<https://leetcode.com/problems/word-ladder-ii/>

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter.

Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.

sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return *all the* ***shortest transformation sequences*** *from* beginWord *to* endWord*, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words* [beginWord, s1, s2, ..., sk].

**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"]]

Explanation: There are 2 shortest transformation sequences:

"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 there is no valid transformation sequence.

**Constraints:**

1 <= beginWord.length <= 5

endWord.length == beginWord.length

1 <= wordList.length <= 500

wordList[i].length == beginWord.length

beginWord, endWord, and wordList[i] consist of lowercase English letters.

beginWord != endWord

All the words in wordList are **unique**.

The **sum** of all shortest transformation sequences does not exceed 10^5.

**Attempt 1: 2023-01-06**

**Solution 1:  BFS + DFS Backtracking (?min)**

**Not get correct one yet, but BFS + DFS Backtracking  style is definitely the answer which no need to store all path in a queue**

**Revolution steps from wrong answer to TLE:**

**Version 1: Not deep copy of 'curPath' to create 'newPath', which modify 'curPath' and impact next stage since when we add 'curPath' back to the queue, if modified then bad result**

**Version 2: Not guaranteed shortest path**

Input

beginWord = "a"

endWord = "c"

wordList = ["a","b","c"]

22 / 36 testcases passed

Output

[["a","c"],["a","b","c"]]

Expected

[["a","c"]]

// Not guaranteed shortest path

import java.util.\*;

public class Solution {

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

        List<List<String>> result = new ArrayList<List<String>>();

        List<String> visited = new ArrayList<String>();

        // Have to initialize queue as List<String> type because only record

        // a word is not enough, we have to record how the paths developing

        // initialize by that word, each element in queue is a path, some

        // will add into final result as ending with 'endWord', others won't

        Queue<List<String>> q = new LinkedList<List<String>>();

        List<String> path = new ArrayList<String>();

        path.add(beginWord);

        q.offer(path);

        visited.add(beginWord);

        while(!q.isEmpty()) {

            int size = q.size();

            for(int i = 0; i < size; i++) {

                List<String> curPath = q.poll();

                String curWord = curPath.get(curPath.size() - 1);

                List<String> curNeighbours = getNeighbours(curWord, wordList);

                for(String neighbour : curNeighbours) {

                    if(!visited.contains(neighbour)) {

                        visited.add(neighbour);

                        //curPath.add(neighbour);

                        List<String> newPath = new ArrayList<String>(curPath);

                        newPath.add(neighbour);

                        if(neighbour.equals(endWord)) {

                            //result.add(curPath);

                            result.add(newPath);

                            // Have to remove 'endWord' for other candidate paths

                            // have to check 'endWord' again

                            visited.remove(endWord);

                        } else {

                            //q.offer(curPath);

                            q.offer(newPath);

                        }

                    }

                }

            }

        }

        return result;

    }

    private List<String> getNeighbours(String cur, List<String> wordList) {

        List<String> result = new ArrayList<String>();

        for(int i = 0; i < cur.length(); i++) {

            char[] chars = cur.toCharArray();

            for(char c = 'a'; c <= 'z'; c++) {

                chars[i] = c;

                String str = new String(chars);

                if(wordList.contains(str) && !result.contains(str)) {

                    result.add(str);

                }

            }

        }

        return result;

    }

    public static void main(String[] args) {

        Solution s = new Solution();

        String beginWord = "hit", endWord = "cog";

        List<String> wordList = new ArrayList<>();

        wordList.add("hot");

        wordList.add("dot");

        wordList.add("dog");

        wordList.add("lot");

        wordList.add("log");

        wordList.add("cog");

        List<List<String>> result = s.findLadders(beginWord, endWord, wordList);

        System.out.println(result);

    }

}

**Version 3: 'visited' set cannot share between candidate paths => besides endWord 'tax', 'tex' in the shared 'visited' set is a problem, when 2nd solution has 'ted' -> 'tex', then 'tex' added into 'visited' set, when 3rd solution attempt to do 'rex' -> 'tex', it blocked**

Input

beginWord = "red"

endWord = "tax"

wordList = ["ted","tex","red","tax","tad","den","rex","pee"]

26 / 36 testcases passed

Output

[["red","ted","tad","tax"],["red","ted","tex","tax"]]

Expected

[["red","ted","tad","tax"],["red","ted","tex","tax"],["red","rex","tex","tax"]]

// visited set cannot share between candidate paths ??? => besides endWord 'tax', 'tex' in the shared visited set is a problem,

// when 2nd solution has 'ted' -> 'tex', then 'tex' added into visited, when 3rd solution attempt to do 'rex' -> 'tex', it blocked

import java.util.\*;

public class Solution {

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

        List<List<String>> result = new ArrayList<List<String>>();

        Set<String> visited = new HashSet<String>();

        // Have to initialize queue as List<String> type because only record

        // a word is not enough, we have to record how the paths developing

        // initialize by that word, each element in queue is a path, some

        // will add into final result as ending with 'endWord', others won't

        Queue<List<String>> q = new LinkedList<List<String>>();

        List<String> path = new ArrayList<String>();

        path.add(beginWord);

        q.offer(path);

        visited.add(beginWord);

        while(!q.isEmpty()) {

            int size = q.size();

            for(int i = 0; i < size; i++) {

                List<String> curPath = q.poll();

                String curWord = curPath.get(curPath.size() - 1);

                List<String> curNeighbours = getNeighbours(curWord, wordList);

                for(String neighbour : curNeighbours) {

                    if(!visited.contains(neighbour)) {

                        visited.add(neighbour);

                        //curPath.add(neighbour);

                        List<String> newPath = new ArrayList<String>(curPath);

                        newPath.add(neighbour);

                        if(neighbour.equals(endWord)) {

                            // Condition 1: BFS guaranteed if no path added before, the first path is the shortest path

                            // Condition 2: If already have at least one shortest path, the new path should not oversize,

                            // and since BFS the new path size cannot less than previous found paths size, only able to

                            // equal to the previous found paths size

                            if(result.size() == 0 || newPath.size() == result.get(0).size()) {

                                //result.add(curPath);

                                result.add(newPath);

                                // Have to remove 'endWord' for other candidate paths

                                // have to check 'endWord' again

                                visited.remove(endWord);

                            }

                        } else {

                            //q.offer(curPath);

                            q.offer(newPath);

                        }

                    }

                }

            }

        }

        return result;

    }

    private List<String> getNeighbours(String cur, List<String> wordList) {

        List<String> result = new ArrayList<String>();

        for(int i = 0; i < cur.length(); i++) {

            char[] chars = cur.toCharArray();

            for(char c = 'a'; c <= 'z'; c++) {

                chars[i] = c;

                String str = new String(chars);

                if(wordList.contains(str) && !result.contains(str)) {

                    result.add(str);

                }

            }

        }

        return result;

