# 33 单词接龙

更新时间：2019-09-23 10:18:14


> 读书而不思考，等于吃饭而不消化。
>
> ——波尔克

## 刷题内容

> 难度: Hard
>
> 题目链接：https://leetcode-cn.com/problems/word-ladder-ii/submissions/

## 题目描述

给定两个单词（beginWord 和 endWord）和一个字典 wordList，找出所有从 beginWord 到 endWord 的最短转换序列。转换需遵循如下规则：

每次转换只能改变一个字母。
转换过程中的中间单词必须是字典中的单词。
说明:

如果不存在这样的转换序列，返回一个空列表。
所有单词具有相同的长度。
所有单词只由小写字母组成。
字典中不存在重复的单词。
你可以假设 beginWord 和 endWord 是非空的，且二者不相同。
示例 1:

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

输出:
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]
示例 2:

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

输出: []

解释: endWord "cog" 不在字典中，所以不存在符合要求的转换序列。

# 解题方案

## 思路1 时间复杂度: O(len(wordList) * max(len(word)) 空间复杂度: O(len(wordList))

这道题我们可以从当前的beginWord去着手，先找到它可能的所有的下一变化单词，然后判断下其是否存在于我们的wordList中，如果存在，那么这个词就被用过了，我们记录下这个变换的路径，以此类推，直到找到endWord或者是所有的词都被用过了，但是没有endWord。

如果是因为找到endWord而退出的前面的while循环，那么我们知道存在变换方式， 那么就去找到所有的这种变换方式即可。这里我们的dfs函数就是去从最后一个单词一个一个往前面推，去看它们的上一个单词是谁，直到退回到beginWord，此时我们把整个path作为结果中的一种。

***Python beat 69.09%***

```python
class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:
        def dfs(res, prev_words_lookup, path, cur_word):
            if len(prev_words_lookup[cur_word]) == 0:  # 发现word是第一个词
                res.append([cur_word] + path)
            else:
                for prev_word in prev_words_lookup[cur_word]:
                    dfs(res, prev_words_lookup, [cur_word] + path, prev_word)

        all_words = set(wordList) | set([beginWord])  # 所有的词（包括beginWord）
        res, cur_queue = [], set([beginWord]),
        prev_words_lookup = {word: [] for word in all_words}  # k:v = word: [prev_word1, prev_word2...]
        while cur_queue and endWord not in cur_queue:  # 直到找到endWord或者是所有的词都被用过了，但是没有endWord
            nxt_queue = set()  # 下一轮要用的词
            for cur_word in cur_queue:
                all_words.remove(cur_word)  # 用过的词从all_words中删除掉
            for cur_word in cur_queue:
                for i in range(len(cur_word)):
                    for j in 'abcdefghijklmnopqrstuvwxyz':
                        nxt_word = cur_word[:i] + j + cur_word[i + 1:]
                        if nxt_word in all_words:  # 如果generate出来的这个candidate可以选
                            nxt_queue.add(nxt_word)
                            prev_words_lookup[nxt_word].append(cur_word)  # candidate前面的词有一个是这里的word
            cur_queue = nxt_queue

        if cur_queue:  # 如果是因为找到endWord而退出的前面的while循环，那么我们知道存在变换方式
            dfs(res, prev_words_lookup, [], endWord)
        return res
```

