An Algorithm to Find Anagrams Using Python

George Flanagin
University of Richmond
gflanagin@richmond.edu

Saturday 10th July, 2021

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.

This program is the ANAGRAMMAR. Its source code and this documentation may be found in https://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 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`, `e`, `u` 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, 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.

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 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 being relatively easy to define in English, yet having no obvious representation in symbols. Given that programming languages are a type of symbolic

[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

representation, we will benefit from having a strong symbolic notation for anagrams, their parts, and their construction.

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 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 an anagram that contains one of them. Whether the collection of real words associated with a key has one or more than one element, 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]

3.3 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 $\mathfrak{R}$. 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.

[3] In fact, if the original phrase is shorter than the longest words in the dictionary, we must include the "one word" solution. For example, if the target phrase is *gimp count*, there is a one word solution to be found: computing.

4 Programming

This paper is being written in 2021, so the programming is 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 on GitHub.

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.[4]

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 in a way that 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.

[4] `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`. *[Note:* The container is represented as a `tuple`, but it is also a `set` because each element occurs exactly once. *— end note.]*

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`.[5]

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. It is good 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']
```

[5] 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 using the plus (`+`) operator, and it 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,[6] which means we will need to modify it slightly in our subclass for it to be an implementation of $\ominus$.

The code in Figure 3 (with most comments removed for bevity) accomplishes the required changes 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 when we want to use a `CountedWord` as a key.

4.3 N-ary trees in Python

GitHub is filled with trees for Python, and the reason is very likely that Python (as of 3.8) has no native tree in its standard library. For anagrams, we need nothing complex like self-balancing red-black trees, nor even B-trees. We need a flexible N-ary tree.

It is fairly easy to construct one starting from the idea behind the `defaultdict` from the `collections` module, although it does not quite work for us as delivered. We need to support all the following abstractions:

1. From graph theory fundamentals, we know that a tree is a data structure in which there is exactly one path from any node to another node. This implies that a linear sequence is also a tree, although not a very ornamental example.
2. An important property of a tree can be derived from the definition: if we cut a tree into two pieces by removing the unique connection between any two nodes, the resulting two data structures are each trees. To look at it from the opposite direction (the direction more useful for a recursive search like finding anagrams), we can take any two trees and perform a grafting operation at a node to join them into a single tree.

[6] 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

The needed operations in Python can be accomplished with the `dict` because `dict`s may contain `dict`s as members. From a programming standpoint, this operation needs to be as intuitive as possible if we are to avoid common programming mistakes.

The single data structure for trees in this program is known as the `SloppyTree`, a name that speaks for itself. It is directly derived from `dict`, with the following behaviors modified:

- ✧ `SloppyTree` provides a `__missing__` function that creates a new key with the default value of an empty `SloppyTree` in the case where the key is not found. The native `dict` function raises a `KeyError` when elements are not found; instead we want to automatically add them. This behavior is also found in `defaultdict`.
- ✧ Consistent with `class`-like behavior, `SloppyTree` provides the trio of member access operations: `__getattr__`, `__setattr__`, and `__delattr__`. These operations are not required for the case of anagrams, but `SloppyTree` has other uses.
- ✧ Consistent with our desire to write as little code as possible, `SloppyTree` has a `__str__` operator that invokes the `pprint.pformat` function to write out the contents in a way suitable for review.

As much as we benefit from the `CountedWord` class, it is unfortunate that they are not hashable, and therefore cannot be keys for the `SloppyTree` nor any other `dict`-derived type. This is why the `CountedWord.__str__` operator returns the extra class member `as_str` — we really do not want to be constantly rebuilding them.

5 Using the Anagrammar

5.1 Command line parameters

Launching the program produces a familiar style of command line usage message:

```
usage: anagrammar [--help] [--cpu CPU] --dictionary DICTIONARY [--min-len MIN_LEN]
                  [--no-dups] [--none-of NONE_OF] [--order {0,1,2}] [-v {0,1,2,3}]
                  phrase [phrase ...]
anagrammar: error: the following arguments are required: phrase
```

A polite inquiry with `--help` provides slightly more information as shown in Figure 4. Unfortunately, it is not possible to provide all the information needed in just a line or two. Let's examine the options in alphabetic order.

