**Word Ladder**

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 *the****number of words****in the****shortest transformation sequence****from* beginWord *to* endWord*, or*0*if no such sequence exists.*

**Example 1:**

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

**Output:** 5

**Explanation:** One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

**Example 2:**

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

**Output:** 0

**Explanation:** The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

**Constraints:**

* 1 <= beginWord.length <= 10
* endWord.length == beginWord.length
* 1 <= wordList.length <= 5000
* wordList[i].length == beginWord.length
* beginWord, endWord, and wordList[i] consist of lowercase English letters.
* beginWord != endWord
* All the words in wordList are **unique**.

/\*\*

\* @param {string} beginWord

\* @param {string} endWord

\* @param {string[]} wordList

\* @return {number}

\*/

var ladderLength = function(beginWord, endWord, wordList) {

};

Solution Article

We are given a beginWord and an endWord. Let these two represent start node and end node of a graph. We have to reach from the start node to the end node using some intermediate nodes/words. The intermediate nodes are determined by the wordList given to us. The only condition for every step we take on this ladder of words is the current word should change by just one letter.

Diagram

Description automatically generated

We will essentially be working with an undirected and unweighted graph with words as nodes and edges between words which differ by just one letter. The problem boils down to finding the shortest path from a start node to a destination node, if there exists one. Hence it can be solved using Breadth First Search approach.

One of the most important step here is to figure out how to find adjacent nodes i.e. words which differ by one letter. To efficiently find the neighboring nodes for any given word we do some pre-processing on the words of the given wordList. The pre-processing involves replacing the letter of a word by a non-alphabet say, \*.

Shape

Description automatically generated with medium confidence

This pre-processing helps to form generic states to represent a single letter change.

For e.g. Dog ----> D\*g <---- Dig

Both Dog and Dig map to the same intermediate or generic state D\*g.

The preprocessing step helps us find out the generic one letter away nodes for any word of the word list and hence making it easier and quicker to get the adjacent nodes. Otherwise, for every word we will have to iterate over the entire word list and find words that differ by one letter. That would take a lot of time. This preprocessing step essentially builds the adjacency list first before beginning the breadth first search algorithm.

For eg. While doing BFS if we have to find the adjacent nodes for Dug we can first find all the generic states for Dug.

1. Dug => \*ug
2. Dug => D\*g
3. Dug => Du\*

The second transformation D\*g could then be mapped to Dog or Dig, since all of them share the same generic state. Having a common generic transformation means two words are connected and differ by one letter.

Approach 1: Breadth First Search

**Intuition**

Start from beginWord and search the endWord using BFS.

**Algorithm**

1. Do the pre-processing on the given wordList and find all the possible generic/intermediate states. Save these intermediate states in a dictionary with key as the intermediate word and value as the list of words which have the same intermediate word.
2. Push a tuple containing the beginWord and 1 in a queue. The 1 represents the level number of a node. We have to return the level of the endNode as that would represent the shortest sequence/distance from the beginWord.
3. To prevent cycles, use a visited dictionary.
4. While the queue has elements, get the front element of the queue. Let's call this word as current\_word.
5. Find all the generic transformations of the current\_word and find out if any of these transformations is also a transformation of other words in the word list. This is achieved by checking the all\_combo\_dict.
6. The list of words we get from all\_combo\_dict are all the words which have a common intermediate state with the current\_word. These new set of words will be the adjacent nodes/words to current\_word and hence added to the queue.
7. Hence, for each word in this list of intermediate words, append (word, level + 1) into the queue where level is the level for the current\_word.
8. Eventually if you reach the desired word, its level would represent the shortest transformation sequence length.

Termination condition for standard BFS is finding the end word.

class Solution {

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

// Since all words are of same length.

int L = beginWord.length();

// Dictionary to hold combination of words that can be formed,

// from any given word. By changing one letter at a time.

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

wordList.forEach( word -> {

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

// Key is the generic word

// Value is a list of words which have the same intermediate generic word.

