Concept Matching

Mapping spoken phrases to potential concepts within a given user utterance.

The solution

A program which reads in a user phrase and returns the matches from a dictionary of millions of potential concepts, with a complexity that only relies on the size of the utterance.

Assumptions

Here are the assumptions made by the solution:

- **Format of input phrase**. The solution assumes words within the utterance are separated by whitespaces, spelled correctly and without punctuation.
- **Size of input phrase**. It is assumed that the utterances are under 20 words, as per specifications. Nonetheless, longer input phrases would not break the solution. *More under Evaluation*.
- **Source of concepts**. Production systems may decide to retrieve the list of concepts from any source, rather than from a hard-coded file as per this solution. The dictionary can be updated on the fly, with search results reflecting these changes.

Problems addressed

Here are the problems the solution addresses:

- **Concept list size**. The list of concepts to match against could run into the millions. It was essential to use an efficient data structure to check and retrieve potential matches. A self-resizing hashtable was implemented from scratch for this.
- **Capitalisation**. Even though the specifications do not instruct about capitalisation of words, one of the examples provided seems to indicate that it should not affect the matching process. The solution uses a custom compare function to ignore capitalisation and return the matches as they appear in the dictionary of concepts.
- **Duplicate matches**. The solution returns a set of matching concepts without duplicates. This choice was arbitrary and could be easily changed to include duplicate matches for concepts that appear more than once.

Implementation

Phase 1 - Efficient dictionary generation

The first phase is to create an efficient data structure to store the list of concepts. The idea is to use a dictionary of dictionaries where each word that makes up a concept is effectively a key and its value is a dictionary for the words that may follow.

This structure efficiently shares existing dictionaries for common sequences of words. For example, `South East European` and `South East Asian` will share the same dictionaries at the first two levels: `South` and `East`. If a dictionary contains the `None` key, then we know the parent key is a concept to be matched on its own.

In the `dictionary.py` file, we add each concept to the dictionary, first by finding an existing path that best matches the sequence of words we are trying to add. Then we generate a sub-dictionary for the words that have never been added before and insert it at the location found.

Part 2 - Fast concept matching

Having generated a dictionary of concepts, we find the matches for a given input as follows. For each word that makes up the utterance we search the dictionary starting from the top level. A match is found when the dictionary associated with a potential concept contains the `None` key. As there might be further matches down the line, we continue searching sub dictionaries of that word until we don't find matches anymore.

Evaluation

A analysis of the time and space complexity of the solution.

Custom HashTable. The solution provides a custom implementation of a HashTable. This data structure uses buckets and hashes to store values and retrieve them efficiently. To avoid hash collision which would turn its complexity linear, it is programmed to double its size when it reaches a certain threshold. This helps guaranteeing a time complexity of `O(1)` and a space complexity of `O(n)`, where **n** is the number of elements stored.

Dictionary generation. The process of generating a dictionary of concepts is of time complexity `O(n * m)`, where **n** is the number of concepts (potentially millions) and **m** is the average number of words per concept. Assuming **m** is smaller than 20 - the maximum length of an utterance - complexity reduces to `O(n)`. The space complexity is `O(n)`. A dictionary has an underlying HashTable, which starts with a size of 1, allowing most one-word dictionaries to be as efficient as possible. This process only runs once.

Concept matching. For each word in an utterance, we search the dictionary for that word and for any further words, potentially up to the length of the utterance itself. This means that the complexity of this process is `O(w^2)` where **w** is the number of words per utterance. We know this number to be less than 20, making it an acceptable complexity. The time complexity for concept matching under this solution **does not depend on the number of concepts**.

Running the program

Python should be available on MacOS with developer tools installed. Python 3 is needed for list unpacking.

The tool can be run with:

```
make run
```

Running tests

All tests can be run with:

```
make test
```

> Developed by James Reaver in March 2020.