An Exaustive Anagram Generator in Python

(Using Only Integer Math and Trees)

George Flanagin
University of Richmond
`gflanagin@richmond.edu`

Sunday 17th September, 2023

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, or some other list of words, as a source for allowed vocabulary.

This program is the Anagrammar. Its source code and this documentation may be found at `github.com/georgeflanagin/anagrammatic`

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 later in life, at 37 years old. An episode of *The Simpsons* named "Lisa's Rival" aired 11 September 1994, and in it Lisa visits 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 show's example. Lisa was overwhelmed.

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

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, and the one most suited to Lisa Simpson's game may be *long fearing age.*

The technical properties of an anagram are:

1. Two phrases are anagrams of each other if and only if all the letters of one phrase are present in the other, the same counts for each letter present. 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. Each phrase in a pair of anagrams must consist of one or more 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`, `e`, `u` in Spanish. For the purpose of this paper, we will ignore diacritics.
3. Additional rules are sometimes applied. Common constraints are:
   - ✧ A maximum or minimum number of words in the anagrams of the original phrase, and the related metric, minimum word length.

- The elimination or inclusion of proper nouns in the derived anagrams.
- A lack of words shared with the original phrase. More discussion of this constraint lies ahead.

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 (*i.e.*, $2^{14} - 1$), 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 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.

2 The 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

[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.

2.1 Building a dictionary

The algorithm in Anagrammar is all integer math, providing simplicity and speed. Each letter in the alphabet is equated with a prime number, and it makes sense to use the first 26 prime numbers, $2, 3, ..101$. The order is unimportant, but choosing the order

```
eariotnslcudpmhgbfywkvxzjq
```

tends to give the smallest numeric values for the products that represent the words. The more familiar `etaon...` order of frequency is based on normal text in English, but we are concerned with getting our text from a dictionary where each word occurs only once, and the above sequence represents the frequency of letters of the words in a dictionary. Raw experimentation suggests that the composite numbers representing words have a reduced bit length of 15 to 20%, which is substantial.

```
primes26 = (2, 3, 5, 7, 11,
    13, 17, 19, 23, 29,
    31, 37, 41, 43, 47,
    53, 59, 61, 67, 71,
    73, 79, 83, 89, 97,
    101 )
primes = dict(zip("eariotnslcudpmhgbfywkvxzjq", primes26))

def word_value(word:str) -> int:
    return math.prod(primes[_] for _ in word)
```

[*Note:* In the remainder of this document, all numeric values shown are based on the above mapping of letters to integers. — *end note.*]

The sample phrase, *george flanagin*, evaluates to a staggering 836654888347740 using the above mapping, but it is still only 49.57 bits, or 3.54 bits per letter.

The fundamental theorem of arithmetic ensures that each word is represented by a unique numeric value. However, many different words may map to the *same* value, for example, `care`, `acre`, and `race` contain the same letters. If one of them is a part of an anagram, then all of the others are also a part of that anagram.

A dictionary consists of a mapping, or in Python's language, a `dict`, in which the keys are the integers and the values are a set of words.

One additional convenience of this method is that we are only concerned with the integer keys — there is no need to examine the words until the end of the process. This cuts down considerably on the memory requirement for the running program.

2.2 Select candidates from 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 a collection of values that divide the value of the target phrase. This is more easily represented in Python than in conventional set notation from mathematics.

In fact, it is so simple that there is no real benefit to making it a function except for the clarity of notation. The `filter` argument is the phrase (or sub-phrase . . . this is a recursive algorithm) we are searching with the `candidates`, a list of potential factors.

```
def prune_dict(filter:int, candidates:tuple) -> tuple:
    return tuple(k for k in candidates if not filter % k)
```

Given that this is a common operation in the Anagrammar, it is worth some future experimentation to see if it can be squeezed down into fewer instructions.

2.3 Trying the words

It is useful to think of a target phrase and all its anagrams as an n-ary tree,[3] in which the root is the target phrase, and each path from the root to a leaf is an anagram. Using our chosen phrase, `georgeflanagin`, we can think of the relationship represented like this:

```
root ⟶ long ⟶ fearing ⟶ age
   \⟶ nine ⟶ ear ⟶ golf ⟶ gag
```

The obvious first step is to try all the words against the original phrase and see what is left after the division. In our n-ary tree, there will be at most 2777 branches from the root because there are 2777 factors. Ugly? Yes, but we only need to build it rather than plant it in the garden and admire it, and we will do our best to prune as we go.

[3] n-ary trees are not a native citizen of Python, and the discussion of the implementation of trees is discussed in Section 3.

2.4 Recursion and Bookkeeping

Finding anagrams is a recursive process: if a word is a component of an anagram for a given phrase, then all of the word's anagrams are also elements of the phrase's set of anagrams. For example, we can break apart `fearing` into `near-fig`, and substitute to have `long-near-fig-age`.

Let's number the first connection to the root of the tree as edge zero, and let's sort the factors that represent the words in ascending order. A key point is that the set of factors is not merely *well ordered*, but has *strict total ordering*: the factors are distinct, and they are all comparable with <.

In our programming, we can refer to the edge numbering as the *depth* because computer science trees are usually construed to grow from a root in the sky toward the ground. Edges that branch from the initial test of all the factors will be numbered 1, and so on.

