November 23, 2022 ▪ Graph

Word Ladder-II (Optimised Approach) G-31

Given two distinct words **startWord** and **targetWord**, and a list denoting wordList of unique words of equal lengths. Find all shortest transformation sequence(s) from startWord to targetWord. You can return them in any order possible.

In this problem statement, we need to keep the following conditions in mind:

- A word can only consist of lowercase characters.
- Only one letter can be changed in each transformation.
- Each transformed word must exist in the wordList including the targetWord.
- startWord may or may not be part of the wordList.

Search

Search

Recent Posts

2023 – Striver's SDE Sheet Challenge

Top Array Interview Questions – Structured Path with Video Solutions

Longest Subarray with sum K | [Postives and Negatives]

- Return an empty list if there is no such transformation sequence.

Important Note:

Please watch the previous video of this series before moving on to this particular problem as this is just the optimized approach for the problem Word Ladder-II that is being discussed there.

The approach used for this problem is based on the concepts of **Competitive Programming,** so it is highly advised to read it just for the sake of improving your logic. This is not intended to be used in an interview, for that purpose you can easily explain the approach used in the G-30 article.

Examples:

```
Example 1:

Input:
startWord = "der", targetWord =
"dfs",
wordList =
{"des","der","dfr","dgt","dfs"}
Output:
[ [ “der”, “dfr”, “dfs” ], [
“der”, “des”, “dfs”] ]
Explanation:
The length of the smallest
transformation sequence here is
3.
Following are the only two
shortest ways to get to the
targetWord from the startWord :
```

Count Subarray sum Equals K

Binary Tree Representation in Java

Accolite Digital Amazon Arcesium arrays Bank of America Barclays BFS Binary Search Binary Search Tree Commvault CPP DE Shaw DFS DSA Self Paced google HackerEarth Hashing infosys inorder Interview Experience Java Juspay Kreeti Technologies Morgan Stanley Newfold Digital Oracle post order recursion Samsung SDE Core Sheet SDE Sheet Searching set-bits sorting Strivers A2ZDSA Course sub-array subarray Swiggy takeuforward TCS TCS CODEVITA TCS Ninja

```
"der" -> ( replace ‘r’ by ‘s’ )
-> "des" -> ( replace ‘e’ by ‘f’
) -> "dfs".
"der" -> ( replace ‘e’ by ‘f’ )
-> "dfr" -> ( replace ‘r’ by ‘s’
) -> "dfs".

Example 2:
Input:
startWord = "gedk", targetWord=
"geek"
wordList = {"geek", "gefk"}
Output:
[ [ “gedk”, “geek” ] ]
Explanation:
The length of the smallest
transformation sequence here is
2.
Following is the only shortest
way to get to the targetWord
from the startWord :
"gedk" -> ( replace ‘d’ by ‘e’ )
-> "geek".
```

TCS NQT

VMware XOR

Solution

***Disclaimer*: Don't jump directly to the solution, try it out yourself first.**

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Approach:

The only way to optimize this problem to a greater extent is to use a hack that is mainly used in competitive programming.

Initial configuration:

- **Vector:** Define a vector to store the final shortest sequences of transformation from the beginWord to the endWord.
- **Map:** A map of the form word -> level to store words along with the level on which they appear during the BFS traversal.
- **Hash Set:** Create a hash set to store the elements present in the word list to carry out the search and delete operations in O(1) time.
- **Queue:** Define a queue data structure to store the level-wise transformed words which also are present in the wordList.

The Algorithm is divided into majorly 2 steps :

Step 1: Finding the minimum number of steps to reach the endWord and storing the step number for every string in a data structure. So that we can backtrack at later stages.

- We follow a similar approach as that of the Word Ladder-I problem to find out the minimum number of steps in order to transform the beginWord to the endWord.
- First, insert the beginWord in a queue data structure and then start the BFS traversal.
- Now, we pop the first element out of the queue and carry out the BFS traversal

where, for each word popped out of the queue, we try to replace every character with ‘a’ – ‘z’, and we get a transformed word. We check if the transformed word is present in the wordList or not.

- If the word is present, we push it in the queue as well as push in the map and increase the count of level by 1 in the map. If the word is not present, we simply move on to replacing the original character with the next character.
- Remember, we also need to delete the word from the wordList if it matches with the transformed word to ensure that we do not reach the same point again in the transformation which would only increase our sequence length.
- Now, we pop the next element out of the queue ds and if at any point in time, the transformed word becomes the same as the targetWord, we stop the BFS.

