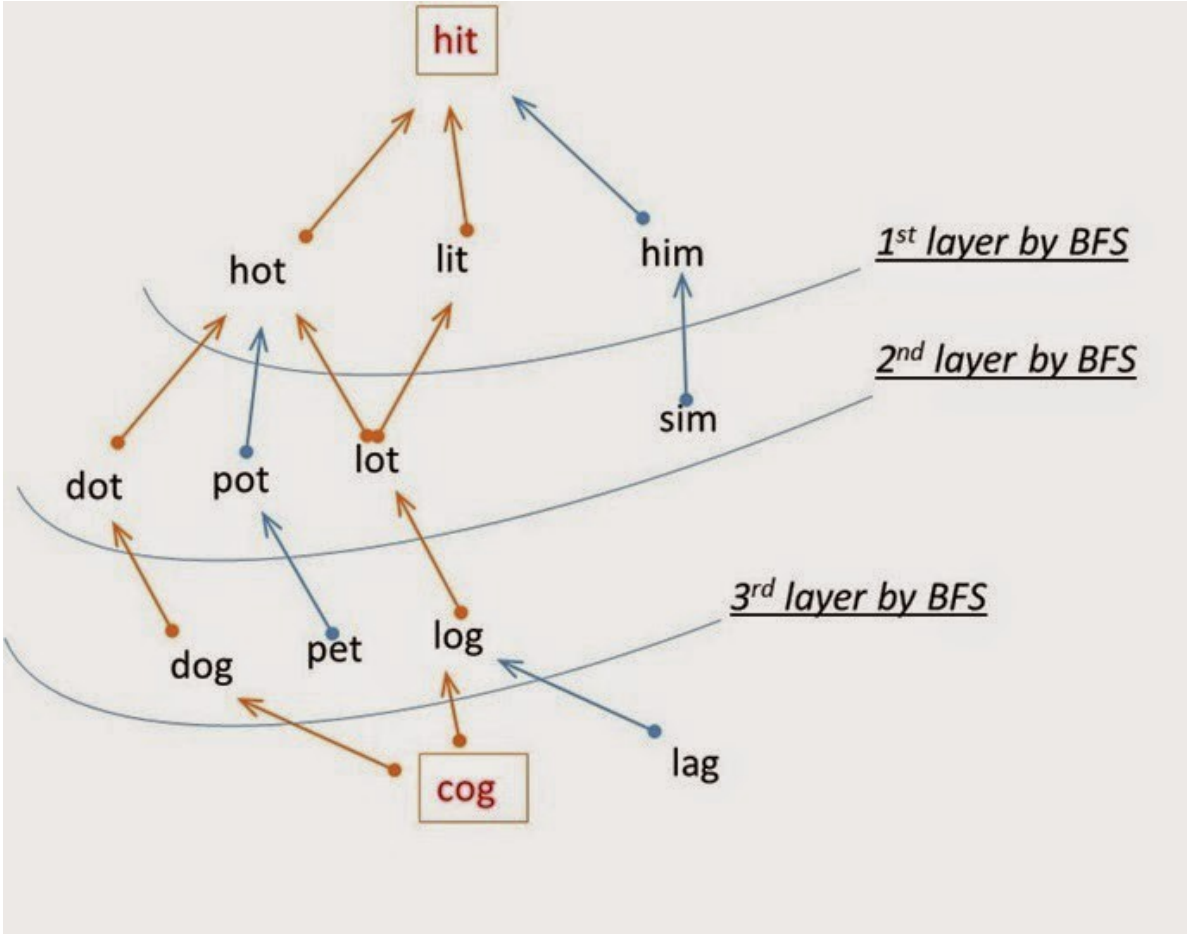
medium.com

Word Ladder II solution - spylogsster - Medium

spylogsster

4-5 minutes

Example of word ladder



Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) 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.

Note:

- Return an empty list 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.

Example 1:

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

Example 2:

Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]**Output:** []**Explanation:** The endWord "cog" is not in wordList, therefore no possible transformation.

Solution

The idea is to first use BFS to search from `beginWord` to `endWord` and generate the word-to-children mapping at the same time. Then, use DFS (backtracking) to generate the transformation sequences according to the mapping.

```
class Solution {
    // Finds endWord by BFS and create parent->children word relations
  bool findEndWordByBFS(const string& beginWord,
                        const string& endWord,
                        const vector<string>& wordList,
                        unordered_map<string, vector<string>>& children) {
    unordered_set<string> dict(wordList.begin(), wordList.end()), current, next;
    current.insert(beginWord);
    while (true) {
      for (string word : current) {
        dict.erase(word);
      }

            for (string word : current) {
        string parent = word;
        for (int i = 0; i < word.size(); i++) {
          char t = word[i];
          for (int j = 0; j < 26; j++) {
            word[i] = 'a' + j;
            if (dict.find(word) != dict.end()) {
              next.insert(word);
```

```
                    children[parent].push_back(word);
                }
            }
            word[i] = t;
        }
    }
                if (next.empty())
        return false;
                if (next.find(endWord) != next.end())
        return true;
    current.clear();
        swap(current, next);
    }
        return false;
  }
  //  Use DFS (backtracking) to generate the transformation
sequences according to the mapping
  void buildLadders(const string& beginWord,
                const string& endWord,
                vector<string>& ladder,
                vector<vector<string>>& ladders,
                unordered_map<string, vector<string>>& children) {
    if (beginWord == endWord) {
      ladders.push_back(ladder);
    } else {
      for (string word : children[beginWord]) {
        ladder.push_back(word);
        buildLadders(word, endWord, ladder, ladders, children);
        ladder.pop_back();
```

```
            }
        }
    }
public:
    vector<vector<string>> findLadders(string beginWord, string
endWord, vector<string>& wordList) {
        unordered_set<string> dict(wordList.begin(), wordList.end());
        if (dict.find(endWord) == dict.end())
            return {};
        unordered_map<string, vector<string>> children;
        if (!findEndWordByBFS(beginWord, endWord, wordList,
children))
            return {};
        vector<vector<string>> ladders;
        vector<string> ladder;
        ladder.push_back(beginWord);
        buildLadders(beginWord, endWord, ladder, ladders, children);
        return ladders;
    }
};
```

Resources

- http://slientcode.blogspot.com/2014/11/word-ladder-ii.html
  https://leetcode.com/problems/word-ladder-ii/discuss/241927/C%2B%2B-BFS-%2B-DFS