    }

    public static void main(String[] args) {

        Solution s = new Solution();

//        String beginWord = "hit", endWord = "cog";

//        List<String> wordList = new ArrayList<>();

//        wordList.add("hot");

//        wordList.add("dot");

//        wordList.add("dog");

//        wordList.add("lot");

//        wordList.add("log");

//        wordList.add("cog");

        String beginWord = "a", endWord = "c";

        List<String> wordList = new ArrayList<>();

        wordList.add("a");

        wordList.add("b");

        wordList.add("c");

        List<List<String>> result = s.findLadders(beginWord, endWord, wordList);

        System.out.println(result);

    }

}

**Version 4: TLE version, based on Version 3, we cannot do visited check before add to path since we cannot exclude potential same word can be generated by different branches**

e.g ["red","ted","tex","tax"],["red","rex","tex","tax"]

For "ted -> tex" there is no issue for creating a new path, but soon as 'tex' will be added into 'visited' set and we do 'visited' check before we create new path in for loop like below:

    for(String neighbour : curNeighbours) {

        if(!visited.contains(neighbour)) {

            visited.add(neighbour);

            List<String> newPath = new ArrayList<String>(curPath);

            newPath.add(neighbour);

then we will miss the another potential path as "rex -> tex" which also generate 'tex' and blocked by 'tex' in 'visited'

**The solution is we don't do visited check before add to path, add neighbour word to path will not be blocked especially when in same level previous path also hit that neighbour word, instead, we will collect all visited neighbour words after build current level path, remove all these neighbour words from candidate word list (wordList), remain words are the only ones can be used in next level, no duplicate visit guaranteed**

import java.util.\*;

public class Solution {

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

        List<List<String>> result = new ArrayList<List<String>>();

        Set<String> visited = new HashSet<String>();

        // Have to initialize queue as List<String> type because only record

        // a word is not enough, we have to record how the paths developing

        // initialize by that word, each element in queue is a path, some

        // will add into final result as ending with 'endWord', others won't

        Queue<List<String>> q = new LinkedList<List<String>>();

        List<String> path = new ArrayList<String>();

        path.add(beginWord);

        q.offer(path);

        visited.add(beginWord);

        while(!q.isEmpty()) {

            int size = q.size();

            for(int i = 0; i < size; i++) {

                List<String> curPath = q.poll();

                String curWord = curPath.get(curPath.size() - 1);

                List<String> curNeighbours = getNeighbours(curWord, wordList);

                for(String neighbour : curNeighbours) {

                    //if(!visited.contains(neighbour)) {

                        visited.add(neighbour);

                        //curPath.add(neighbour);

                        List<String> newPath = new ArrayList<String>(curPath);

                        newPath.add(neighbour);

                        if(neighbour.equals(endWord)) {

                            // Condition 1: BFS guaranteed if no path added before, the first path is the shortest path

                            // Condition 2: If already have at least one shortest path, the new path should not oversize,

                            // and since BFS the new path size cannot less than previous found paths size, only able to

                            // equal to the previous found paths size

                            if(result.size() == 0 || newPath.size() == result.get(0).size()) {

                                //result.add(curPath);

                                result.add(newPath);

                                // Have to remove 'endWord' for other candidate paths

                                // have to check 'endWord' again

                                // remove this line since remove "if(!visited.contains(neighbour))" already

                                // even we add 'endWord' by "visited.add(neighbour)" in this level, we won't

                                // check next level

                                //visited.remove(endWord);

                            }

                        } else {

                            //q.offer(curPath);

                            q.offer(newPath);

                        }

                    //}

                }

            }

            // remove used words from wordList to avoid going back

            for(String s : visited) {

                wordList.remove(s);

            }

        }

        return result;

    }

    private List<String> getNeighbours(String cur, List<String> wordList) {

        List<String> result = new ArrayList<String>();

        for(int i = 0; i < cur.length(); i++) {

            char[] chars = cur.toCharArray();

            for(char c = 'a'; c <= 'z'; c++) {

                chars[i] = c;

                String str = new String(chars);

                if(wordList.contains(str) && !result.contains(str)) {

                    result.add(str);

                }

            }

        }

        return result;

    }

    public static void main(String[] args) {

        Solution s = new Solution();

//        String beginWord = "hit", endWord = "cog";

//        List<String> wordList = new ArrayList<>();

//        wordList.add("hot");

//        wordList.add("dot");

//        wordList.add("dog");

//        wordList.add("lot");

//        wordList.add("log");

//        wordList.add("cog");

//        String beginWord = "a", endWord = "c";

//        List<String> wordList = new ArrayList<>();

//        wordList.add("a");

//        wordList.add("b");

//        wordList.add("c");

        String beginWord = "red", endWord = "tax";

        List<String> wordList = new ArrayList<>();

        wordList.add("ted");

        wordList.add("tex");

        wordList.add("red");

        wordList.add("tax");

        wordList.add("tad");

        wordList.add("den");

        wordList.add("rex");

        wordList.add("pee");

        List<List<String>> result = s.findLadders(beginWord, endWord, wordList);

        System.out.println(result);

    }

}

**Version 5: TLE version, BFS + DFS version**

import java.util.\*;

public class Solution {

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

        List<List<String>> result = new ArrayList<List<String>>();

        Map<String, Integer> distance = new HashMap<>();

        Map<String, List<String>> neighbours = new HashMap<>();

        // Add 'beginWord' into wordList, otherwise will throw NullPointException

        wordList.add(beginWord);

        bfs(beginWord, endWord, wordList, distance, neighbours);

        dfs(beginWord, endWord, distance, neighbours, result, new ArrayList<String>());

        return result;

    }

    // BFS: Trace every node's distance from the start node(level by level) in BFS,

    // we can be sure that the distance of each node is the shortest one, because

    // once we have visited a node, we update the distance, if we first met one

    // node, it must be the shortest distance, if we met the node again, its

    // distance must not be less than the distance we first met and set.

    private void bfs(String beginWord, String endWord, List<String> wordList,

                    Map<String, Integer> distance, Map<String, List<String>> neighbours) {

        for(String word : wordList) {

            neighbours.put(word, new ArrayList<String>());

        }

        Queue<String> q = new LinkedList<>();

        q.offer(beginWord);

        // Distance from start to itself is 0

        distance.put(beginWord, 0);

        while(!q.isEmpty()) {

            int size = q.size();

            for (int i = 0; i < size; i++) {

                String cur = q.poll();

                List<String> curNeighbours = getNeighbours(cur, wordList);

                for (String neighbour : curNeighbours) {

                    // Add all the neighbors into map's current node entry

                    neighbours.get(cur).add(neighbour);

                    // If we have not previously visited this node (check not visited before

                    // to guarantee not update to larger distance if visited same node again)

                    if (!distance.containsKey(neighbour)) {

                        distance.put(neighbour, distance.get(cur) + 1);

                        // If reach the end

                        if (neighbour.equals(endWord)) {

                            // We don't want to return as there may well be more paths from

                            // start to end. So we need to break inner for loop instead,

                            // the next step is go back to outer for loop and poll out

                            // another string if exists

                            break;

                        } else {

                            q.offer(neighbour);

                        }

                    }

                }

            }

        }

    }

    // DFS: output all paths with the shortest distance.