Step 2: Backtrack in the map from end to beginning to get the answer sequences.

- We follow the DFS traversal here but in a reverse manner.
- Starting from only the targetWord in the sequence we replace the character by character from a-z in that word and check whether the transformed word is present in the map and at the previous level of the targetWord or not.
- If that is the case, we push the word into the sequence and then continue a similar traversal until we reach the beginWord.
- Following this technique eventually, we would get all the shortest possible sequences to reach from beginWord to targetWord but in reverse order. So the moment we encounter the beginWord in the traversal, we reverse the current sequence, insert it into the answer array and then re-reverse it to continue the DFS traversal as it is.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Intuition:

The main reason why the previous approach was giving TLE over strict time constraints was that we used to store the whole updated sequence in a queue data structure which consumed a lot of time as well as space. Now, as the first step instead of storing the sequences, we just store the words as we progress during the BFS traversal. This would give us an idea about the length of the shortest sequences possible. We also store the words along with the level on which they appear during the traversal in a map data structure so that we can already make a count of the possible number of paths to reach the targetWord.In the next step, we

backtrack from the end to begin to get the answer sequences. Through this, exploration of the paths would be minimal if we start from the back and unnecessary paths wouldn't be explored.

Code:

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution
{
    // Create a map of type word->le
    // on which level the word comes
    unordered_map<string, int> mpp;

    // A vector for storing the fina
    vector<vector<string>> ans;
    string b;

private:
    void dfs(string word, vector<str
    {
        // Function for implementing
        // in reverse order to find

        // Base condition :
        // If word equals beginWord,
        // simply reverse the sequen
        if (word == b)
        {
            reverse(seq.begin(), seq
            ans.push_back(seq);

            // reverse again so that
            reverse(seq.begin(), seq
            return;
        }
        int sz = word.size();
        int steps = mpp[word];
```

```
            // Replace each character of
            // and check whether the tra
            // and at the previous level
            for (int i = 0; i < sz; i++)
            {
                char original = word[i];
                for (char ch = 'a'; ch <
                {
                    word[i] = ch;
                    if (mpp.find(word) !
                    {
                        seq.push_back(wo
                        dfs(word, seq);
                        // pop the curre
                        // to traverse o
                        seq.pop_back();
                    }
                }
                word[i] = original;
            }
        }

public:
        vector<vector<string>> findLadde
        {
            // Push all values of wordLi
            // to make deletion from it
            unordered_set<string> st(wor

            // Perform BFS traversal and
            // as soon as they're found
            queue<string> q;
            b = beginWord;
            q.push({beginWord});

            // beginWord initialised wit
            mpp[beginWord] = 1;
            int sz = beginWord.size();
            st.erase(beginWord);
            while (!q.empty())
            {

                string word = q.front();
                int steps = mpp[word];
                q.pop();
```

```
                // Break out if the word
                if (word == endWord)
                    break;

                // Replace each characte
                // and check whether the
                // wordList or not, if y
                for (int i = 0; i < sz;
                {
                    char original = word

                    for (char ch = 'a';
                    {

                        word[i] = ch;
                        if (st.count(wor
                        {
                            q.push(word)
                            st.erase(wor

                            // push the 
                            // in the ma
                            mpp[word] =
                        }
                    }
                    word[i] = original;
                }
            }

            // If we reach the endWord, 
            // that is to perform revers
            if (mpp.find(endWord) != mpp
            {
                vector<string> seq;
                seq.push_back(endWord);
                dfs(endWord, seq);
            }
            return ans;
        }
    };

    // A comparator function to sort the
    bool comp(vector<string> a, vector<s
    {
        string x = "", y = "";
        for (string i : a)
```

```
        x += i;
    for (string i : b)
        y += i;

    return x < y;
}

int main()
{

    vector<string> wordList = {"des"
    string startWord = "der", target
    Solution obj;
    vector<vector<string>> ans = obj

