of Slovak Republic.

References

Eugen Antal and Pavol Zajac. 2024. Can artificial intelligence solve the mysterious anagram from the church of the Poor Clares in Bratislava? In *Proceedings of the 7th International Conference on Historical Cryptology (HistoCrypt 2024)*.

Eugen Antal, Pavol Zajac, and Otokar Grošek. 2020. Diplomatic ciphers used by Slovak Attaché during the WW2. In *International Conference on Historical Cryptology*.

W.G. Barker. 1994. *Cryptanalysis of the Double Transposition Cipher*. A cryptographic series. Aegean Park Press.

F.L. Bauer. 2002. *Decrypted Secrets: Methods and Maxims of Cryptology*. Decrypted Secrets: Methods and Maxims of Cryptology. Springer.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. Introduction to algorithms (3-rd edition). *MIT Press and McGraw-Hill*.

Elonka Dunin and Klaus Schmeh. 2020. *Codebreaking: A Practical Guide*. Little, Brown Book Group.

H.F. Gaines. 1956. *Cryptanalysis: A Study of Ciphers and Their Solution*. Dover Brain Games & Puzzles. Dover Publications.

David Kahn. 1996. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner.

Richard M Karp. 2009. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, pages 219–241. Springer.

Donald E Knuth. 2011. *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Addison-Wesley Professional.

George Lasry, Nils Kopal, and Arno Wacker. 2014. Solving the double transposition challenge with a divide-and-conquer approach. *Cryptologia*, 38(3):197–214.

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. 2013. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT 2013)*, pages 746–751.

Masaaki Nishino, Sho Takase, Tsutomu Hirao, and Masaaki Nagata. 2019. Generating natural anagrams: Towards language generation under hard combinatorial constraints. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6408–6412.