    // This is backtracking

    private void dfs(String beginWord, String endWord, Map<String, Integer> distance,

                    Map<String, List<String>> neighbours, List<List<String>> result, List<String> path) {

        path.add(beginWord);

        // Base case

        if(beginWord.equals(endWord)) {

            // Deep copy

            result.add(new ArrayList<>(path));

        }

        for(String neighbour : neighbours.get(beginWord)) {

            // In case that the next node is the next level of current node, otherwise it can be

            // one of the parent nodes of current node, or it is not the shortest node (visit again)

            // Since in BFS, we record both the next level nodes and the parent node as neighbors

            // of current node, use "distance.get(beginWord) + 1" to make sure the path is the

            // shortest one

            if(distance.get(neighbour) == distance.get(beginWord) + 1) {

                dfs(neighbour, endWord, distance, neighbours, result, path);

            }

        }

        // Back tracking

        // You keep on going down the 'tree' until you get to 'endWord'. Add the sequence to the results.

        // Then you need to backtrack the neighbours and travel down the tree on a different path.

        path.remove(path.size() - 1);

    }

    // DFS style 2, more intuitive to follow conventional format

    private void dfs2(String beginWord, String endWord, Map<String, Integer> distance,

                    Map<String, List<String>> neighbours, List<List<String>> result, List<String> path) {

        //path.add(beginWord);

        // Base case

        if(beginWord.equals(endWord)) {

            // Have to add 'beginWord' and backtrack again because before calling current

            // level DFS, we only have "path.add(beginWord)" to add the previous level

            // 'beginWord' onto the path, it is one step away from its 'neighbour', and the

            // 'neighbour' we pass into current level DFS will transfer into a new 'beginWord'

            // for current level, when this new 'beginWord' equal to the 'endWord', we didn't

            // record it onto path yet, to make it up, have to add and backtrack onto path again

            path.add(beginWord);

            // Deep copy

            result.add(new ArrayList<>(path));

            path.remove(path.size() - 1);

            return;

        }

        for(String neighbour : neighbours.get(beginWord)) {

            // In case that the next node is the next level of current node, otherwise it can be

            // one of the parent nodes of current node, or it is not the shortest node (visit again)

            // Since in BFS, we record both the next level nodes and the parent node as neighbors

            // of current node, use "distance.get(beginWord) + 1" to make sure the path is the

            // shortest one

            if(distance.get(neighbour) == distance.get(beginWord) + 1) {

                path.add(beginWord);

                dfs2(neighbour, endWord, distance, neighbours, result, path);

                path.remove(path.size() - 1);

            }

        }

        // Back tracking

        // You keep on going down the 'tree' until you get to 'endWord'. Add the sequence to the results.

        // Then you need to backtrack the neighbours and travel down the tree on a different path.

        //path.remove(path.size() - 1);

    }

    private List<String> getNeighbours(String cur, List<String> wordList) {

        List<String> result = new ArrayList<>();

        for(int i = 0; i < cur.length(); i++) {

            char[] chars = cur.toCharArray();

            for(char c = 'a'; c <= 'z'; c++) {

                chars[i] = c;

                String str = new String(chars);

                if(wordList.contains(str) && !result.contains(str)) {

                    result.add(str);

                }

            }

        }

        return result;

    }

    public static void main(String[] args) {

        Solution s = new Solution();

//        String beginWord = "hit", endWord = "cog";

//        List<String> wordList = new ArrayList<>();

//        wordList.add("hot");

//        wordList.add("dot");

//        wordList.add("dog");

//        wordList.add("lot");

//        wordList.add("log");

//        wordList.add("cog");

//        String beginWord = "a", endWord = "c";

//        List<String> wordList = new ArrayList<>();

//        wordList.add("a");

//        wordList.add("b");

//        wordList.add("c");

        String beginWord = "red", endWord = "tax";

        List<String> wordList = new ArrayList<>();

        wordList.add("ted");

        wordList.add("tex");

        wordList.add("red");

        wordList.add("tax");

        wordList.add("tad");

        wordList.add("den");

        wordList.add("rex");

        wordList.add("pee");

        List<List<String>> result = s.findLadders(beginWord, endWord, wordList);

        System.out.println(result);

    }

}

**Refer to**

[https://leetcode.com/problems/word-ladder-ii/solutions/40475/my-concise-java-solution-based-on-bfs-and-dfs](https://leetcode.com/problems/word-ladder-ii/solutions/40475/my-concise-java-solution-based-on-bfs-and-dfs/?orderBy=most_votes)

class Solution {

public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

List<List<String>> results = new ArrayList<List<String>>();

Set<String> dictionary = new HashSet<String>(wordList);

// Neighbors of every node

Map<String, List<String>> nodeNeighbors = new HashMap<String, List<String>>();

// Distance of every node from start node

Map<String, Integer> distance = new HashMap<String, Integer>();

List<String> solution = new ArrayList<String>();

// Add 'beginWord' into dictionary, otherwise will throw NullPointException

// on line81 because of no entry for 'beginWord' in map

dictionary.add(beginWord);

bfs(beginWord, endWord, dictionary, nodeNeighbors, distance);

dfs(beginWord, endWord, nodeNeighbors, distance, solution, results);

return results;

}

// BFS: Trace every node's distance from the start node(level by level) in BFS,

// we can be sure that the distance of each node is the shortest one, because

// once we have visited a node, we update the distance, if we first met one

// node, it must be the shortest distance, if we met the node again, its

// distance must not be less than the distance we first met and set.

private void bfs(String start, String end, Set<String> dictionary, Map<String, List<String>> nodeNeighbors, Map<String, Integer> distance) {

// For each word in dictionary set up hashmap to store all of its adjacent neighbors

for(String str : dictionary) {

nodeNeighbors.put(str, new ArrayList<String>());

}

Queue<String> queue = new LinkedList<String>();

// Add very start of beginWord to start BFS.

queue.offer(start);

// Distance from start to itself is 0

distance.put(start, 0);

while(!queue.isEmpty()) {

// You just want to loop each level's neighbors which is just the size

// of the queue at the current level.

int size = queue.size();

for(int i = 0; i < size; i++) {

String cur = queue.poll();

// Get the current node's neighbors

List<String> currentNodesNeighbors = getNeighbors(cur, dictionary);

// Loop through current node's neighbors

for(String neighbor : currentNodesNeighbors) {

// Add all the neighbors into map's current node entry

nodeNeighbors.get(cur).add(neighbor);

// If we have not previously visited this node

if(!distance.containsKey(neighbor)) {

// Increment distance we have traveled

distance.put(neighbor, distance.get(cur) + 1);

// If reach the end

if(end.equals(neighbor)) {

// We don't want to return as there may well be more paths from

// start to end. So we need to break instead.

break;

} else {

// Add neighbor for next level

queue.offer(neighbor);

}

}

}

}

}

}

// DFS: output all paths with the shortest distance.