String newWord = word.substring(0, i) + '\*' + word.substring(i + 1, L);

List<String> transformations = allComboDict.getOrDefault(newWord, new ArrayList<>());

transformations.add(word);

allComboDict.put(newWord, transformations);

}

});

// Queue for BFS

Queue<Pair<String, Integer>> Q = new LinkedList<>();

Q.add(new Pair(beginWord, 1));

// Visited to make sure we don't repeat processing same word.

Map<String, Boolean> visited = new HashMap<>();

visited.put(beginWord, true);

while (!Q.isEmpty()) {

Pair<String, Integer> node = Q.remove();

String word = node.getKey();

int level = node.getValue();

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

// Intermediate words for current word

String newWord = word.substring(0, i) + '\*' + word.substring(i + 1, L);

// Next states are all the words which share the same intermediate state.

for (String adjacentWord : allComboDict.getOrDefault(newWord, new ArrayList<>())) {

// If at any point if we find what we are looking for

// i.e. the end word - we can return with the answer.

if (adjacentWord.equals(endWord)) {

return level + 1;

}

// Otherwise, add it to the BFS Queue. Also mark it visited

if (!visited.containsKey(adjacentWord)) {

visited.put(adjacentWord, true);

Q.add(new Pair(adjacentWord, level + 1));

}

}

}

}

return 0;

}

}

**Complexity Analysis**

* Time Complexity: O({M}^2 \times N)*O*(*M*2×*N*), where M*M* is the length of each word and N*N* is the total number of words in the input word list.
  + For each word in the word list, we iterate over its length to find all the intermediate words corresponding to it. Since the length of each word is M*M* and we have N*N* words, the total number of iterations the algorithm takes to create all\_combo\_dict is M \times N*M*×*N*. Additionally, forming each of the intermediate word takes O(M)*O*(*M*) time because of the substring operation used to create the new string. This adds up to a complexity of O({M}^2 \times N)*O*(*M*2×*N*).
  + Breadth first search in the worst case might go to each of the N*N* words. For each word, we need to examine M*M* possible intermediate words/combinations. Notice, we have used the substring operation to find each of the combination. Thus, M*M* combinations take O({M} ^ 2)*O*(*M*2) time. As a result, the time complexity of BFS traversal would also be O({M}^2 \times N)*O*(*M*2×*N*).

Combining the above steps, the overall time complexity of this approach is O({M}^2 \times N)*O*(*M*2×*N*).

* Space Complexity: O({M}^2 \times N)*O*(*M*2×*N*).
  + Each word in the word list would have M*M* intermediate combinations. To create the all\_combo\_dict dictionary we save an intermediate word as the key and its corresponding original words as the value. Note, for each of M*M* intermediate words we save the original word of length M*M*. This simply means, for every word we would need a space of {M}^2*M*2 to save all the transformations corresponding to it. Thus, all\_combo\_dict would need a total space of O({M}^2 \times N)*O*(*M*2×*N*).
  + Visited dictionary would need a space of O(M \times N)*O*(*M*×*N*) as each word is of length M*M*.
  + Queue for BFS in worst case would need a space for all O(N)*O*(*N*) words and this would also result in a space complexity of O(M \times N)*O*(*M*×*N*).

Combining the above steps, the overall space complexity is O({M}^2 \times N)*O*(*M*2×*N*) + O(M \* N)*O*(*M*∗*N*) + O(M \* N)*O*(*M*∗*N*) = O({M}^2 \times N)*O*(*M*2×*N*) space.

**Optimization:** We can definitely reduce the space complexity of this algorithm by storing the indices corresponding to each word instead of storing the word itself.

Approach 2: Bidirectional Breadth First Search

**Intuition**

The graph formed from the nodes in the dictionary might be too big. The search space considered by the breadth first search algorithm depends upon the branching factor of the nodes at each level. If the branching factor remains the same for all the nodes, the search space increases exponentially along with the number of levels. Consider a simple example of a binary tree. With each passing level in a complete binary tree, the number of nodes increase in powers of 2.

We can considerably cut down the search space of the standard breadth first search algorithm if we launch two simultaneous BFS. One from the beginWord and one from the endWord. We progress one node at a time from both sides and at any point in time if we find a common node in both the searches, we stop the search. This is known as bidirectional BFS and it considerably cuts down on the search space and hence reduces the time and space complexity.