```
positional arguments:
  phrase                The phrase. If it contains spaces, it must be in quotes.

optional arguments:
  -h, --help            show this help message and exit
  --cpu CPU             Set a maximum number of CPU seconds for execution.
  --dictionary DICTIONARY
                        Name of the dictionary of words, or a pickle of the
                        dictionary.
  --min-len MIN_LEN     Minimum length of any word in the anagram
  --no-dups             Disallow words that were in the original phrase.
  --none-of NONE_OF     Exclude all words in the given filename.
  --order {0,1,2}       Key ordering: 0: random, 1:shortest first, 2:longest first
  -v {0,1,2,3}, --verbose {0,1,2,3}
                        Be chatty about what is taking place -- on a scale of 0 to 3
```

Figure 4: Anagrammar help text

`--cpu` This option is primarily useful to prevent runaway searches when the phrase being anagrammed is capable of making a large number of anagrams. Given that the algorithm is tail-recursive and single-threaded, the processor time correlates well with the number of branches in the tree.

`--dictionary` Half the fun in this program is changing out the dictionary of allowed words, and the other half is changing the original phrase. Only the non-suffix part of the filename needs to be given, and the Anagrammar will accept the name you give and assume that the pair of dictionary files is named `.forward` and `.reversed`

`--min-len` This parameter is the most significant one in changing the results and altering the execution time. The default value is `2`, which is too low for the most interesting results.

Note what happens when you allow a value of `1` with the input phrase "george flanagin." Using the two one-letter words of English, this amounts to running the algorithm twice with `a` and `i` extracted in turn on the remaining 13 letters.

Note that there are problems at the other end, as well. For example, suppose the input phrase has 18 letters, and the words must be six letters long. With these initial conditions, the only possible combinations are three six-letter words or two words where the pairs are `(9, 9)`, `(8, 10)`, `(7, 11)`, and `(6, 12)` letters.

`--nice` Niceness is available for user programs, and the higher the value, the more willing the OS will be to execute another program. On the other hand, it may give the program a longer quantum and fewer interruptions when it is running.

`--no-dups` When this parameter is present and the input phrase is written as more than one word, *i.e.*, *george flanagin* rather than *georgeflanagin*, none of the words are allowed in the resulting solution set of anagrams.

`--none-of` The value of this parameter is a string that is interpreted as a filename containing white-space delimited words to be removed from consideration. The methods for removing entries from a pair of `dict`s with thousands of entries is a kludge, and this option does not exist as an option to creating a custom dictionary. See Section **??** for information about constructing custom dictionaries.

`--order` This parameter exists primarily to do demonstrations of the importance of the search order for anagrams. The options are:

0 random order, created by trying the possible keys in an order provided by the Python builtin, `random.sample`. The results are not repeatable, nor can the sample method be tuned.

1 shortest keys first. This is the "correct" operation.

2 longest keys first. This method produces only a subset of the possible anagrams, but it is quite useful to illustrate the importance of search order.

`--verbose` At this time, only `--verbose 3` has a profound effect on output. The high setting of verbosity creates an effective flow trace showing the keys being tried and exhausted, and the recursions noted.

5.2 Example outputs

5.2.1 Statistics

As mentioned in Section 1.2, the number of anagrams can become large, quickly, and for a variety of reasons – the dictionary used, the letters making up the phrase. Therefore, the example that is used in this paper is a little contrived.

First, let's take a look at the standard output that is created while the program is running in its standard mode as shown in Figure 5. Note that the results will vary widely across operating systems and processors.

On the line that precedes the table, we can read the size of the vocabulary that has qualified for use in the search. In this example, there are 880 keys (sorted strings) that represent 1054 words taken from the dictionary. The output is updated every 100 branch evaluations as the program runs.

```
Initial pruning: 880 keys representing 1054 words.

D | branch |  dead  |  user  |  sys   |  page  |  I/O  | WAIT | USEDQ |  Tails  |
  | evals  |  ends  |  secs  |  secs  | faults |  sig  |  sig |       |         |
--+--------+--------+--------+--------+--------+-------+------+-------+---------|
  1    26017     8186     2.18     0.07    32829       0      7   1269      8640

