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 intermediate word must exist in the word list

For example,

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

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.

Solution 1

```
//BFS,  two-end method
//traverse the path simultaneously from start node and end node, and merge in the
middle
//the speed will increase (logN/2)^2 times compared with one-end method
int ladderLength(string start, string end, unordered_set<string> &dict) {
    unordered_set<string> begSet, endSet, *set1, *set2;
    begSet.insert(start);
    endSet.insert(end);
    int h=1, K=start.size();
    while(!begSet.empty()&&!endSet.empty()){
        if(begSet.size()<=endSet.size()){   //Make the size of two sets close for
optimization
            set1=&begSet;   //set1 is the forward set
            set2=&endSet;   //set2 provides the target node for set1 to search
        }
        else{
            set1=&endSet;
            set2=&begSet;
        }
        unordered_set<string> itmSet;   //intermediate Set
        h++;
        for(auto i=set1->begin();i!=set1->end();i++){
            string cur=*i;
            for(int k=0;k<K;k++){   //iterate the characters in string cur
                char temp=cur[k];
                for(int l=0;l<26;l++){  //try all 26 alphabets
                    cur[k]='a'+l;
                    auto f=set2->find(cur);
                    if(f!=set2->end())return h;
                    f=dict.find(cur);
                    if(f!=dict.end()){
                        itmSet.insert(cur);
                        dict.erase(f);
                    }
                }
                cur[k]=temp;
            }
        }
        swap(*set1, itmSet);
    }
    return 0;
}
```

written by VaultBoy original link here

Solution 2

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.

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, unordered_set<string>& wor
dDict) {
        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;
        }
    }
};
```

The above code can still be speeded up if we also begin from `end`. Once we meet the same word from `start` and `end`, we know we are done. This link provides a nice two-end search solution. I rewrite the code below for better readability. Note that the use of two pointers `phead` and `ptail` save a lot of time. At each round of BFS, depending on the relative size of `head` and `tail`, we point `phead` to the smaller set to reduce the running time.

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, unordered_set<string>& wor
dDict) {
        unordered_set<string> head, tail, *phead, *ptail;
        head.insert(beginWord);
        tail.insert(endWord);
        int dist = 2;
        while (!head.empty() && !tail.empty()) {
            if (head.size() < tail.size()) {
                phead = &head;
                ptail = &tail;
            }
            else {
                phead = &tail;
                ptail = &head;
            }
            unordered_set<string> temp;
            for (auto itr = phead -> begin(); itr != phead -> end(); itr++) {
                string word = *itr;
                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 (ptail -> find(word) != ptail -> end())
                            return dist;
                        if (wordDict.find(word) != wordDict.end()) {
                            temp.insert(word);
                            wordDict.erase(word);
                        }
                    }
                    word[p] = letter;
                }
            }
            dist++;
            swap(*phead, temp);
        }
        return 0;
    }
};
```

written by jianchao.li.fighter original link here

Solution 3

Shouldn't the output of the test case below be 1? Because "a" could directly be changed to "c", which needs only once edit. Input: "a", "c", ["a","b","c"] Output: 1 Expected: 2

written by wywangywy original link here