<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 105.

**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

private void dfs(String beginWord, String endWord, Map<String, List<String>> nodeNeighbors, Map<String, Integer> distance, List<String> individualSequence, List<List<String>> results) {

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);

}

}