A Minimal Algorithm to Find Anagrams Using Python

George Flanagin
University of Richmond
gflanagin@richmond.edu

Sunday 19th July, 2020

Abstract

Anagrams are fun and difficult. They are an excellent way to explore Python's data structures, and an excuse to develop a few new ones expressly for this purpose. This paper defines an efficient algorithm for finding all anagrams of a given collection of letters using the Python Standard Library for the base code, and the system dictionary as a source for allowed words.

Contents

List of Figures

1 Getting started

> *Of course just about every pronounceable combination of five letters has been used to spell or misspell something somewhere, at some point in history.*
>
> Donald E. Knuth, 2011
> *Combinatorial Algorithms, Part 1*, p. 519
> answer to Problem 25 in Section 7

1.1 Definitions

My interest in anagrams started late in life. An episode of *The Simpsons* named "Lisa's Rival" aired 11 September 1994, and in it Lisa goes to visit a new girl in the school, Allison Taylor. At Allison's house, Lisa is asked to join the Taylors in a specialized game of anagrams in which the names of famous people are rearranged to form descriptive anagrams: *Alec Guinness* becomes *genuine class* in the example. Lisa was overwhelmed.

To avoid offending or upsetting readers, I will be using my own name as the basis for the examples in this paper. Mercifully for the contruction of examples, my name consists only of common letters in English, and it has a useful proportion of vowels. Let's call the starting point the *target phrase*, or just the *target*.

To begin, we must have a definition of an anagram. Anagrams always appear in pairs. It makes no sense to say "*George Flanagin* is an anagram," without stating its partner phrase, the most descriptive of which may be *long fearing age*.

The technical properties of an anagram are:

1. Two phrases are anagrams of each other if they contain the same set of letters. In the case of *George Flanagin*, its anagrams must have exactly three *g*-s, two each of *a*, *e*, and *n*, and one each of *f*, *i*, *l*, *o*, and *r*. Fourteen letters.

2. Anagrams must consist of complete words in some dictionary. Exactly which dictionary is a point of æsthetics, as is whether one should consider single letter words such as `a` in English and Spanish, `I` in English, and `y`, `o` in Spanish, and other oddities like apostrophes and diacritics.

3. Additional rules are sometimes applied. Common constraints are:

   ✧ A maximum number of words in the anagrams of the original phrase.

   ✧ The elimination or inclusion of proper nouns in the derived anagrams.

✧ A lack of words shared with the original phrase.

Fortunately, none of the above boundaries is difficult to selectively enforce in a program.

1.2 The big picture of the search

At first, the search seems terribly complicated, and it is certainly not trivial. Everyone who has dealt with combinatorics knows how quickly the the number of subsets grows, 16383 in the case of fourteen letters, and the number of partitions, the fourteenth Bell number, is $27,644,437$.[1]

The constraint is the dictionary. The largest one generally used in computing is the list of words that ships as the spelling dictionary with Linux, and it has a mere 479826 words. In any search for anagrams except *the quick brown fox jumped over the lazy dog*, we will eliminate all the words that contain any letter not found in the target. In the case of our example, we can eliminate not only all the words that begin with $[b{-}d, h, j, k, m, n, p, q, s{-}z]$, but all the words that contain even one of those letters. Doing no programming at all, we can put the upper limit on the number of words to examine as 2777 if only lower case words are considered, and 5032 if we allow words that normally have capitals in English.[2] These are the upper bounds, because the figures are not derived by looking for the words that have no more of each letter than are found in the target.

For many programmers, the following is a useful abstraction to consider how we go about constructing both *an* anagram, and *all* anagrams. Consider the target phrase and bit-string of the same length. In Figure 1, we see two anagrams, *george+flanagin* and *long+fearing+age*, where each character is (1) or is not (0) used. For both of the anagrams, there is exactly one 1 in each column, and this tabulation is similar to the way that you might search for anagrams with a pile of Scrabble® letters. The equivalent big endian numeric represenations are shown in the right column.

2 Notation

Anagrams suffer from their essence being relatively easy to state in English, yet having no obvious representation in symbols. Given that programming languages are a type of symbolic representation, we will benefit from having a strong symbolic notation for anagrams, their parts, and their construction.