***c++ beats 12%***

```
class Solution {
public:
    //从endWord遍历到beginWord，遍历所有的最短变换数组
    //ret：结果数组，为了节省复杂度，dfs过程中直接把结果丢进去
    void dfs(vector<vector<string>>& ret, vector<string>& current,
        map<string, vector<string>>& prev,
        string& beginWord, string& nowWord) {
        current.push_back(nowWord);
        if (nowWord == beginWord) {
            ret.push_back(vector<string>(current.rbegin(), current.rend()));
            current.pop_back();
            return;
        }
        //遍历所有的前置字符串
        for (auto s: prev[nowWord]) {
            dfs(ret, current, prev, beginWord, s);
        }
        //恢复到函数调用之前的状态
        current.pop_back();
    }
    vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList) {
        set<string> wordSet(wordList.begin(), wordList.end());

        //prev表示字符串的前置
        map<string, vector<string>> prev;
        //Q表示bfs的队列，排在前面离beginWord越近
        queue<string> Q;
        Q.push(beginWord);
        while (!Q.empty()) {
            //属于这一层的字符串
            set<string> levelSet;
            int n = Q.size();
            for (int i = 0;i < n;i++) {
                string s = Q.front();
                Q.pop();

                //遍历跟s距离为1的字符串
                for (int j = 0;j < s.size();j++) {
                    for (char c = 'a';c <= 'z';c++) {
                        if (s[j] == c) {
                            continue;
                        }
                        string temp = s;
                        temp[j] = c;

                        //如果该字符串已经在该层出现，那么s->temp也是在最短路径上
                        if (levelSet.find(temp) != levelSet.end()) {
                            prev[temp].push_back(s);
                        } else if (wordSet.find(temp) != wordSet.end()) {
                            prev[temp].push_back(s);
                            wordSet.erase(wordSet.find(temp));
                            levelSet.insert(temp);
                            Q.push(temp);
                        }
                    }
                }
            }
            if (levelSet.find(endWord) != levelSet.end()) {
                break;
            }
        }
        vector<vector<string>> ret;
        vector<string> current;
        dfs(ret, current, prev, beginWord, endWord);
        return ret;
    }
};
```

***java beats 36%***

```
class Solution {
    //从endWord遍历到beginWord，遍历所有的最短变换数组
    //ret：结果数组，为了节省复杂度，dfs过程中直接把结果丢进去
    private void dfs(List<List<String>> ret, LinkedList<String> current,
        Map<String, List<String>> prev,
        String beginWord, String nowWord) {
        current.addFirst(nowWord);
        if (nowWord.equals(beginWord)) {
            ret.add(new ArrayList<String>(current));
            current.removeFirst();
            return;
        }
        //遍历所有的前置字符串
        for (String s: prev.get(nowWord)) {
            dfs(ret, current, prev, beginWord, s);
        }
        //恢复到函数调用之前的状态
        current.removeFirst();
    }

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {
        Set<String> wordSet = new HashSet<String>(wordList);

        //prev表示字符串的前置
        Map<String, List<String>> prev = new HashMap<>();
        //Q表示bfs的队列，排在前面离beginWord越近
        Queue<String> Q = new LinkedList<>();
        Q.add(beginWord);
        while (!Q.isEmpty()) {
            //属于这一层的字符串
            Set<String> levelSet = new HashSet<>();
            int n = Q.size();
            for (int i = 0;i < n;i++) {
                String s = Q.poll();

                //遍历跟s距离为1的字符串
                for (int j = 0;j < s.length();j++) {
                    for (char c = 'a';c <= 'z';c++) {
                        if (s.charAt(j) == c) {
                            continue;
                        }
                        StringBuffer sb = new StringBuffer(s);
                        sb.setCharAt(j, c);
                        String temp = sb.toString();

                        //如果该字符串已经在该层出现，那么s->temp也是在最短路径上
                        if (levelSet.contains(temp)) {
                            prev.get(temp).add(s);
                        } else if (wordSet.contains(temp)) {
                            prev.put(temp, new ArrayList<String>());
                            prev.get(temp).add(s);
                            wordSet.remove(temp);
                            levelSet.add(temp);
                            Q.add(temp);
                        }
                    }
                }
            }
            if (levelSet.contains(endWord)) {
                break;
            }
        }
        List<List<String>> ret = new ArrayList<>();
        if (!prev.containsKey(endWord)) {
            return ret;
        }
```

```
        LinkedList<String> current = new LinkedList<>();
        dfs(ret, current, prev, beginWord, endWord);
        return ret;
    }
}
```