// This is backtracking

individualSequence.add(beginWord);

// Base case for beginWord that has reached endWord and individualSequence has the complete

// path from the beginWord to endWord. So add to the final results list.

if(endWord.equals(beginWord)) {

// Deep copy

results.add(new ArrayList<String>(individualSequence));

}

// We have not reached the end so we must perform DFS on the neighbors.

for(String neighbor : nodeNeighbors.get(beginWord)) {

// This is just in case that the next node is the next level of current node， otherwise

// it can be one of the parent nodes of current node，or it is not the shortest node

// Since in BFS, we record both the next level nodes and the parent node as neighbors

// of current node. use distance.get(beginWord)+1 we can make sure the path is the

// shortest one.

if(distance.get(neighbor) == distance.get(beginWord) + 1) {

// Be careful: Use 'neighbor' instead of 'beginWord' for next traverse

dfs(neighbor, endWord, nodeNeighbors, distance, individualSequence, results);

}

}

// Back tracking

// You keep on going down the 'tree' until you get to 'cog'. Add the sequence to the results.

// Then you need to backtrack the neighbors and travel down the tree on a different path.

// In this example go to '3' then travel down that path to 'cog' again.

// Keep removing until you backtrack to the very start of beginWord which in this case is "hit".

individualSequence.remove(individualSequence.size() - 1);

}

// Find all next level nodes

private List<String> getNeighbors(String node, Set<String> dictionary) {

List<String> neighbors = new ArrayList<String>();

for(int i = 0; i < node.length(); i++) {

for(char c = 'a'; c <= 'z'; c++) {

if(c == node.charAt(i)) {

continue;

}

String neighbor = replace(node, i, c);

if(dictionary.contains(neighbor)) {

neighbors.add(neighbor);

}

}

}

return neighbors;

}

private String replace(String node, int index, char c) {

char[] chars = node.toCharArray();

chars[index] = c;

return new String(chars);

}

}

**Rework with chatGPT**

class Solution {

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

        // The 'predecessorMap' variable serves to store the connections between

        // words where each key is a word, and the corresponding value is a list

        // of words that can be transformed into the key word by changing one

        // character. This relationship is built during the BFS traversal and is

        // used later to perform DFS to find all paths from endWord to beginWord.

        Map<String, List<String>> predecessorMap = new HashMap<>();

        Set<String> dict = new HashSet<>(wordList);

        Set<String> visited = new HashSet<>();

        Set<String> toVisit = new HashSet<>();

        Queue<String> q = new LinkedList<>();

        q.offer(beginWord);

        visited.add(beginWord);

        boolean endWordFound = false;

        while(!q.isEmpty() && !endWordFound) {

            int size = q.size();

            toVisit.clear();

            for(int i = 0; i < size; i++) {

                String curWord = q.poll();

                char[] chars = curWord.toCharArray();

                for(int j = 0; j < chars.length; j++) {

                    char originalChar = chars[j];

                    for(char c = 'a'; c <= 'z'; c++) {

                        if(c != originalChar) {

                            chars[j] = c;

                            String nextWord = new String(chars);

                            if(dict.contains(nextWord)) {

                                if(nextWord.equals(endWord)) {

                                    endWordFound = true;

                                }

                                if(!visited.contains(nextWord)) {

                                    // Do not add 'nextWord' immediately in visited list when encounter,

                                    // add all of them to visited list only after finish all current

                                    // level words' next words calculation as 'visited.addAll(toVisit)'

                                    // in a following step, which unblock all potential paths reach to

                                    // the same 'nextWord', otherwise, if add the 'nextWord' immediately

                                    // after first detection from a current level word, it will block

                                    // all later current level words potential transformation into this

                                    // same 'nextWord', hence block all other potential paths

                                    toVisit.add(nextWord);

                                    predecessorMap.computeIfAbsent(nextWord, k -> new ArrayList<>()).add(curWord);

                                }

                            }

                        }

                    }

                    chars[j] = originalChar;

                }

            }

            visited.addAll(toVisit);

            for(String word : toVisit) {

                q.offer(word);

            }

        }

        List<List<String>> result = new ArrayList<>();

        if(endWordFound) {

            List<String> path = new ArrayList<>();

            path.add(endWord);

            helper(path, result, beginWord, endWord, predecessorMap);

        }

        return result;

    }

    private void helper(List<String> path, List<List<String>> result, String beginWord, String word, Map<String, List<String>> predecessorMap) {

        // Base condition: encounter 'beginWord' means current path end

        if(word.equals(beginWord)) {

            List<String> tmp = new ArrayList<>(path);

            Collections.reverse(tmp);

            result.add(tmp);

            return;

        }

        // Backtracking to find all paths

        if(predecessorMap.containsKey(word)) {

            for(String predecessor : predecessorMap.get(word)) {

                path.add(predecessor);

                helper(path, result, beginWord, predecessor, predecessorMap);

                path.remove(path.size() - 1);

            }

        }

    }

}

Time Complexity

The algorithm performs a breadth-first search (BFS) to find the shortest path from beginWord to endWord,

while also recording all predecessors of each word along the way. In the worst-case scenario, every word

in the word list can be connected to every other word, leading to a complete graph-like structure. BFS in

this case would have a time complexity of O(V + E), where V is the number of vertices (words) and E is the

number of edges (possible transformations). For each word, there are L possible single-letter changes, and

each change has 26 possible outcomes (the alphabet size), so the total number of edges would be O(26 \* L \* V).

The time complexity can be further broken down as follows:

BFS to find the shortest paths and record predecessors: O(L \* 26 \* V), where L is the length of each word,

and V is the number of words in the word list.

Depth-first search (DFS) to reconstruct paths: In the worst case, all paths from beginWord to endWord will

need to be found. If there are P such shortest paths and the depth of the search is D (corresponding to the

number of transformations needed), this operation is O(P\*D).

Overall, the time complexity of this algorithm would be O(L \* 26 \* V + P \* D).

Space Complexity

The space complexity is primarily governed by the storage of words, the queue used in BFS, and the recursive

stack for DFS.

Words storage: O(V), where V is the size of the wordList.

BFS Queue: O(V), the queue could at most contain all the words.

BFS prev map and dist map: These maps will store at most O(V) entries if every word is encountered.

DFS Paths: Up to O(P \* D) space may be needed to store all P paths, each with a depth of maximum D.

Thus, the total space complexity is O(V + P \* D).

