127. Word Ladder

Feb. 17, 2019 | 209.1K views
Average Rating: 4.63 (177 votes)

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:

- 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" -> "cc
return its length 5.
```

Example 2:

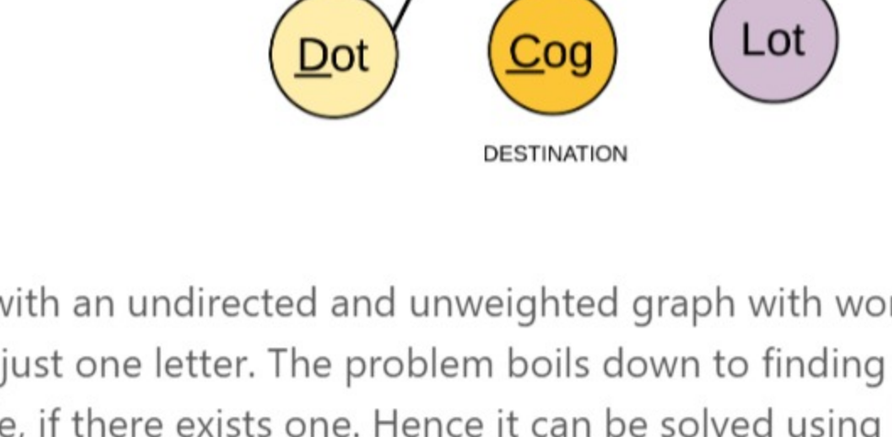
```
Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

Output: 0

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

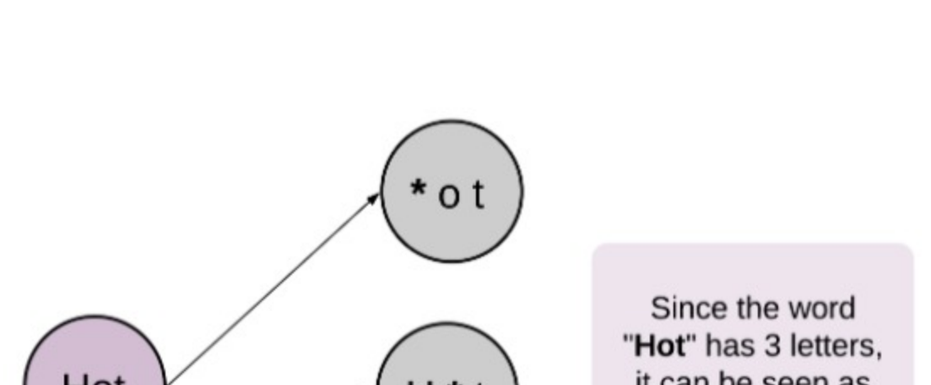
Solution

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.

Java | Python | Copy

```python
from collections import defaultdict
class Solution(object):
    def ladderLength(self, beginWord, endWord, wordList):
        """
        :type beginWord: str
        :type endWord: str
        :type wordList: List[str]
        :rtype: int
        """

        if endWord not in wordList or not endWord or not beginWord or not wordList:
            return 0

        # Since all words are of same length.
        L = len(beginWord)

        # Dictionary to hold combination of words that can be formed,
        # from any given word. By changing one letter at a time.
        all_combo_dict = defaultdict(list)
        for word in wordList:
            for i in range(L):
                # Key is the generic word
                # Value is a list of words which have the same intermediate generic word.
                all_combo_dict[word[:i] + "*" + word[i+1:]].append(word)


        # Queue for BFS
```

Complexity Analysis

- Time Complexity: $O(M^2 \times N)$, where M is the length of each word and 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 and we have N words, the total number of iterations the algorithm takes to create `all_combo_dict` is $M \times N$. Additionally, forming each of the intermediate word takes $O(M)$ time because of the substring operation used to create the new string. This adds up to a complexity of $O(M^2 \times N)$.
- Breadth first search in the worst case might go to each of the N words. For each word, we need to examine M possible intermediate words/combinations. Notice, we have used the substring operation to find each of the combination. Thus, M combinations take $O(M^2)$ time. As a result, the time complexity of BFS traversal would also be $O(M^2 \times N)$.