### *Go beats 61.06%*

```
var res [][]string

func dfs(prevWordsLookup map[string][]string, path []string, curWord string) {
    if len(prevWordsLookup[curWord]) == 0 { // 发现word是第一个词
        tmp := append(path, curWord)
        tmpRes := make([]string, 0)
        for i := len(tmp) - 1; i > -1; i-- {
            tmpRes = append(tmpRes, tmp[i])
        }
        res = append(res, tmpRes)
    } else {
        for _, prevWord := range prevWordsLookup[curWord] {
            dfs(prevWordsLookup, append(path, curWord), prevWord)
        }
    }
}

func findLadders(beginWord string, endWord string, wordList []string) [][]string {
    allWords, curQueue := map[string]bool{}, map[string]bool{}
    for _, w := range wordList { // allWords是所有的词（包括beginWord）
        allWords[w] = true
    }
    allWords[beginWord] = true
    curQueue[beginWord] = true
    prevWordsLookup := map[string][]string{}

    _, ok := curQueue[endWord]
    for len(curQueue) > 0 && !ok { // 直到找到endWord或者是所有的词都被用过了，但是没有endWord
        nxtQueue := map[string]bool{} // 下一轮要用的词
        for curWord := range curQueue {
            delete(allWords, curWord)  // 用过的词从all_words中删除掉
        }
        for curWord := range curQueue {
            for i := 0; i < len(curWord); i++ {
                for _, j := range "abcdefghijklmnopqrstuvwxyz" {
                    nxtWord := curWord[:i] + string(j) + curWord[i+1:]
                    if _, ok1 := allWords[nxtWord]; ok1 { // 如果generate出来的这个candidate可以选
                        nxtQueue[nxtWord] = true
                        prevWordsLookup[nxtWord] = append(prevWordsLookup[nxtWord], curWord) // candidate前面的词有一个是这里的word
                    }
                }

            }
        }
        curQueue = nxtQueue
        _, ok = curQueue[endWord]
    }
    if len(curQueue) > 0 { // 如果是因为找到endWord而退出的前面的while循环，那么我们知道存在变换方式
        dfs(prevWordsLookup, []string{}, endWord)
    }
    realRes := res
    res = [][]string{}
    return realRes
}
```

# DFS和BFS相关

**DFS/BFS**

可以让stack/queue记录更多一些的东西，因为反正stack/queue更像通用结构

```
   A
 /   \
C     B
\    / \
 \   D E
  \   /
    F


graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

**DFS**

迭代版本

```
def dfs(graph, start): # iterative
    visited, stack = [], [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.append(vertex)
            stack.extend(graph[vertex] - set(visited))
    return visited
print(dfs(graph, 'A')) # ['A', 'C', 'F', 'E', 'B', 'D'] 这只是其中一种答案
```

递归版本

```
def dfs(graph, start, visited=None): # recursive
    if visited is None:
        visited = []
    print('visiting', start)
    visited.append(start)
    for next in graph[start]:
        if next not in visited:
            dfs(graph, next, visited)
    return visited
print(dfs(graph, 'A')) # ['A', 'C', 'F', 'E', 'B', 'D'] 这只是其中一种答案
```

迭代打印出从出发点到终点的路径

```
def dfs_paths(graph, start, goal): # iterative
    stack = [(start, [start])]
    while stack:
        (vertex, path) = stack.pop()
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                stack.append((next, path + [next]))
print(list(dfs_paths(graph, 'A', 'F'))) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

递归打印出从出发点到终点的路径

```
def dfs_paths(graph, start, goal, path=None): # recursive
    if path is None:
        path = [start]
    if start == goal:
        yield path
    for next in graph[start] - set(path):
        yield from dfs_paths(graph, next, goal, path + [next])
print(list(dfs_paths(graph, 'C', 'F'))) # [['C', 'A', 'B', 'E', 'F'], ['C', 'F']]
```

**BFS**

只有迭代版本，和DFS唯一的区别就是pop(0)而不是pop()

```
def bfs(graph, start): # iterative
    visited, queue = [], [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.append(vertex)
            queue.extend(graph[vertex] - set(visited))
    return visited
print(bfs(graph, 'A')) # ['A', 'C', 'B', 'F', 'D', 'E']
```

返回两点之间的所有路径，第一个一定是最短的

```
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))
print(list(bfs_paths(graph, 'A', 'F'))) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

知道了这个特性，最短路径就很好搞了