In our representation, there is some least factor that may or may not correspond with the "shortest word" we are considering. Our representation has moved beyond the ambiguities of having several three letter words and a reliance on a lexicographic ordering to differentiate between them. We no longer need to test twice, once for length and once for alpha order.

In Python, we can represent the sorted factors with this line of code, where `phrase_v` is the value associated with the term we are trying to decompose into an anagram of dictionary words.

```
factors = tuple(sorted(prune_dict(phrase_v, factors)))
```

We can test the factors in a completely unoptimized for loop safely using integer division because the factors are all known to divide `phrase_v` evenly — that's what it means to be a factor.

```
for factor in factors:
    residual = phrase_v // factor
```

The idea of recursion is simple enough to present as repeating the above operation with the `residual` and its factors that will be a proper subset of the factors of `phrase_v`. Before we can recurse, we must keep track of where we are in the tree.

Lather, rinse, repeat is a risky process in programming because it requires us to ensure we do not do the recursive operation infinitely, and there are many things that will trip us up along the way. Let's use the name `find_words` for the recursive function. For each call to `find_words` we need to know the level of recursion; this way we will add edges to the correct level of the tree. This gives us a function signature of

```
def find_words(phrase_v:int,
    factors:tuple,
    depth:int=0) -> SloppyTree:
```

with the recursive call having the form:

```
t = find_words(residual,
    tuple(_ for _ in factors if _ < residual),
    depth+1)
```

Not every `residual` needs to be recursed or the program would not make progress. This fact implies that we must have good logic to know when to stop looking and to recognize an anagram when we find one. The initial decision is simple.

1. Is the `residual` less than the smallest factor? If so we have reached the end, and the edge leading to this value can be removed.
2. Is the `residual` in the list of factors? This is the final element of one of the anagrams for the original phrase.
3. If it is neither of the above, then we recurse.

The trilemma above misses a common situation that will hide some anagrams from the search. Suppose that the residual is the value associated with a word like *boxcar*. If we abandon the search at this point and declare *boxcar* to be the leaf node, we will not find the factors that represent *box* and *car*.

The solution is to try the words in increasing numerical value: both *box* (53867) and *car* (435) will have been tried before *boxcar* (23432145). If we use this approach at each depth, we can be assured we will not miss any occurrences down the path toward the leaves.

This leads us to another way to avoid redundancies in the search. Reconsider the *boxcar* case. If at level 0 we go through the factors in increasing order, then *car* will have been seen at level 0 long before we find *boxcar* as a residual near the leaf of some other path from root. This suggests that as we can save a good bit of work by checking if a factor has already been examined at level 0.

As we work our way through the factors at level 0, we can add each one to a `set` that can be checked at other levels. Here again, we must be careful. It is certainly possible that an original phrase might contain the same word twice: *I cannot open the can.* If *can* (1479) is the current level 0 word, then the simple strategy above will fail to find anagrams in which the same factor occurs twice. To solve the problem, we introduce a global that represents the current factor at level 0, or more intuitively, the root of our current branch in the tree.

```
if not depth:
    seen_factors.add(factor)
    current_root = factor
```

3 Trees in Python

When it comes to native data structures, Python has quite a few to choose from, and all are useful. Along with `list`, `dict`, and `tuple`, there are the many extended data structures in the `collections` module. Nevertheless, any kind of tree is absent, and that may well be because there are so many different kinds of trees, enough that Donald E. Knuth has been giving an annual lecture on new trees for the entire millennium, minus the Covid years of 2020 and 2021.

Anagrammar uses an n-ary tree implemented as a `class` derived from `dict`. Unlike trees that require building nodes and linking them to other nodes, this tree builds nodes automatically when they are first mentioned, and we have given it the name `SloppyTree`™. `UglyTree` might have been a better name, but the chosen name allows us to have a Python module with the name `slop.py`.

`SloppyTree` is directly derived from `dict`, and the automatic node creation is achieved by having the `__missing__` function create a key whose value is an empty `SloppyTree`. Because subscript notation of deeply nested `dict`s can become tedious,

```
t[’key1’][’key2’][’key3’] = 6
```

the `__getattr__` and `__setattr__` are added so that the above line may be written as:

```
t.key1.key2.key3 = 6
```

In fact, there is no need to set a value to create the node. If you are unsure of the value at the time the node is created, an *expression* like

```
t.key1.key2.key3
```

is really a *statement* (!) that creates the cascade of needed `dict`s and their keys. It is equivalent to:

```
t = SloppyTree()
t['key1'] = SloppyTree()
t['key1']['key2'] = SloppyTree()
t['key1']['key2']['key3'] = SloppyTree()
```

`SloppyTree` has a few additional features. An iterator for a `dict` only iterates over the keys, but we need to see the values, and be able to distinguish between leaf and non-leaf nodes. For its use in Anagrammar, we need an easy way to show all the anagrams, which is to say, each path from the root to each leaf. Therefore, we have an additional iterator/generator for these, through the function `SloppyTree.tree_as_table()`.

George Flanagin, Provost Office
Academic Research Computing

Phone: +1.804.287.6392
Address: Richmond Hall, Office 104
114 UR Drive
University of Richmond
Richmond, VA 23173
Email: gflanagin@richmond.edu
ORCID 0000-0002-2084-5831