[1] The Bell numbers are OEIS sequence A000110, `https://oeis.org/A000110`

[2] These numbers are the results of the two naïve regular expression searches of the Linux spelling dictionary with `^[aegnfilor]\+$` in case sensitive and case insensitive modes.

	g	e	o	r	g	e	f	l	a	n	a	g	i	n	
george	1	1	1	1	1	1	0	0	0	0	0	0	0	0	16128
flanagin	0	0	0	0	0	0	1	1	1	1	1	1	1	1	255
long	1	0	1	0	0	0	0	1	0	1	0	0	0	0	10320
fearing	0	1	0	1	1	0	1	0	1	0	0	0	1	1	5795
age	0	0	0	0	0	1	0	0	0	0	1	1	0	0	268

Figure 1: The relationship between anagrams and bit masks

In Figure 2 we can see the basic notation that is invented for the purpose of this discussion. The other symbols used, such as assignment, absolute value, and non-anagram set operations, are expressed in conventional notation, and will mean what you expect them to.

The notation suggests that our Python representation will need all of these operators/operations:

1. Sorting the letters of a string to make an new string of the same length.
2. The always useful "partial ordering" operator, $\leq$, so that we can determine if words can be used to make an anagram of the target phrase.
3. A method to combine strings beyond concatenation, and a method to remove letters from a string.
4. A method to filter the useful words in a dictionary to make a new, subset dictionary.
5. A method of bookkeeping to allow us to track the anagrams that have been found, and avoid searching the same path twice.

In fact, these operations lead us directly to a discussion of the algorithm to find all anagrams.

Symbol	Use and meaning
w, w_n	word or words from a dictionary.
$\mathfrak{S}$, $\mathfrak{T}$	phrases for which are finding anagrams.
$\overrightarrow{w}$, $\overrightarrow{\mathfrak{S}}$	representations of w and $\mathfrak{S}$ where the letters have been sorted. In the programming section, we will be using a standard lexical sort, but this is unimportant as long as the same sort-order is used throughout. If w is `loaf`, then $\overrightarrow{w}$ is `aflo`.
$w \leq \mathfrak{S}$	This expression is true iff w can be constructed from the letters in $\mathfrak{S}$. For example `foal` $\leq$ `georgeflanagin`, and `foal` $\leq$ `loaf`. Note that this expression is true or false without regard to whether the letters in each term have been sorted.
$w_1 \oplus w_2$	The result is a collection of all the letters in the two words, without preserving the order of the letters in each word. The $\oplus$ operator was chosen over + because the bare plus sign is used as a string concatentation operator in many programming languages, including Python.
$w_1 \ominus w_2$	As with the *oplus* operation above, except that we are removing all the letters in w_2 from w_1. This operation is only defined (or meaningful) iff $w_2 \leq w_1$, otherwise in the grammar of anagrams (anagrammar?), the statement is like dividing by zero.
$w_1 \odot w_2$	This expression is true iff w_1 and w_2 are anagrams, so expanding on the above examples, `foal` $\odot$ `loaf` is true, as is: (`long` $\oplus$ `fearing` $\oplus$ `age`) $\odot$ `georgeflanagin`.
r, r_n	r is for remainder, so when we perform a $\ominus$ operation, the result will be a value expressed as an r, so $r_1 = \mathfrak{S} \ominus w_1$
$\mathfrak{D}$, $\mathfrak{D}'$, *etc.*	Throughout, we will use $\mathfrak{D}$ to represent the core dictionary, and $\mathfrak{D}'$ and $\mathfrak{D}''$ to represent derived dictionaries such that $\mathfrak{D}'' \subset \mathfrak{D}' \subset \mathfrak{D}$, in other words, a filter.

Figure 2: Symbols used to discuss anagrams

3 Algorithm

> *Homer Jay, how do you keep your hair so rich and full?*
> *Lather, rinse, and repeat. Always repeat.*
>
> Homer Simpson
> D'oh-in' in the Wind
> 15 November 1998

Given the small number of qualifying words, and the vast available memory combined with the computational abilities of even bottom-self computers, finding all anagrams of a phrase could be done by brute force. That approach is not very satisfying. Instead, we are searching for elegance and comprehension, two nouns that often appear side-by-side.

3.1 Pruning the dictionary

Our first step is the elimination of all the words that are made from incompatible collections of letters. In our grammar, we seek to construct

$$\mathfrak{D}' := \{w : w \leq \mathfrak{S} \wedge w \in \mathfrak{D}\} \tag{1}$$

This is often a small collection of words, and we know that

$$\forall w : |w| \leq |\mathfrak{S}| \tag{2}$$

3.2 Trying the words

It makes sense to start with the longest words in $\mathfrak{D}'$, a fact that will guide us when we start the programming in the next section. Each word in the dictionary will be subtracted from the target phrase, and will leave a complementary set of remainders.

$$\mathfrak{R} = \{\forall w : \mathfrak{S} \ominus w\} \tag{3}$$

Equation 3 is well suited to the list comprehension construct in Python.