```
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None
print(shortest_path(graph, 'A', 'F'))  # ['A', 'C', 'F']
```

**DFS 和 BFS的特点**

*一、深度优先搜索（**dfs**）的特点是：*

1. 深度优先搜索法有递归以及非递归两种设计方法。一般的，当搜索深度较小、问题递归方式比较明显时，用递归方法设计好，它可以使得程序结构更简捷易懂。当数据量较大时，由于系统堆栈容量的限制，递归容易产生溢

出，用非递归方法设计比较好。

2. 深度优先搜索方法有广义和狭义两种理解。广义的理解是，只要最新产生的结点（即深度最大的结点）先进行扩展的方法，就称为深度优先搜索方法。在这种理解情况下，深度优先搜索算法有全部保留和不全部保留产生的结点的两种情况。而狭义的理解是，仅仅只保留全部产生结点的算法。本书取前一种广义的理解。不保留全部结点的算法属于一般的回溯算法范畴。保留全部结点的算法，实际上是在数据库中产生一个结点之间的搜索树，因此也属于图搜索算法的范畴。
3. 不保留全部结点的深度优先搜索法，由于把扩展望的结点从数据库中弹出删除，这样，一般在数据库中存储的结点数就是深度值，因此它占用的空间较少，所以，当搜索树的结点较多，用其他方法易产生内存溢出时，深度优先搜索不失为一种有效的算法。
4. 不一定会得到最优解，这个时候需要修改原算法：把原输出过程的地方改为记录过程，即记录达到当前目标的路径和相应的路程值，并与前面已记录的值进行比较，保留其中最优的，等全部搜索完成后，才把保留的最优解输出。

*二、广度优先搜索法的显著特点是：*

1. 在产生新的子结点时，深度越小的结点越先得到扩展，即先产生它的子结点。为使算法便于实现，存放结点的数据库一般用队列的结构。
2. 无论问题性质如何不同，利用广度优先搜索法解题的基本算法是相同的，但数据库中每一结点内容，产生式规则，根据不同的问题，有不同的内容和结构，就是同一问题也可以有不同的表示方法。
3. 当结点到跟结点的费用（有的书称为耗散值）和结点的深度成正比时，特别是当每一结点到根结点的费用等于深度时，用广度优先法得到的解是最优解，但如果不成正比，则得到的解不一定是最优解。这一类问题要求出最优解，一种方法是使用后面要介绍的其他方法求解，另外一种方法是改进前面深度（或广度）优先搜索算法：找到一个目标后，不是立即退出，而是记录下目标结点的路径和费用，如果有多个目标结点，就加以比较，留下较优的结点。把所有可能的路径都搜索完后，才输出记录的最优路径。
4. 广度优先搜索算法，一般需要存储产生的所有结点，占的存储空间要比深度优先大得多，因此程序设计中，必须考虑溢出和节省内存空间得问题。
5. 比较深度优先和广度优先两种搜索法，广度优先搜索法一般无回溯操作，即入栈和出栈的操作，所以运行速度比深度优先搜索算法法要快些。

***总之，一般情况下，深度优先搜索法占内存少但速度较慢，广度优先搜索算法占内存多但速度较快，在距离和深度成正比的情况下能较快地求出最优解。因此在选择用哪种算法时，要综合考虑。决定取舍。***

1.BFS是用来搜索最短径路的解是比较合适的，比如求最少步数的解，最少交换次数的解，因为BFS搜索过程中遇到的解一定是离根最近的，所以遇到一个解，一定就是最优解，此时搜索算法可以终止。这个时候不适宜使用DFS，因为DFS搜索到的解不一定是离根最近的，只有全局搜索完毕，才能从所有解中找出离根的最近的解。（当然这个DFS的不足，可以使用迭代加深搜索ID-DFS去弥补）

2.空间优劣上，DFS是有优势的，DFS不需要保存搜索过程中的状态，而BFS在搜索过程中需要保存搜索过的状态，而且一般情况需要一个队列来记录。

3.DFS适合搜索全部的解，因为要搜索全部的解，那么BFS搜索过程中，遇到离根最近的解，并没有什么用，也必须遍历完整棵搜索树，DFS搜索也会搜索全部，但是相比DFS不用记录过多信息，所以搜素全部解的问题，DFS显然更加合适。

上面提到的迭代加深搜索（ID-dfs）我觉得充分吸收了BFS和DFS各自的长处

## Improvement/Follow up

1. 一旦BFS/DFS与更具体的，更有特性的data structure结合起来，比如binary search tree，那么BFS/DFS会针对这个tree traversal显得更有特性。
2. it's worth mentioning that there is an optimized queue object in the collections module called deque) for which removing items from the beginning ( or popleft ) takes constant time as opposed to O(n) time for lists.

**Resources**

1. Depth-and Breadth-First Search
2. Edd Mann
3. graph - Depth-first search in Python
4. DFS 和 BFS的特点
5. 能不能系统讲讲什么时候用BFS和DFS

# 总结

朴素解法，一步一步逻辑清晰地写出任务，分开把所有逻辑处理好。

}

32 K 个一组翻转链表

34 寻找最近的回文数