[medium,BFS, string]127. Word Ladder

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

For example,

Given: *beginWord* = `"hit"` *endWord* = `"cog"` *wordList* = `["hot","dot","dog","lot","log","cog"]`

As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`, return its length `5`.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

UPDATE (2017/1/20): The *wordList* parameter had been changed to a list of strings (instead of a set of strings). Please reload the code definition to get the latest changes.

Well, this problem has a nice BFS structure.

Let's see the example in the problem statement.

`start = "hit"`

`end = "cog"`

`dict = ["hot", "dot", "dog", "lot", "log"]`

Since only one letter can be changed at a time, if we start from `"hit"`, we can only change to those words which have only one different letter from it, like `"hot"`. Putting in graph-theoretic terms, we can say that `"hot"` is a neighbor of `"hit"`.

The idea is simpy to begin from `start`, then visit its neighbors, then the non-visited neighbors of its neighbors... Well, this is just the typical BFS structure.

To simplify the problem, we insert `end` into `dict`. Once we meet `end` during the BFS, we know we have found the answer. We maintain a variable `dist` for the current distance of the transformation and update it by `dist++` after we finish a round of BFS search (note that it should fit the definition of the distance in the problem statement). Also, to avoid visiting a word

for more than once, we erase it from `dict` once it is visited.

The code is as follows.

确实挺难的。

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord,
unordered_set<string>& wordDict) {
        wordDict.insert(endWord);
        queue<string> toVisit;
        addNextWords(beginWord, wordDict, toVisit);
        int dist = 2;
        while (!toVisit.empty()) {
            int num = toVisit.size();
            for (int i = 0; i < num; i++) {
                string word = toVisit.front();
                toVisit.pop();
                if (word == endWord) return dist;
                addNextWords(word, wordDict, toVisit);
            }
            dist++;
        }
    }
private:
    void addNextWords(string word, unordered_set<string>& wordDict,
queue<string>& toVisit) {
        wordDict.erase(word);
        for (int p = 0; p < (int)word.length(); p++) {
            char letter = word[p];
            for (int k = 0; k < 26; k++) {
                word[p] = 'a' + k;
                if (wordDict.find(word) != wordDict.end()) {
                    toVisit.push(word);
                    wordDict.erase(word);
                }
            }
            word[p] = letter;
        }
    }
};
```