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

- Space Complexity: $O(M^2 \times N)$.
- Each word in the word list would have 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 intermediate words we save the original word of length M. This simply means, for every word we would need a space of M^2 to save all the transformations corresponding to it. Thus, `all_combo_dict` would need a total space of $O(M^2 \times N)$.
- `Visited` dictionary would need a space of $O(M \times N)$ as each word is of length M.
- Queue for BFS in worst case would need a space for all $O(N)$ words and this would also result in a space complexity of $O(M \times N)$.

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

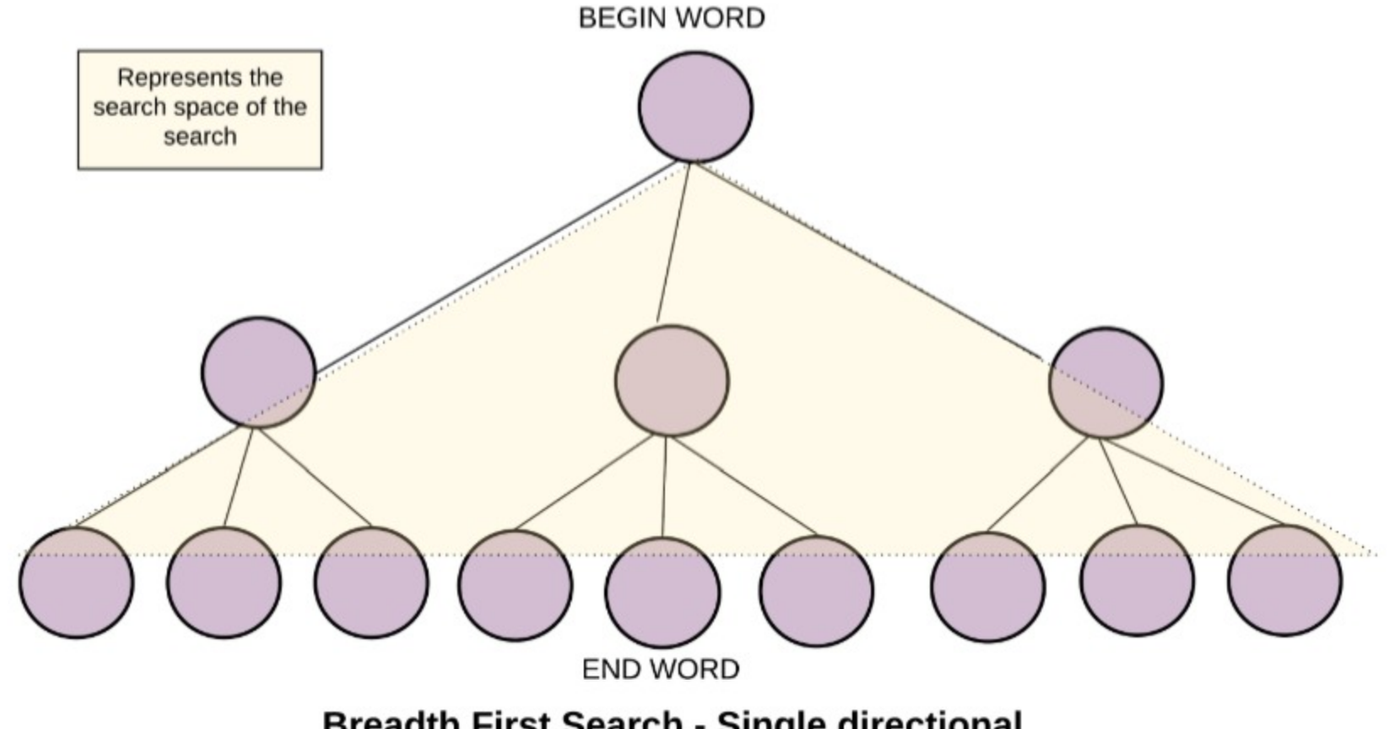
Optimization: We can definitely reduce the space complexity of this algorithm by storing the indices corresponding to each word instead of storing the word itself.

