README.md

Boggle

Implementation of a Boggle word game solver using DFS to search the board and a Trie data structure to store a dictionary of valid words.

Author: Sanjana Marcé

Date: June 29, 2021

Problem Specification

Given a *board* and a *dictionary*, find words (strings that appear in the *dictionary*) that can be formed by a sequence of adjacent (top, bottom, left, right, diagonal) letters on the *board*. Words must be 3 or more letters. You may move to any of 8 adjacent letters, however, a word should not have multiple instances of the same cell.

In the original problem specification, the *board* is specified to be 4x4, however I generalized my implementation to accept any board of rectangular dimensions.

Usage

To play the boggle game, run the `driver.py` program on the command line with optional arguments `--board` and `--vocab`.

Run `python driver.py --board <board_string> --vocab <path_to_dictionary_file>`

To run using defaults, run `python driver.py`.

- **`<board_string>`**: the input rectangular `board` is specified as a string to the `driver.py` file. Each row is specified from top to bottom as a word, and rows are separated by a space. For example, the argument `--board "rael mofs teok nati"` specifies the following board:
  -
r	a	e	l
m	o	f	s
t	e	o	k
n	a	t	i
  - If no `<board_string>` is specified, the above board is used as the default.
- **`<path_to_dictionary_file>`**: the `vocab` is specified as a `txt` file where each valid word is on a new line. The file `dictionary.txt` is an example of a valid such file.
  - If no `<path_to_dictionary_file>` is specified, this open source dictionary of 194,000 English words downloaded from gwicks.net (stored in the file `dictionary.txt`) is used as the default.

Approach & Optimizations

In my initial implementation, I stored the *dictionary* as a set and ran a basic depth first search (DFS) of the *board* initiated at each of the 16 cells of the *board* to generate all possible string combinations, and then returned those strings that existed in the *dictionary* and met the requirements for minimum length. For an M x N board, this approach requires exponential time complexity O(MN * 8^MN), as for each of the MN cells from which we initiate the search, there are 8^MN possible strings starting with that cell. Looking up each of these words for membership in the dictionary is O(1) on average (set lookup) but O(dict_size) in the worst case.

My first optimization was to reduce the size of the input *dictionary* to only use the subset of the *dictionary* with words made up of characters that actually appear in the given *board*. Reducing the dictionary requires time complexity O(dict_size * MN) to run through the entire dictionary once and only keep those words that only contain characters present in the MN cells on the board. This reduction improves the dictionary lookup to confirm a word is valid by reducing the dict_size.

My second optimization was to store the *dictionary* in a Trie data structure instead of a set to leverage the fact that the words we generate via DFS share prefixes with those words generated previously in the search. Thus, we can store the dictionary as a Trie to take into account this prefix structure to effectively prune the search space as we find word prefixes for which no valid words exist. For example, if no words appear in the dictionary starting with 'kf' then searching deeper for strings of the form 'kfa', 'kfe', etc. is not necessary. Constructing the tree takes complexity O(dict_size * max_dict_word_len), where dict_size is the number of words in the input dictionary and max_dict_word_len is the length of the longest word in the dictionary. Determining whether a string or any words that have that string as a prefix exist in the dictionary takes time proportional to the length of the string. In the case of our Boggle board, determining the validity of the longest possible string from the board requires O(MN), which is much faster than O(dict_size). In the worst case, every possible word in the *board* appears in the dictionary and running the DFS on the search space of possible words is not faster than the previous implementation. However, in the average case (and with any realistic dictionary) the search space will be pruned substantially using prefixes and observed runtime will be much faster.

Files

- **`driver.py`** : main functions to run the boggle word finder using DFS on an input board (storing a dictionary of valid words as a Trie)
- **`vocab.py`** : functions to load in a dictionary from a txt file and to reduce this dictionary size based on a specified alphabet
- **`trie.py`** : defines the TrieNode and Trie classes and their associated functions
- **`dictionary.txt`** : sample dictionary txt file from gwicks.net containing a list of 194,000 valid English words

Notes

- Currently, implementation is case sensitive, so words found on the *board* must match an element in the *dictionary* exactly (up to capitalization).
- Currently, if the same word can be made in multiple ways on the *board* (i.e. 'foot' on the sample board shown above), that word is printed multiple times in the output. If this behavior is not desired, we could amend the existing code by either converting the output list to a set (the code for which is commented at the end of the `main` function in `driver.py`) or by removing found words as valid words from the Trie as we find them.