**1. Refer to chatGPT Solution:**

Leetcode 126, "Word Ladder II," is a problem that requires finding all the shortest transformation sequences from a starting word to an ending word using a given word list. This problem can be solved using a combination of Breadth-First Search (BFS) for finding the shortest paths and Depth-First Search (DFS) for reconstructing the paths.

**Solution**

Here's a solution that uses BFS to build the shortest paths and then DFS to collect all valid paths:

import java.util.\*;

class Solution {

public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

List<List<String>> result = new ArrayList<>();

Set<String> wordSet = new HashSet<>(wordList);

if (!wordSet.contains(endWord)) {

return result;

}

Map<String, List<String>> graph = new HashMap<>();

Set<String> visited = new HashSet<>();

Set<String> toVisit = new HashSet<>();

Queue<String> queue = new LinkedList<>();

queue.offer(beginWord);

visited.add(beginWord);

boolean foundEndWord = false;

while (!queue.isEmpty() && !foundEndWord) {

int levelSize = queue.size();

toVisit.clear();

for (int i = 0; i < levelSize; i++) {

String currentWord = queue.poll();

char[] wordChars = currentWord.toCharArray();

for (int j = 0; j < wordChars.length; j++) {

char originalChar = wordChars[j];

for (char c = 'a'; c <= 'z'; c++) {

if (c == originalChar) continue;

wordChars[j] = c;

String nextWord = new String(wordChars);

if (wordSet.contains(nextWord)) {

if (nextWord.equals(endWord)) {

foundEndWord = true;

}

if (!visited.contains(nextWord)) {

toVisit.add(nextWord);

graph.computeIfAbsent(nextWord, k -> new ArrayList<>()).add(currentWord);

}

}

}

wordChars[j] = originalChar;

}

}

visited.addAll(toVisit);

for (String word : toVisit) {

queue.offer(word);

}

}

if (foundEndWord) {

List<String> path = new ArrayList<>();

path.add(endWord);

dfs(result, path, endWord, beginWord, graph);

}

return result;

}

private void dfs(List<List<String>> result, List<String> path, String word, String beginWord, Map<String, List<String>> graph) {

if (word.equals(beginWord)) {

List<String> temp = new ArrayList<>(path);

Collections.reverse(temp);

result.add(temp);

return;

}

if (graph.containsKey(word)) {

for (String nextWord : graph.get(word)) {

path.add(nextWord);

dfs(result, path, nextWord, beginWord, graph);

path.remove(path.size() - 1);

}

}

}

public static void main(String[] args) {

Solution solution = new Solution();

String beginWord = "hit";

String endWord = "cog";

List<String> wordList = Arrays.asList("hot","dot","dog","lot","log","cog");

System.out.println(solution.findLadders(beginWord, endWord, wordList));

}

}

**Explanation**

Initialization:

Convert the word list to a set for fast lookups.

Create a graph (map) to store the transformation relationships.

Use visited to keep track of visited words and toVisit to track the words to visit in the current level.

Initialize a queue with the beginWord.

BFS for Shortest Paths:

Perform BFS to explore all possible transformations level by level.

For each word at the current level, generate all possible one-character transformations.

If the transformation is in the word list and not yet visited, add it to the graph and toVisit.

If the endWord is found, set foundEndWord to true and stop further exploration.

DFS for Path Reconstruction:

If the endWord is found, use DFS to construct all paths from endWord to beginWord using the graph.

The DFS adds the current word to the path, recursively explores its predecessors, and adds the completed path to the result.

Output:

The final result contains all shortest transformation sequences.

This solution ensures that all shortest paths are found using BFS and then reconstructed using DFS.

**2. Wrong Solution by just modify "visited" logic:**

**i just modify one line as below:add "visited.add(nextWord);" inside "if (!visited.contains(nextWord)) {...}" and remove"visited.addAll(toVisit);"**

**but it error out as below:**

**Input**

**beginWord = "hit"**

**endWord = "cog"**

**wordList = ["hot","dot","dog","lot","log","cog"]**

**Output[["hit","hot","lot","log","cog"]]**

**Expected[["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]**

class Solution {

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

        List<List<String>> result = new ArrayList<>();

        Set<String> wordSet = new HashSet<>(wordList);

        if (!wordSet.contains(endWord)) {

            return result;

        }

        Map<String, List<String>> graph = new HashMap<>();

        Set<String> visited = new HashSet<>();

        Set<String> toVisit = new HashSet<>();

        Queue<String> queue = new LinkedList<>();

        queue.offer(beginWord);

        visited.add(beginWord);

        boolean foundEndWord = false;

        while (!queue.isEmpty() && !foundEndWord) {

            int levelSize = queue.size();

            toVisit.clear();

            for (int i = 0; i < levelSize; i++) {

                String currentWord = queue.poll();

                char[] wordChars = currentWord.toCharArray();

                for (int j = 0; j < wordChars.length; j++) {

                    char originalChar = wordChars[j];

                    for (char c = 'a'; c <= 'z'; c++) {

                        if (c == originalChar) continue;

                        wordChars[j] = c;

                        String nextWord = new String(wordChars);

                        if (wordSet.contains(nextWord)) {

                            if (nextWord.equals(endWord)) {

                                foundEndWord = true;

                            }

                            if (!visited.contains(nextWord)) {

                                toVisit.add(nextWord);

// Relocate the "visited.addAll(toVisit)" here

                                visited.add(nextWord);

                                graph.computeIfAbsent(nextWord, k -> new ArrayList<>()).add(currentWord);

                            }

                        }

                    }

                    wordChars[j] = originalChar;

                }

            }

            //visited.addAll(toVisit);

            for (String word : toVisit) {

                queue.offer(word);

            }

        }

        if (foundEndWord) {

            List<String> path = new ArrayList<>();

            path.add(endWord);

            dfs(result, path, endWord, beginWord, graph);

        }

        return result;

    }

    private void dfs(List<List<String>> result, List<String> path, String word, String beginWord, Map<String, List<String>> graph) {

        if (word.equals(beginWord)) {

            List<String> temp = new ArrayList<>(path);

            Collections.reverse(temp);

            result.add(temp);

            return;

        }

        if (graph.containsKey(word)) {

            for (String nextWord : graph.get(word)) {

                path.add(nextWord);

                dfs(result, path, nextWord, beginWord, graph);

                path.remove(path.size() - 1);

            }

        }

    }

}

**3. Analysis from chatGPT about the wrong solution:**

The change you made causes the BFS to prematurely mark nodes as visited, which interferes with the correct exploration of all paths at the current level. Here's why this modification fails to produce the correct result:

**Original Behavior**

In the original solution, nodes are added to the visited set only after the entire current level of the BFS is processed:

for (String word : toVisit) {

queue.offer(word);

}

visited.addAll(toVisit);

**Modified Behavior**

In the modified solution, nodes are added to the visited set immediately within the same BFS level processing loop:

if (!visited.contains(nextWord)) {

toVisit.add(nextWord);

visited.add(nextWord);

graph.computeIfAbsent(nextWord, k -> new ArrayList<>()).add(currentWord);