Diagram

Description automatically generated

**Algorithm**

1. The algorithm is very similar to the standard BFS based approach we saw earlier.
2. The only difference is we now do BFS starting two nodes instead of one. This also changes the termination condition of our search.
3. We now have two visited dictionaries to keep track of nodes visited from the search starting at the respective ends.
4. If we ever find a node/word which is in the visited dictionary of the parallel search we terminate our search, since we have found the meet point of this bidirectional search. It's more like meeting in the middle instead of going all the way through.

Termination condition for bidirectional search is finding a word which is already been seen by the parallel search.

1. The shortest transformation sequence is the sum of levels of the meet point node from both the ends. Thus, for every visited node we save its level as value in the visited dictionary.

Diagram, schematic

Description automatically generated

class Solution {

private int L;

private Map<String, List<String>> allComboDict;

Solution() {

this.L = 0;

// Dictionary to hold combination of words that can be formed,

// from any given word. By changing one letter at a time.

this.allComboDict = new HashMap<>();

}

private int visitWordNode(

Queue<Pair<String, Integer>> Q,

Map<String, Integer> visited,

Map<String, Integer> othersVisited) {

for (int j = Q.size(); j > 0; j--) {

Pair<String, Integer> node = Q.remove();

String word = node.getKey();

int level = node.getValue();

for (int i = 0; i < this.L; i++) {

// Intermediate words for current word

String newWord = word.substring(0, i) + '\*' + word.substring(i + 1, L);

// Next states are all the words which share the same intermediate state.

for (String adjacentWord : this.allComboDict.getOrDefault(newWord, new ArrayList<>())) {

// If at any point if we find what we are looking for

// i.e. the end word - we can return with the answer.

if (othersVisited.containsKey(adjacentWord)) {

return level + othersVisited.get(adjacentWord);

}

if (!visited.containsKey(adjacentWord)) {

// Save the level as the value of the dictionary, to save number of hops.

visited.put(adjacentWord, level + 1);

Q.add(new Pair(adjacentWord, level + 1));

}

}

}

}

return -1;

}

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

if (!wordList.contains(endWord)) {

return 0;

}

// Since all words are of same length.

this.L = beginWord.length();

wordList.forEach( word -> {

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

// Key is the generic word

// Value is a list of words which have the same intermediate generic word.

String newWord = word.substring(0, i) + '\*' + word.substring(i + 1, L);

List<String> transformations =

this.allComboDict.getOrDefault(newWord, new ArrayList<>());

transformations.add(word);

this.allComboDict.put(newWord, transformations);

}

});

// Queues for birdirectional BFS

// BFS starting from beginWord

Queue<Pair<String, Integer>> Q\_begin = new LinkedList<>();

// BFS starting from endWord

Queue<Pair<String, Integer>> Q\_end = new LinkedList<>();

Q\_begin.add(new Pair(beginWord, 1));

Q\_end.add(new Pair(endWord, 1));

// Visited to make sure we don't repeat processing same word.

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

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

visitedBegin.put(beginWord, 1);

visitedEnd.put(endWord, 1);

int ans = -1;

while (!Q\_begin.isEmpty() && !Q\_end.isEmpty()) {

// Progress forward one step from the shorter queue

if (Q\_begin.size() <= Q\_end.size()) {

ans = visitWordNode(Q\_begin, visitedBegin, visitedEnd);

} else {

ans = visitWordNode(Q\_end, visitedEnd, visitedBegin);

}

if (ans > -1) {

return ans;

}

}

return 0;

}

}

**Complexity Analysis**

* Time Complexity: O({M}^2 \times N)*O*(*M*2×*N*), where M*M* is the length of words and N*N* is the total number of words in the input word list. Similar to one directional, bidirectional also takes O({M}^2 \times N)*O*(*M*2×*N*) time for finding out all the transformations. But the search time reduces to half, since the two parallel searches meet somewhere in the middle.
* Space Complexity: O({M}^2 \times N)*O*(*M*2×*N*), to store all M*M* transformations for each of the N*N* words in the all\_combo\_dict dictionary, same as one directional. But bidirectional reduces the search space. It narrows down because of meeting in the middle.