    // If no transformation sequence
    if (ans.size() == 0)
        cout << -1 << endl;
    else
    {
        sort(ans.begin(), ans.end(),
        for (int i = 0; i < ans.size
        {
            for (int j = 0; j < ans[
            {
                cout << ans[i][j] <<
            }
            cout << endl;
        }
    }

    return 0;
}
```

Output:

der des dfs

der dfr dfs

Time Complexity and Space Complexity: It cannot be predicted for this particular algorithm because there can be multiple sequences of transformation from startWord to targetWord depending upon the example, so we cannot define a fixed range of time or space in which this program would run for all the test cases.

Java Code

```
import java.util.*;
import java.lang.*;
import java.io.*;

// A comparator function to sort the
class comp implements Comparator < L

    public int compare(List < String
        String x = "";
        String y = "";
        for (int i = 0; i < a.size()
            x += a.get(i);
        for (int i = 0; i < b.size()
            y += b.get(i);
        return x.compareTo(y);
    }
}

public class word_ladder {

    public static void main(String[]
        String startWord = "der", ta
        List < String > wordList = n
            {
                add("des");
                add("der");
                add("dfr");
                add("dgt");
                add("dfs");
            }
        };

        Solution obj = new Solution(
        List < List < String >> ans

        // If no transformation sequ
```

```
            if (ans.size() == 0)
                System.out.println(-1);
            else {

                Collections.sort(ans, ne
                for (int i = 0; i < ans.
                    for (int j = 0; j <
                        System.out.print
                    }
                    System.out.println()
                }
            }
        }
    }

    class Solution {
        String b;

        // Create a hashmap of type word
        // on which level the word comes

        HashMap < String, Integer > mpp;

        // A list for storing the final
        List < List < String >> ans;
        private void dfs(String word, Li

            // Function for implementing
            // in reverse order to find

            // Base condition :
            // If word equals beginWord,
            // simply reverse the sequen
            if (word.equals(b)) {

                // Since java works with
                // a duplicate and store
                List < String > dup = ne
                Collections.reverse(dup)
                ans.add(dup);
                return;
            }
            int steps = mpp.get(word);
            int sz = word.length();

            // Replace each character of
```

```
        // and check whether the tra
        // and at the previous level
        for (int i = 0; i < sz; i++)

            for (char ch = 'a'; ch <
                char replacedCharArr
                replacedCharArray[i]
                String replacedWord 
                if (mpp.containsKey(
                    mpp.get(replaced

                    seq.add(replaced
                    dfs(replacedWord

                    // pop the curre
                    // to traverse o
                    seq.remove(seq.s
                }
            }
        }
    }
    public List < List < String >> f
        List < String > wordList) {

        // Push all values of wordLi
        // to make deletion from it
        Set < String > st = new Hash
        int len = wordList.size();
        for (int i = 0; i < len; i++
            st.add(wordList.get(i));
        }

        // Perform BFS traversal and
        // as soon as they're found
        Queue < String > q = new Lin
        b = beginWord;
        q.add(beginWord);
        mpp = new HashMap < > ();

        // beginWord initialised wit
        mpp.put(beginWord, 1);
        int sizee = beginWord.length
        st.remove(beginWord);
        while (!q.isEmpty()) {
            String word = q.peek();
            int steps = mpp.get(word
```

```
                q.remove();

                // Break out if the word
                if (word.equals(endWord)

                // Replace each characte
                // and check whether the
                // wordList or not, if y
                for (int i = 0; i < size

                    for (char ch = 'a';
                        char replacedCha
                        replacedCharArra
                        String replacedW
                        if (st.contains(
                            q.add(replac
                            st.remove(re

                            // push the 
                            // in the ma
                            mpp.put(repl
                        }
                    }


                }
            }
            ans = new ArrayList < > ();

            // If we reach the endWord, 
            // that is to perform revers
            if (mpp.containsKey(endWord)
                List < String > seq = ne
                seq.add(endWord);
                dfs(endWord, seq);
            }
            return ans;
        }
    }
```

Output:

der des dfs

der dfr dfs

Time Complexity and Space Complexity: It cannot be predicted for this particular algorithm because there can be multiple sequences of transformation from startWord to targetWord depending upon the example, so we cannot define a fixed range of time or space in which this program would run for all the test cases.

> Special thanks to **Priyanshi Goel** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

« Previous Post
G-30 : Word Ladder-II

Next Post »
Dijkstra's Algorithm – Using Priority Queue : G-32

Load Comments