Word Ladder

Previous Next

127. Word Ladder

Average Rating: 4.73 (89 votes)

Feb. 17, 2019 | 74.5K views

Given two words (beginWord and endWord), and a dictionary's word list, find the length of shortest transformation sequence from beginWord to endWord, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume beginWord and endWord are non-empty and are not the same.

Example 1:

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

Output: 5

Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
return its length 5.
```

Example 2:

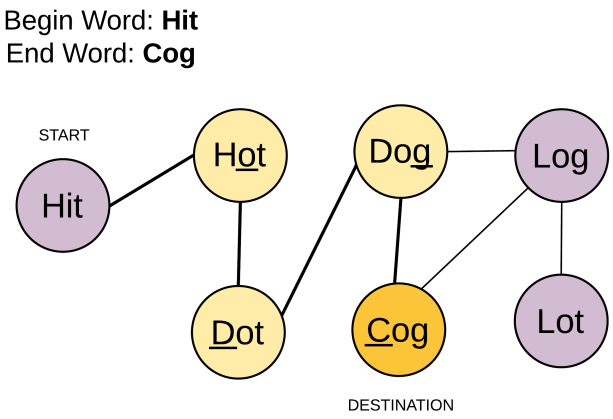
```
Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]
```

```
Output: 0

Explanation: The endWord "cog" is not in wordList, therefore no possible transformation.
```

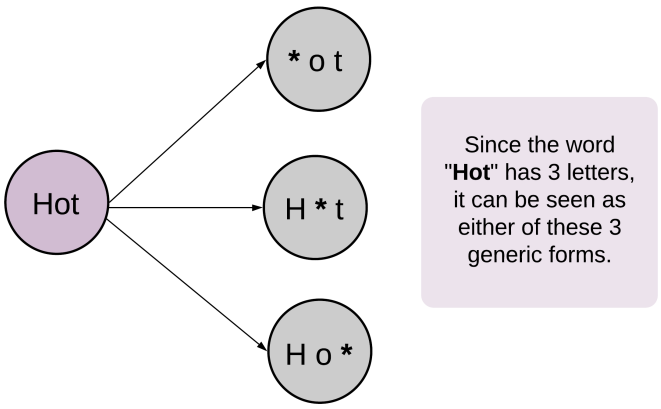
Solution

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`.



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, `*`.



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.

```
25
26  // Queue for BFS
27  Queue<Pair<String, Integer>> Q = new LinkedList<Pair<String, Integer>>();
28  Q.add(new Pair(beginWord, 1));
29
30  // Visited to make sure we don't repeat processing same word.
```

$O(M \times N)$ M N M N N $O(M \times N)$ M N N