}

**Explanation of Failure**

**The problem arises because when a word is marked as visited immediately after it's discovered, it prevents the BFS from exploring all possible paths to that word from the current level. This is crucial in BFS because it ensures that all shortest paths are explored. By marking a word as visited immediately, you prevent other possible paths (at the same BFS level) from being considered, leading to an incomplete graph construction.**

**Why This Matters**

**For example, consider both same level (level 3) words "log" and "dog" able to transform into same word "cog" (level 4) as "log" -> "cog" and "dog" -> "cog". In BFS, we want to explore both "log" and "dog" potential transformations completely before moving to the next level. But when process "log" and "dog" one after another, you will see blocking issue happen as "cog" marked into "visited" list immediately after "log" -> "cog" transformation happens first, and when perform same level (level 3) word "dog" -> "cog" transformation, since "cog" already in "visited" list, in the graph we only record "log" -> "cog" path, but missing "dog" -> "cog" path, the immediate adding next level word into "visited" list prevents other same level words transform to that word, resulting in missing paths in the final solution.**

**Correct Approach**

To maintain the correct BFS level order processing, you should only mark nodes as visited after the current level is fully processed:

for (String word : toVisit) {

queue.offer(word);

}

visited.addAll(toVisit);

This ensures that all paths at the current level are explored and added to the graph before moving on to the next level.

**Visualize the missing path issue in detail explain**

Input

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log","cog"]

Use Testcase

Output

[["hit","hot","lot","log","cog"]]

Expected

[["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]

Wrong one:

"visited" change step by step:

hit -> hit hot -> hit hot dot -> hit hot dot lot -> hit hot dot lot log -> hit hot dot lot log dog -> hit hot dot lot log dog cog

transfer process level by level (split by ###)

lvl0->1 lvl1->2 lvl2->3 lvl3->4

hit->hot ### hot->dot hot->lot ### lot->dot(x) lot->hot(x) lot->log dot->hot(x) dot->lot(x) dot->dog ### log->cog log->dog(x) log->lot(x) dog->cog dog->log(x) dog->dot(x)

Visualize into level by level transform:

hit level 0

hot level 1

[branch 1] [branch 2]

lot dot level 2

dot(x) hot(x) log hot(x) lot(x) dog level 3

cog dog(x) lot(x) cog(x) log(x) dot(x) level 4

^ ^

After "log - cog" transfer in same level transfer between level 3 to 4 but in branch 2

between level 3 and 4, the since "cog" add into visited in branch 1 immediately after log->cog,

"cog" even as endWord also if (!visited.contains(nextWord)) {...} block us to add "cog - dog"

add into "visited" now: into graph, so graph only have one mapping for "cog" as "cog - log",

[hit hot dot lot log dog cog] instead of the correct two mappings as "cog - log + dog"

graph add "cog - log"

Final graph

lot -> hot

log -> lot

dot -> hot

cog -> log

hot -> hit

dog -> dot

Correct one:

"visited" change step by step:

hit -> hit hot -> hit hot lot dot -> hit hot lot dot log dog -> hit hot lot dot log dog cog

Final graph

lot -> hot

log -> lot

dot -> hot

cog -> log dog

hot -> hit

dog -> dot

In correct one, "log"("lot" -> "log") and "dog"("dot" -> "dog") added into visited list by one time,

during this process, mapping in graph happened for both "cog ->log" and "cog -> dog"

**4. Why Leetcode 127 don't have similar logic (add words into visited list only after current level done) ?**

Leetcode 127 and Leetcode 126, while similar in concept, differ significantly in their requirements. Leetcode 127 requires only the length of the shortest transformation sequence, whereas Leetcode 126 requires all possible shortest transformation sequences. This difference in requirements fundamentally affects how the visited set is managed during the BFS traversal.

**Leetcode 127: Word Ladder (Length of Shortest Transformation Sequence)**

**Goal:** Find the length of the shortest transformation sequence from beginWord to endWord.

**Approach:** Since only the length of the shortest transformation sequence is needed, it is sufficient to mark nodes as visited as soon as they are discovered in the BFS traversal. This is because once a word is visited, any subsequent encounters of that word in the same level or deeper levels will not yield a shorter path.

**Key Points:**

Marking nodes as visited immediately ensures no cycles.

We only care about reaching the endWord as quickly as possible.

**Leetcode 126: Word Ladder II (All Shortest Transformation Sequences)**

**Goal:** Find all shortest transformation sequences from beginWord to endWord.

**Approach:** To ensure all shortest paths are discovered, nodes should be marked as visited only after processing all nodes at the current BFS level. This ensures that all paths of the same length are explored and added to the graph before moving on to the next level.

**Key Points:**

**Delaying the marking of nodes as visited allows capturing all possible shortest paths.**

**Each word can be part of multiple paths, so marking them prematurely can miss valid paths.**

**Why Leetcode 127 Does Not Need the Delayed Visited Marking**

In Leetcode 127, the primary objective is to determine the shortest path length. Since BFS inherently finds the shortest path in an unweighted graph, marking nodes as visited immediately upon discovery prevents redundant paths and cycles, simplifying the solution.

Here's a concise explanation of why the immediate visited marking works for Leetcode 127:

BFS Level Order Traversal: BFS ensures that the first time a node (word) is visited, it is through the shortest path. Subsequent encounters of the same node will only be through longer paths.

Early Termination: As soon as the endWord is encountered, the search can terminate early because we have found the shortest path.

**Key Differences Summarized**

Leetcode 127: Immediate visited marking is sufficient because it only needs the shortest path length.

Leetcode 126: Requires delayed visited marking to ensure all shortest paths are captured.

By understanding the differing requirements, we can see why Leetcode 127's approach doesn't directly translate to Leetcode 126. The nature of BFS ensures that marking nodes immediately in Leetcode 127 suffices for finding the shortest path, whereas Leetcode 126's need for all shortest paths requires a more cautious approach to visited marking.

**5. What's the variable 'graph' used in above solution ?**

In the provided solution, the variable graph is a Map<String, List<String>> that is used to build an adjacency list representation of the word ladder graph. This graph helps in tracking the parent-child relationships between words as the BFS traverses through different levels.

Here's a detailed explanation of how graph is used in the solution:

**Purpose of graph**

The graph variable serves to store the connections between words where each key is a word, and the corresponding value is a list of words that can be transformed into the key word by changing one character. This relationship is built during the BFS traversal and is used later to perform DFS to find all paths from endWord to beginWord.

**Building the Graph**

During the BFS traversal, for each word in the current level, all possible transformations (by changing one character) are considered. If a valid transformation is found (i.e., the transformed word is in the wordSet), the graph is updated to record this transformation. Specifically, the transformed word is added as a key, and the current word is added to its list of predecessors.