26017 branches in the tree. 8186 dead ends. Max depth 3.
```

Figure 5: Standard program output.

The left-most column shows the current recursion depth, and it is more for evidence of operation than any practical use. It will always be `1` when the program terminates. The other parameters are more significant.

branch evals The count of the number of attempts to add a branch to the tree.

dead ends The number of failed branches during tree construction.

user secs The number of seconds the program has been executing. This value added to the `sys secs` value gives the "clock time" of operation.

sys secs The number of seconds the system has been performing tasks on behalf of the program.

page faults The number of times the program has had to request a page from memory. Typically, this number gains most of its value when the dictionaries are read into memory, and it grows only slightly as the tree of results grows.

I/O sig The number of times the program has stopped to perform I/O with storage (disc), as opposed to memory.

WAIT sig The number of times the program has been told to wait by the operating system.

USEDQ All operating systems allow a program to run for a specific amount of time, known as the *quantum*, at which point the scheduler interrupts to be sure there is nothing more important for the current core to do. The number of interruptions is highly dependent on the CPU and the OS, but the ratio of `USEDQ / (WAIT + I/O)` is a clear indication of the degree to which the task is "CPU-bound."

Tails The final remainder is called a *tail*, a suitable name because the algorithm is *tail recursive*. At the end of each failure to find an anagram, the terminating event is a

```
anagram --min-len 4 --dict mit10000 "sample phrase"

['sample', 'phrase']
Initial pruning: 86 keys representing 98 words.

 D | branch |  dead  |  user  |  sys   |  page  |  I/O  | WAIT | USEDQ |  Tails  |
   | evals  |  ends  |  secs  |  secs  | faults |  sig  |  sig |       |         |
---+--------+--------+--------+--------+--------+-------+------+-------+---------|
  1      119       95     0.11     0.03     4935       0      6    103       102

119 branches in the tree. 95 dead ends. Max depth 3.
{   'maple': 'phrases',
    'peas': {('arms', 'mars'): 'help'},
    'phrase': 'sample',
    'sphere': 'plasma',
    ('males', 'salem', 'meals'): 'perhaps',
    ('phases', 'shapes'): 'palmer'}
```

Figure 6: Anagrams of *sample phrase* using MIT 10000 word dictionary

tail that makes no word in the dictionary, or a tail that has been discovered to not be decomposable into two or more words in the dictionary. To avoid traversing the same tails more than once, each new tail is added to a Python `set`. The number shown in this column is the final total.

5.2.2 Anagrams of a sample phrase

Using *sample phrase* as the sample phrase, and setting the minimum length to 4, we get interesting results when using the full Linux spelling dictionary. The Anagrammar's normal output is shown in Figure 7. The less interesting, but more easily understood collection of anagrams from the MIT 10000 most common words is shown in Figure 6.

The tree is printed so that the anagrams may be located by looking at the lines of the report. Looking at Figure 6, we see that *maple* and *phrases* is an easy anagram, the kind that Lisa Simpson posited for "Jeremy Irons" ... "Jeremy's Iron," missing "minor jersey."

The next line features a three word anagram, *peas*, *help*, and either of the self-anagrams, *arms* and *mars*. Since we did not exclude the original terms, they appear in the third line. Contrast this with the penultimate anagram in the list constructed from the larger dictionary. It shows that *phrase* has two self-anagrams in the dictionary, *shaper* and *seraph*.

```
anagram --min-len 4 --dict words "sample phrase"

['sample', 'phrase']
Initial pruning: 262 keys representing 448 words.

 D | branch |  dead  |  user  |  sys   |  page  |  I/O  | WAIT | USEDQ |  Tails  |
   | evals  |  ends  |  secs  |  secs  | faults |  sig  |  sig |       |         |
---+--------+--------+--------+--------+--------+-------+------+-------+---------|
  1      344      267     1.21     0.07    32558       0      3    363        301