At this point, we should also take note of the one-to-many relationship between $\overrightarrow{w}$ and the dictionary words w. Words that are anagrams of each other share the same sorted representation of their letters, so in a key-value look up table, if the keys are of the form $\overrightarrow{w}$, then they must support a list (tuple) of one or more w-s as the values.

While programming this data structure takes us into the shallow end of the pool of rolling our own data structures later on, it does mean that we do not need to try any of the words in the tuple to make an anagram, we need only concern ourselves with the sorted key. Consider this specific case: $acer \longrightarrow \{acre, race, care\}$

The sorted representation, *acer*, is meaningless. But we can freely substitute any of the three English words in anagrams that contain one of them. Whether the collection of real words associated with a key is one or more than one, we only need to bother with the key. Thus, Equation 3 becomes the more manageable expression seen here:

$$\mathfrak{R} = \{\forall \overrightarrow{w} : \mathfrak{S} \ominus \overrightarrow{w}\} \tag{4}$$

We cannot neglect the fact that $\forall w : w \odot w$, or in plain English, every word is an anagram of itself, so we must check to see whether each remainder is a key in the dictionary and a complement of some other key in the same dictionary. So before any recursive decent begins, we must check for the "two word" solution.

3.3 Recursion and bookkeeping

At this point, we have the algorithm reasonably well in mind, if not in hand. We take our collection of remainders and derive $\mathfrak{D}''$ from $\mathfrak{D}'$, and reapply the testing of all the keys in $\mathfrak{D}''$ to $\mathfrak{R}$. Practially speaking, the dictionary rapidly becomes small.

Additionally, we can exploit the fact that we are keeping track of the $\overrightarrow{w}$ terms as we go to ensure that we do not test them more than once, and this is where bookkeeping enters the picture. It seems fairly natural to think of this as a forest of n-ary trees, where the root node of each is a $\overrightarrow{w}$ term. For $\overrightarrow{w}$ terms that offer no completion (*i.e.*, dead ends) we can saw these to the ground, and experience has shown that dead ends will constitute the majority of the $\overrightarrow{w}$ terms we try.

4 Programming

This paper is being written in 2020, so the programming is definitely done using Python 3. At this time, I am using Python 3.8, although I do not think any of the features that first appear in Python 3.8 (such as the "walrus operator," `:=`) are used in the code that appears here.

This is a paper about anagrams, Python's data structures, and rolling a few of our own data structures. It is not about PEP-8 style, sane exception handling, type hints, nor how to organize code modules in the project. With that warning, let's get started.

4.1 Style of the dictionary

The familiar `/usr/share/dict/linux.words` file is based on the Webster's Second International Dictionary. It has a number of entries we do not need as its primary use is in spell-check. There are words with punctuation, and acronyms that are all caps. Additionally, it has 1420 words of three letters, 25199 words that start with a capital letter, and with 10230 words of five letters, it far exceeds Knuth's well established list of 5757 five letter words that is a corpus in the Stanford University Graph Base.[3]

If you turn your attention to `dictbuilder.py` you can get a feel for the approach taken to support anagrams. I have chosen to eliminate a large number of the dictionary entries by reading it this way which rids us of duplicates, capital letters, and punctuation all at once, and I have chosen to supply an explicit list of 27 two letter words rather than the 160 in the dictionary. Feel free to adjust the code to suit your use.

`dictbuilder` creates two dictionaries.

1. a `dict` (dictionary) whose keys are the words we have in some way read from the dictionary file, and whose values are the sorted strings of the letters in the word.

   $$w \longrightarrow \overrightarrow{w}$$

   Rather arbitrarily, this is termed the *forward dictionary*, and in the code dictionaries of this type are usually referred to by the sybolic name `f_dict`.

2. a `dict` whose keys are the sorted strings from above, and whose values are a set/tuple of all the words from the dictionary that can be made from this string of letters.

[3] `https://www-cs-faculty.stanford.edu/~knuth/sgb.html` Both the page and Knuth's book are well worth exploring.

$$\overrightarrow{w} \longrightarrow (w_0, w_1, w_2, ..w_n)$$

This is termed the *reverse dictionary*, and it is associated with the symbolic name `r_dict`.

From the standpoint of use in dictionaries, it is required that both the keys and values be hashable, because the values become the keys in the reversed dictionary. Consequently, we cannot use a sorted `list` of letters; it must be a `str` or a `tuple`. Ordinary strings are the most convenient, particularly when printing the results.

