Daily Coding Problem #227

Problem

This problem was asked by Facebook.

Boggle is a game played on a `4 x 4` grid of letters. The goal is to find as many words as possible that can be formed by a sequence of adjacent letters in the grid, using each cell at most once. Given a game board and a dictionary of valid words, implement a Boggle solver.

Solution

Words in Boggle are created by starting with one of the grid letters and repeatedly moving in one of eight possible directions until forming a valid word. For concreteness, we can use the grid and dictionary below.

```
grid = [["a", "b", "c", "d"],
        ["x", "a", "y", "z"],
        ["t", "z", "r", "r"],
        ["s", "q", "q", "q"]]
```

```
dictionary = ['bat', 'car', 'cat']
```

Let's start with the letter `c`. Since `c` has five neighbors, we must look at them one at a time, searching for words we can generate with `cb`, `ca`, `cy`, `cz`, and `cd`, respectively. When we take the `ca` path, we will end up appending `t`, creating `cat`.

This is essentially a modified depth-first search on a graph whose vertices are the letters, and whose edges are links between adjacent letters. Searching a neighbor involves adding the new letter to the current word and checking to see if the word is in our dictionary. The only difference is that even when we form a word, we should continue searching; in the grid above, for example, we might want both `cat` and `cats`.

Our first attempt at a solution, then, involves running our modified depth-first search from every possible starting location, and adding each word we find along the way to a result set.

```
def get_neighbors(location, grid_size=4):
    i, j = location
    neighbors = [(i - 1, j - 1), (i - 1, j), (i - 1, j + 1),
                 (i, j - 1), (i, j + 1),
                 (i + 1, j - 1), (i + 1, j), (i + 1, j + 1)]

    return [n for n in neighbors if 0 <= n[0] < grid_size and 0 <= n[1] <
grid_size]

def search(location, grid, visited, word, result, dictionary):
    visited.add(location)
    if word in dictionary:
        result.add(word)

    for neighbor in get_neighbors(location):
        if neighbor not in visited:
            word += grid[neighbor[0]][neighbor[1]]
            search(neighbor, grid, visited, word, result, dictionary)
            word = word[:-1]
            visited.remove(neighbor)
```

```
def boggle(grid, dictionary):
    visited = set()
    result = set()

    for row in range(len(grid)):
        for col in range(len(grid)):
            word = grid[row][col]
            search((row, col), grid, visited, word, result, dictionary)

    return list(result)
```

While relatively straightforward, this approach is not that efficient.

In the grid above, for example, our algorithm will search fifteen levels deep starting from the letter a, when there is not even a word in our dictionary beginning with a! One improvement, then, would be to only start searching from letters that also begin words in our dictionary. Still, we would have the same search explosion once we reach the second letter.

One way to generalize this improvement to all levels is to convert our dictionary into a trie. With this data structure, we can know, for any prefix, the potential words that can be made starting with that prefix. As a result, we will only search adjacent letters that can help us form words, drastically reducing the search space.

For example, using the word list from above, `['bat', 'car', 'cat']`, our prefix trie would look like the following:

```
{'b':
    {'a':
        {'t': {'#': '#'}}}
 'c':
     {'a':
         {'r': {'#': '#'},
          't': {'#': '#'}}}}
```

The # key tells us that we have reached the end of a valid word. So we no longer need to check if a word is in our dictionary- once we find this character, we know to add the current word to our set of results.

Here is how we can create such a trie:

```
def make_trie(words):
    root = {}
    for word in words:
        current_dict = root
        for letter in word:
            current_dict = current_dict.setdefault(letter, {})
        current_dict['#'] = '#'
    return root
```

The rest of our algorithm remains very similar, except that we make sure to check whether an adjacent letter exists in the trie before searching that path. In addition, storing our current word as a list, rather than a string, will allow us to more efficiently append and pop letters from it.

```
def search(location, grid, trie, visited, word, result):
    visited.add(location)
    letter = grid[location[0]][location[1]]
    if '#' in trie[letter]:
        result.add(''.join(word))

    for neighbor in get_neighbors(location):
        if neighbor not in visited:
            row, col = neighbor
            if grid[row][col] in trie[letter]:
                subtrie = trie[letter]
                word.append(grid[row][col])
                search(neighbor, grid, subtrie, visited, word, result)
                word.pop()
                visited.remove(neighbor)

def boggle(grid, trie):
    visited = set()
    result = set()

    for row in range(len(grid)):
        for col in range(len(grid)):
            if grid[row][col] in trie:
                word = [grid[row][col]]
                search((row, col), grid, trie, visited, word, result)

    return list(result)
```

The time complexity of creating a trie is `O(N * k)`, where `N` is the number of words in our dictionary and `k` is the average word length. Finding a word in a trie is `O(k)`, since we may need to check for a match at each letter of the word.

For this algorithm, since we are only traversing paths that can lead to words, our search will be equivalent to trying to find every word in our dictionary. As a result, if all the letters are distinct the overall complexity will be `O(N * k)`.