344 branches in the tree. 267 dead ends. Max depth 3.
{   'alpha': 'empress',
    'apple': 'smasher',
    'harem': ('sapples', 'papless'),
    'helms': 'sappare',
    'papess': 'harmel',
    'peeps': 'marshal',
    'phases': ('palmer', 'lamper', 'relamp'),
    'remap': 'hapless',
    'slash': 'empaper',
    'spasm': 'preheal',
    'sperma': 'alephs',
    'splash': 'ampere',
    ('hames', 'shame'): 'slapper',
    ('haps', 'hasp', 'pash'): 'resample',
    ('hassel', 'hassle'): ('pampre', 'mapper', 'pamper'),
    ('heaps', 'phase', 'shape'): ('lampers', 'sampler'),
    ('lames', 'males', 'meals'): ('prehaps', 'perhaps'),
    ('lamps', 'plasm', 'palms', 'psalm'): ('reshape', 'rephase'),
    ('lams', 'slam', 'alms'): 'preshape',
    ('peas', 'spae', 'apse', 'apes', 'pase'): {   'harp': ('mels', 'elms'),
                                                  'hemp': 'lars',
                                                  'lash': 'perm',
                                                  'marl': 'pehs',
                                                  'reps': 'halm',
                                                  ('haps', 'hasp', 'pash'): 'merl',
                                                  ('mars', 'arms', 'rams'): 'help',
                                                  ('palm', 'lamp'): ('hers', 'resh'),
                                                  ('raps', 'pars', 'spar', 'rasp'): 'helm',
                                                  ('slap', 'pals', 'alps', 'salp', 'laps'):
                                                        'herm',
                                                  ('spam', 'pams', 'samp', 'maps', 'amps'):
                                                        ('herl', 'lehr')},
    ('plasma', 'lampas'): ('sphere', 'herpes'),
    ('seraph', 'phrase', 'shaper'): 'sample',
    ('sperm', 'perms'): 'phaseal'}
```

Figure 7: Anagrams of *sample phrase*

```
Usage: dictbuilder [-h] [-d] -i INPUT [INPUT ...] [-n PROPERNOUNS] outfile

A program to maintain dictionaries used in anagrammar.

positional arguments:
  outfile               Name of the dictionary to be written (minus the suffixes).

optional arguments:
  -h, --help            show this help message and exit
  -b, --bare            Use only the words in the input dictionary rather than the
                        built-in 2, 3, 4, and 5 letter words.
  -i INPUT [INPUT ...], --input INPUT [INPUT ...]
                        The name[s] of the input dictionaries.
  -n PROPERNOUNS, --propernouns PROPERNOUNS
                        If used, exclude the words found in the file (presumed to
                        be proper nouns)
```

Figure 8: Command line options for the dictbuilder

6 Building dictionaries

The source code contains a single file named `dictbuilder.py`. It is both a standalone program and an `import` for this project because it contains the function `dictloader`. The dictionaries it builds are stored as Python pickles, and it offers useful command line switches to control the way the dictionaries are constructed.

6.1 Basic operations

Running the `dictbuilder` is much like running the ANAGRAMMAR. The help text is shown in Figure 8. The options offer considerable flexibility for making custom dictionaries. *[Note:* When a dictionary is loaded by the ANAGRAMMAR, the entire file is read. There is no option to inspect the dictionary while it is being read. *— end note.]*

`--bare` Normal operation is to use the lists of short words that are a part of the project. The 1, 2, and 3-letter words are in the source code itself, represented as `frozensets` of `str`. The 4-letter words are in the file `four.letter.words` and the 5-letter words are those of Donald Knuth, stored in the file `knuths.5757.five.letter.words.txt`. These may not be the ones you want to use, primarily because Knuth's list of words is unusually large, exceeding even the Webster's International dictionary.

`--input` This argument can be repeated, so that you may build a dictionary from several, merged sources. Normal operation is effectively the following:

```
-i four.letter.words -i knuths.5757.five.letter.words -i
yourownfile
```

`--propernouns` If this option is not present, all the input dictionaries are used *as is*. If it is present, it removes all the words that appear in the dictionaries with an initial capital letter, and then removes all the words in the filename associated with the option, regardless of whether they are capitalized.

outfile The name should be supplied without suffixes, because the construction process creates the two dictionaries as separate files, providing its own suffixes, `.forward` and `.reversed`.

6.2 Methods

The building of the dictionary takes place with the steps in the logical order to avoid conflicts. The so-called *forward* dictionary is built first, with the words read from the various sources as the keys, and the letters that make up each word, sorted in the standard lexicographic way, as the values.

The construction of the reversed dictionary requires more finesse. The construction iterates over the values of the forward dictionary to use as keys, and the value is a `defaultdict` of `list`, to which the corresponding words are appended. Once constructed, neither dictionary is mutable, so the values in the reversed dictionary are transformed into `tuple`s before the pickling process.

Pickles are generally faster to read than write, and they have the advantage that type information is stored with the data, allowing in memory data structures to be populated directly by reading the pickle.

George Flanagin, Provost Office
Data Analysis & Data Science

Phone:	+1.804.287.6392
Address:	Richmond Hall, Office 104 114 UR Drive University of Richmond Richmond, VA 23173
Email:	gflanagin@richmond.edu