**Code Explanation**

Here is the relevant part of the BFS code that builds the graph:

while (!queue.isEmpty() && !foundEndWord) {

int levelSize = queue.size();

toVisit.clear();

for (int i = 0; i < levelSize; i++) {

String currentWord = queue.poll();

char[] wordChars = currentWord.toCharArray();

for (int j = 0; j < wordChars.length; j++) {

char originalChar = wordChars[j];

for (char c = 'a'; c <= 'z'; c++) {

if (c == originalChar) continue;

wordChars[j] = c;

String nextWord = new String(wordChars);

if (wordSet.contains(nextWord)) {

if (nextWord.equals(endWord)) {

foundEndWord = true;

}

if (!visited.contains(nextWord)) {

toVisit.add(nextWord);

// Update the graph to reflect the transformation

graph.computeIfAbsent(nextWord, k -> new ArrayList<>()).add(currentWord);

}

}

}

wordChars[j] = originalChar;

}

}

visited.addAll(toVisit);

for (String word : toVisit) {

queue.offer(word);

}

}

Using the Graph for DFS

After the BFS completes, if the endWord is found, a DFS is performed using the graph to find all paths from endWord to beginWord. The graph helps to backtrack from endWord to beginWord using the recorded transformations.

Here is the DFS part of the code:

if (foundEndWord) {

List<String> path = new ArrayList<>();

path.add(endWord);

dfs(result, path, endWord, beginWord, graph);

}

private void dfs(List<List<String>> result, List<String> path, String word, String beginWord, Map<String, List<String>> graph) {

if (word.equals(beginWord)) {

List<String> temp = new ArrayList<>(path);

Collections.reverse(temp);

result.add(temp);

return;

}

if (graph.containsKey(word)) {

for (String nextWord : graph.get(word)) {

path.add(nextWord);

dfs(result, path, nextWord, beginWord, graph);

path.remove(path.size() - 1);

}

}

}

**Summary**

Graph Building (BFS): During the BFS traversal, the graph records which words can be transformed into each other by one character change.

Path Finding (DFS): After the BFS completes, the graph is used to backtrack from endWord to beginWord to find all shortest transformation paths.

This approach ensures that all valid transformation sequences are captured and returned, leveraging both BFS for shortest path discovery and DFS for path reconstruction.

**Refer to**

<https://algo.monster/liteproblems/126>

**Problem Description**

The problem presents a challenge of finding all the shortest transformation sequences that convert a beginWord to an endWord using a given dictionary wordList. A transformation sequence is made up of intermediate words where each word in the sequence differs by exactly one letter from the previous word. The sequence starts with beginWord, ends with endWord, and each intermediate word must be found in wordList (although beginWord itself does not need to be in wordList). If there is no possible transformation that leads to endWord, the function should return an empty list.

A key point to understand is that the sequences required are the "shortest" possible. This means that if it's possible to transform beginWord to endWord in 4 steps, any sequences that take 5 or more steps should not be included in the output.

**Intuition**

The intuition behind the solution involves two parts: first, using Breadth-First Search (BFS) to find the distance of the shortest paths from beginWord to all other possible words in wordList, and second, using Depth-First Search (DFS) to reconstruct all the paths from endWord back to beginWord that match the shortest distance tracked by BFS.

Here's the step-by-step thought process:

**BFS to find the shortest distance:** The solution initializes a queue with beginWord and uses BFS to expand outwards to all possible transformations. While doing this, the solution tracks two key pieces of information: the distance (dist) from beginWord to each word encountered, and a mapping (prev) from each word to its possible precursors in the sequence. Using BFS ensures that we encounter the shortest path to each word first, and therefore when we reach endWord, we have found the shortest distance to it.

**Tracking predecessor words:** As the BFS proceeds, for each word encountered that is a possible next step from the current word, we track its predecessors - these are words one step closer to beginWord. We only store the predecessors that are part of the shortest path, effectively discarding longer paths.

**DFS to reconstruct paths:** Once BFS finishes and if endWord has been reached, we use DFS starting from endWord to traverse back through the prev mapping, constructing all possible shortest paths to beginWord. Recursive DFS is used because it allows to backtrack (remove the last added word) when a path is fully constructed, which helps to navigate to other potential paths with shared intermediate words.

**Complete the transformation sequences:** The paths constructed from endWord to beginWord are reversed before being added to the final answer list to match the required format (beginWord to endWord).

Through this blend of BFS and DFS, we can efficiently track all the shortest transformation sequences that meet the criteria of the problem.

**Solution Approach**

The solution follows a two-phase approach: first, it uses Breadth-First Search (BFS) to swiftly find the shortest path lengths from beginWord to endWord and all intermediate words, and subsequently, Depth-First Search (DFS) is applied to backtrack and assemble all the shortest transformation sequences. Let's delve into how the solution implements this methodologically:

**BFS (Breadth-First Search)**

**Queue Initialization:** We begin with a queue (q) initialized with the beginWord.

**Distance and Predecessor Tracking:** Two dictionaries are used - dist holds the number of steps from the beginWord to the current word, and prev maintains a set of all possible preceding words for each word encountered.

**Word Processing:** As long as there are words in the queue and the endWord has not been found, the algorithm processes each word by:

Popping the current word from the queue.

Generating all possible single-letter transformations of the current word.

For each valid transformation that is either in the wordList or previously encountered at the same BFS level (indicating the shortest path to that word):

Add the current word to the prev set of the transformed word.

If the transformed word is endWord, mark that endWord is found.

If it's a new word not seen before at a closer level, add it to the queue with an incremented distance value.

**Termination:** Once the endWord is found or the queue is exhausted, BFS terminates.

**DFS (Depth-First Search) with backtracking**

**Path Reconstruction:** Starting with endWord, the algorithm uses DFS in a recursive manner to rebuild all possible paths that lead to it from beginWord, given the information in the prev dictionary.

**DFS Function:** A helper function dfs is defined for the purpose of traversing the prev map. It:

Adds the current word to the path.

**If the current word is the beginWord, it means a complete path has been found, so it is reversed and added to the results.**

If not, the function calls itself recursively for each predecessor of the current word (each possible previous step in the sequence).

**After each recursive call, the last word is removed from the path (backtracking), allowing for exploration of alternative paths.**

**Efficiency**

The design of the solution ensures efficiency by:

**Using BFS for Level-by-Level Exploration:** By expanding the search level by level, we first reach endWord at its shortest possible path because BFS explores the closest nodes first.

**Storing Predecessor Words:** This avoids the need to compute all transformations again during the DFS phase, as we can directly access all valid previous words.

**Pruning Non-Shortest Paths:** By removing words from the wordList once they are visited at the current level, the algorithm eliminates any longer paths that might be visited in the future.

**Execution**

The algorithms and data structures work hand in hand like cogs in a complex machine, with BFS setting the groundwork by identifying all shortest distances and predecessors, and DFS taking up the final stretch by piecing together the information into complete paths. This interplay allows the solution to compile all shortest transformation sequences as per the problem's requirements.

