Can You Add A Vowel To Lose A Syllable?
Riddler Express of April 15, 2022

As you may recall, around this time last year, I ran two puzzles by high school students who were winners of the Regeneron Science Talent Search. This year, I am delighted to have puzzles from two of this year's winners as well.

Luke Robitaille is from Euless, Texas. As part of his research project, he proved that most simple braids — topological structures composed on intertwining strands — are orderly for low numbers of strands. But as the number of strands increases, nearly all simple braids become chaotic. Luke also represented the United States three times in the International Math Olympiad, taking home three gold medals.
Now, I can't recall ever running a straight-up word puzzle in my days as the editor of The Riddler. But Luke's puzzle was too good to pass up, so here goes:

Find a word in the English language to which you can add a vowel, resulting in another word that has fewer syllables.

By "add a vowel," I mean insert one additional letter — a vowel — somewhere in the word (or at the beginning or end), while keeping the ordering of all the other letters the same. For example, you could add a vowel to the word "TASTY" to get the word "TOASTY." However, both words are two syllables, meaning this is not the solution.

Solution

Like other word puzzles presented before in The Riddler like "Can You Find A Number Worth Its Weight In Letters?" (January 10, 2020) and "Can You Find The Fish In State Names?" (May 30, 2020), I turned to my computer to solve this problem using brute force. The approach tests the overlap between each word and every other word that contains it from Peter Norvig's "word.list"[1] and counts the number of syllables using the Carnegie Mellon Pronouncing Dictionary.[2]

First, the setup: we need to download the word list. I used `urllib` to request the words and parse them in a list by splitting on newline characters.

```
from urllib.request import urlopen

def get_words(url: str = "https://norvig.com/ngrams/word.list") -> List[str]:
    """Downloads and returns a set of words from the web"""
    response = urlopen(url)
    return response.read().decode("utf-8").split("\n")
```

For the actual algorithm, I took advantage of a few data structures in Python that are well suited to this task: sets, lists, and dictionaries. For example, instead of comparing "proforma" with "profound", it is much faster to compare the `set` of letters in each word. Sets are data containers with unique elements, so the set of letters in "proforma" is `{'p','r','o','f','m','a'}` while the set of letters in "profound" is `{'p','r','o','f','u','n','d'}`. We can quickly test whether the two sets have any overlap using `set()` operations, like *union*, *intersection, difference* and *symmetric difference.*

In this problem, we want to find the *symmetric difference* between the two sets – the set of all the elements that are *either in* the first set *or* the second set *but not in both.* I used the `^` operator to do this within a custom function `uncommon()`. Continuing with the example above of "proforma" versus "profound", the *symmetric difference* would be `{'m','a','u','n','d'}`, which I called the `uncommonset`.

[1] https://norvig.com/ngrams/word.list
[2] https://github.com/cmusphinx/cmudict

However, the puzzle only cares if the *symmetric difference* is the addition of one vowel (‘a’, ‘e’, ‘i’, ‘o’ or ‘u’), so I stored the vowels as their own set, then found the *union* of the *symmetric difference* from above and the set of vowels, which I called the `commonset`. I added an `if` statement to check whether the uncommonset and the `commonset` each contained only one element before proceeding the count the number of syllables in each word.

```
from Collections import Counter

def uncommon(a,b):
    vowels = {'a','e','i','o','u'}
    # convert both strings into counter dictionary
    dicta = Counter(a)
    dictb = Counter(b)

    # take difference of these dictionaries
    uncommonset = set(dicta) ^ set(dictb)
    # take union of set of vowels and uncommon letters
    commonset = vowels & uncommonset
    # Check if a vowel has been added
    if len(uncommonset) == 1 & len(commonset) == 1:
      # Description to be continued below…
```

The function above is only meant to operate on two words at a time. So, for each word in Norvig’s “words.list,” I created another list of `commonwords` that had the word in it, based on the puzzle’s condition to protect the ordering of all the other letters. This means that my initial attempt at a solution only cared whether the vowel was added at the beginning or the end, not in the middle.

After confirming the length of the `uncommonset` and `commonset` are both one, I referenced two other custom functions that leverage the Carnegie Mellon Pronouncing Dictionary to count the number of syllables in a word.

```
import cmudict
def lookup_word(word):
    return cmudict.dict().get(word)

def count_syllables(word):
    count = 0
    # this returns a list of matching phonetic rep's
    phones = lookup_word(word)
    # if the list isn't empty (the word was found)
    if phones:
        # process the first word in the list
        phones0 = phones[0]
        # count the vowels
        count = len([p for p in phones0 if p[-1].isdigit()])
    return count
```

These two functions look up a word in the Carnegie Mellon Pronouncing Dictionary and counts the number of stresses in its list of phones. This appeared to work pretty well, although other users have documented that the dictionary is not flawless in its marking of stressed syllables. Regardless, I was appreciative of having a corpus to import as opposed to developing from scratch.

Returning to the `if` statement of `uncommon()` function, I counted the syllables in both words to see if a syllable is “dropped” after the vowel was added.

```
        # Description continued from above…
        counta = count_syllables(a)
        countb = count_syllables(b)
        if (counta != countb) and (countb > 0) and (counta > 0):
            return (a, b, uncommonset, commonset, counta, countb)
        else:
            return None
```

Again, I use an `if` statement to determine whether the count differs by one. In testing, I observed a few instances where the Carnegie Mellon Pronouncing Dictionary would register that a word had *zero* syllables, so I also filtered for those. If these conditions were met, the function returned the words, the *symmetric difference* and the syllable count for each word.

Below is the brute force approach to churning through the word list:

```
for w in words:
    commonwords = [x for x in words if w in x and (len(x) == len(w) + 1)]
    for c in commonwords:
        if uncommon(w, c):
            print(uncommon(w, c))
```

After letting the algorithm run in the background, it presented the following as a solution:

```
      ('ids', 'aids', {'a'}, 2, 1)
```

Let's interpret this in the context of what I have already mentioned about algorithm construction:

1. 'ids' – word A
2. 'aids' – word B, which contains word A
3. {'a'} – the *symmetric difference* between word A and word B which <u>must</u> be a singular vowel
4. 2 – the number of syllables in word A
5. 1 – the number of syllables in word B

I had all possible solutions print to the console throughout, so I could verify false positives. As I mentioned above, other users have documented (on GitHub) that the Carnegie Mellon Pronouncing Dictionary is not flawless. I noticed this with the two other results.

Dominique DeRubeis
April 16, 2022