4.2 Going from words to sorted strings

In the Python Standard Library there are wonders, and one of the ones we will be using is `collections`, and within it we will start with the handy `Counter` class. As the documentation states, `Counter` is a type of `dict`.[4]

Referring to Figure 2, we can see that

```
>>> S = 'george flanagin'
>>> sorted_S = str(sorted([ _ for _ in S if _ != ' ' ]))
>>> counted_S = collections.Counter(S)
>>> counted_S
Counter({'g': 3, 'e': 2, 'a': 2, 'n': 2, 'o': 1, 'r': 1,
    'f': 1, 'l': 1, 'i': 1})
```

`Counter` defines a number of operations that are very close to ideal for our use in abstracted algebra to deal with anagrams. This is good in that we get a head start, but we do need to put in a bit of work to customize the `Counter` for our purposes. The most direct route is the exploitation of Python's underlying object model.

`Counter` gives us a useful iterator named `elements()` that we can pass directly to the `sorted` builtin.

```
>>> sorted(counted_S.elements())
['a', 'a', 'e', 'e', 'f', 'g', 'g', 'g', 'i', 'l', 'n', 'n', 'o', 'r']
```

[4]The documentation referred to here and throughout this paper is the collection of web pages at `docs.python.org`. It is searchable, well written, and accurate. You should not only use it, you should prefer it.

As a subclass of `dict`, `Counter` has specialized the `update()` method use the plus (+) operator, and will do exactly what we require to implement the method in our notation written as $\oplus$. Unfortunately for our work, the `subtract()` function allows negative quantities,[5] which means we will need to modify it slightly in our subclass for it to be an implementation of $\ominus$.

Figure 3 code (with most comments removed for bevity) that accomplishes the following in a class named `CountedWord`:

1. As a subclass of `Counter`, we get to use the builtin methods.
2. In keeping with the spirit of the `Counter` implementation, we have superseded the meanings in the original class, so that $a - b$ does not modify a, and $a- = b$ is provided for the cases where that is desired.
3. We have provided a `__str__()` operator that returns the sorted string of the characters in the counter. This is generally the most useful case if we want to, in effect, use a `CountedWord` as a key.

[5] The subclassed `update()` also allows for negative quantities, but when we are adding objects whose count is greater than zero, there is no risk of getting a negative result.

```
@total_ordering
class CountedWord(Counter):
    """
    Each word/phrase corresponds to one CountedWord representation of it.
    For example, CountedWord('georgeflanagin') is 'aaeefgggilnnor'. However,
    the same CountedWord may be a representation of many different words.

    The operators allow us to write code that is somewhat algebraic.
    """
    def __init__(self, s:str):
        """
        Add one class member, the as_str, which is a the word
        represented as a
        """
        Counter.__init__(self, s)
        self.as_str = "".join(sorted(self.elements()))


    def __eq__(self, other:Union[CountedWord,str]) -> bool:
        """
        if CountedWord(w1) == CountedWord(w2), then w1 and w2 are
        anagrams of each other.  For example CountedWord('loaf') ==
        CountedWord('foal').
        """
        if isinstance(other, str): other = CountedWord(other)
        return self.as_str == other.as_str


    def __le__(self, other:Union[CountedWord,str]) -> bool:
        """
        if shred1 <= shred2, then shred1 is in shred2
        """
        if isinstance(other, str): other=Counter(other)

        # Note that there are no zero-counts in the Counter's
        # dict. So all the v-s from self will be > 0.
        return all(other.get(c, 0) >= v for c, v in self.items())


    def __sub__(self, other:CountedWord) -> CountedWord:
        if isinstance(other, (str, Counter)): other = CountedWord(other)
        if other <= self:
            x = copy.copy(self)
            x.subtract(other)
            x.__clean()
            x.as_str = "".join(sorted(x.elements()))
            return x
        else:
            raise ValueError('RHS is not <= LHS')


    def __add__(self, other:CountedWord) -> CountedWord:
        if isinstance(other, (str, Counter)): other = CountedWord(other)
        x = copy.copy(self)
        x.update(other)
        x.as_str = "".join(sorted(self.elements()))
        return x


    def __clean(self) -> None:
        zeros = [ k for k in self if self[k] == 0 ]
        for k in zeros:
            self.pop(k)


    def __str__(self) -> str:
        """
        The contents, sorted, and as a string.
        """
        return self.as_str
```

Figure 3: Example code for derived `Counter` class