**Example Walkthrough**

Let's consider a small example to illustrate the solution approach described above.

Suppose we have the following inputs:

beginWord: "hit"

endWord: "cog"

wordList: ["hot","dot","dog","lot","log","cog"]

Now, let us walk through the BFS and DFS method to find all the shortest transformation sequences.

**BFS (Breadth-First Search)**

Initialize a queue with the beginWord, which is "hit".

Create dist and prev dictionaries to keep track of distances and predecessors.

Process words one by one. Starting with "hit", we examine possible one-letter transformations: "hot".

Since "hot" is a valid word in wordList, we add "hit" as its predecessor, set its distance, and enqueue it.

Now, the queue has "hot". Dequeue and process it, checking its possible transformations: "dot", "lot".

Both words are valid in wordList and are enqueued, with "hot" as their predecessor.

This process continues, with words like "dot" leading to "dog", "lot" to "log", and so on, until "cog" is reached.

Through BFS, we find the shortest path to "cog" is 4 steps: hit -> hot -> dot -> dog -> cog.

**DFS (Depth-First Search) with backtracking**

We start the DFS with endWord which is "cog".

From "cog", we look up its predecessors ("dog"), and recursively call DFS on "dog".

"dog" leads us back to its predecessors ("dot"), and DFS continues with "dot".

The process is repeated until we reach "hit", which has no predecessors.

Each time we hit "hit", we have found a complete path, so we reverse it and add it to our results list.

**Each recursive call we remove the last word from the path (backtracking), allowing for exploration of alternative paths, for example on "cog" besides predecessor as "dog", it also has predecessor as "log", to find the second path we need backtracking to help**

Through DFS, we find the following paths:

["hit", "hot", "dot", "dog", "cog"]

["hit", "hot", "lot", "log", "cog"]

**Efficiency**

This example demonstrates how BFS ensures we hit the endWord with the minimum number of steps, and DFS helps us reconstruct all paths that follow this shortest distance.

**Execution**

Putting both phases together, the solution collects all the shortest transformation sequences from beginWord to endWord using the provided wordList. With this method, the problem is solved efficiently and the paths provided represent all the shortest ways to transform the starting word into the ending word using the given dictionary of words.

**Solution Implementation**

class Solution {

private List<List<String>> allPaths; // List of paths from beginWord to endWord

private Map<String, Set<String>> predecessorsMap; // Map to track the predecessors of each word in the shortest paths

public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

allPaths = new ArrayList<>();

Set<String> wordSet = new HashSet<>(wordList); // Convert word list to a set for efficient lookups

if (!wordSet.contains(endWord)) {

return allPaths; // If endWord is not in the word list, return empty list

}

wordSet.remove(beginWord); // Remove beginWord from the set to prevent cycles

Map<String, Integer> distanceMap = new HashMap<>(); // Map to track the shortest path distances for words

distanceMap.put(beginWord, 0); // Distance from beginWord to itself is 0

predecessorsMap = new HashMap<>(); // Initialize the predecessors map

Queue<String> queue = new ArrayDeque<>(); // Queue for BFS

queue.offer(beginWord);

boolean isEndWordFound = false; // Flag to check if endWord is found

int steps = 0; // Step counter for BFS

while (!queue.isEmpty() && !isEndWordFound) {

++steps;

for (int i = queue.size(); i > 0; --i) {

String currentWord = queue.poll();

char[] currentChars = currentWord.toCharArray();

for (int j = 0; j < currentChars.length; ++j) {

char originalChar = currentChars[j];

for (char c = 'a'; c <= 'z'; ++c) { // Try all possible one-letter mutations

currentChars[j] = c;

String newWord = new String(currentChars);

if (distanceMap.getOrDefault(newWord, 0) == steps) {

predecessorsMap.get(newWord).add(currentWord);

}

if (!wordSet.contains(newWord)) {

continue; // If the new word isn't in the set, skip it

}

// Update distance map and predecessors map for new words

predecessorsMap.computeIfAbsent(newWord, key -> new HashSet<>()).add(currentWord);

wordSet.remove(newWord); // Remove new word to prevent revisiting

queue.offer(newWord);

distanceMap.put(newWord, steps);

if (endWord.equals(newWord)) {

isEndWordFound = true; // Found the endWord; will finish after this level

}

}

currentChars[j] = originalChar; // Restore original character before next iteration

}

}

}

if (isEndWordFound) { // If the end word has been reached

Deque<String> path = new ArrayDeque<>(); // Path stack for reconstructing paths

path.add(endWord);

backtrackPath(path, beginWord, endWord); // Perform DFS to build all shortest paths

}

return allPaths; // Return the list of all shortest paths

}

private void backtrackPath(Deque<String> path, String beginWord, String currentWord) {

if (currentWord.equals(beginWord)) { // If the beginning of the path is reached, add it to allPaths

allPaths.add(new ArrayList<>(path));

return;

}

// Recursively go through all predecessors of the current word, adding them to the path

for (String predecessor : predecessorsMap.get(currentWord)) {

path.addFirst(predecessor); // Push the predecessor onto the path

backtrackPath(path, beginWord, predecessor); // Continue backtracking

path.removeFirst(); // Remove the predecessor to backtrack

}

}

}

**Time and Space Complexity**

**Time Complexity**

The algorithm performs a breadth-first search (BFS) to find the shortest path from beginWord to endWord, while also recording all predecessors of each word along the way. In the worst-case scenario, every word in the word list can be connected to every other word, leading to a complete graph-like structure. BFS in this case would have a time complexity of O(V + E), where **V is the number of vertices (words)** and E is the number of edges (possible transformations). For each word, there are **L possible single-letter changes**, and each change has **26 possible outcomes (the alphabet size)**, so the total number of edges would be O(26 \* L \* V). The time complexity can be further broken down as follows:

BFS to find the shortest paths and record predecessors: O(L \* 26 \* V), **where L is the length of each word, and V is the number of words in the word list.**

Depth-first search (DFS) to reconstruct paths: In the worst case, all paths from beginWord to endWord will need to be found. If there are **P such shortest paths and the depth of the search is D (corresponding to the number of transformations needed)**, this operation is O(P\*D).

Overall, the time complexity of this algorithm would be O(L \* 26 \* V + P \* D).

**Space Complexity**

The space complexity is primarily governed by the storage of words, the queue used in BFS, and the recursive stack for DFS.

Words storage: O(V), where V is the size of the wordList.

BFS Queue: O(V), the queue could at most contain all the words.

BFS prev map and dist map: These maps will store at most O(V) entries if every word is encountered.

DFS Paths: Up to O(P \* D) space may be needed to store all P paths, each with a depth of maximum D.

Thus, the total space complexity is O(V + P \* D).

**Refer to**

[L127.Word Ladder (Ref.L126)](note://64DD4420878F4745B1D326EA563912B6)