Approach 2: Bidirectional Breadth First Search

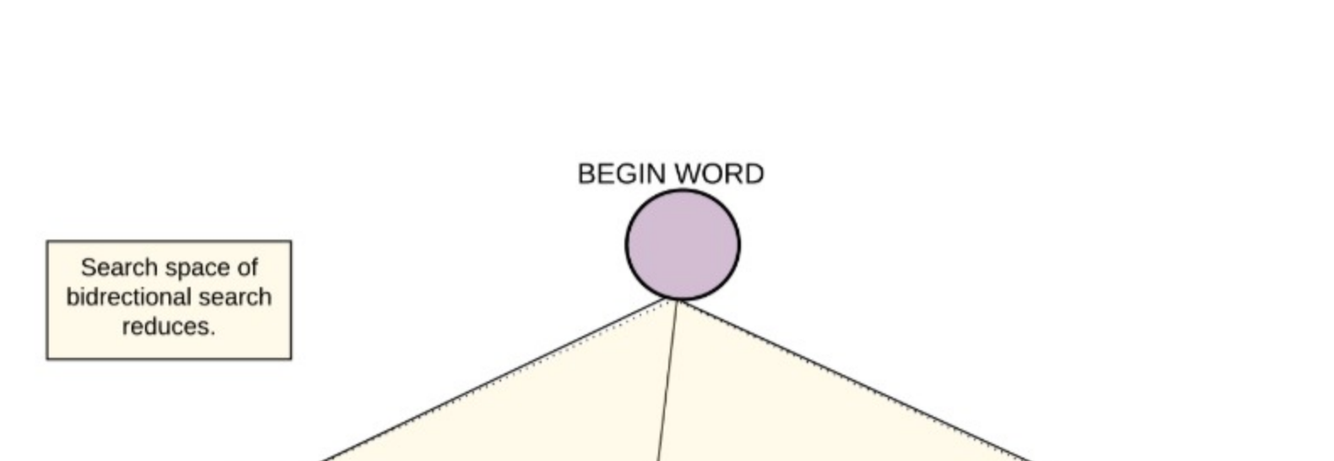
Intuition

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.



Breadth First Search - Single directional



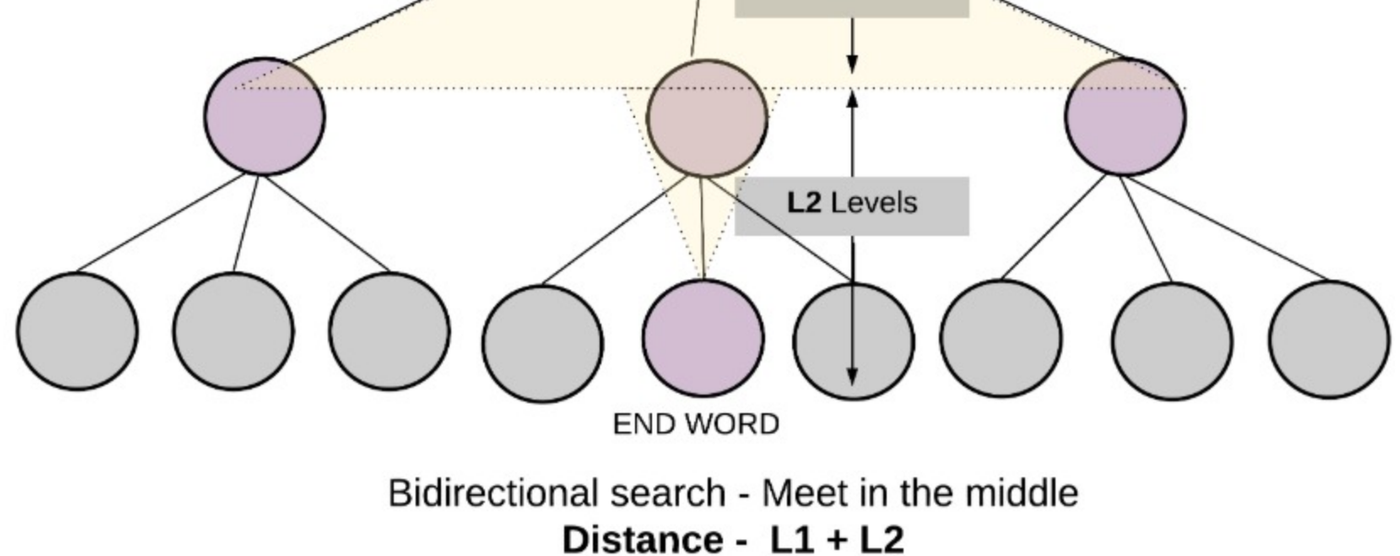
Breadth First Search - Bidirectional

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.

5. 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.



Bidirectional search - Meet in the middle
Distance - L1 + L2

Java | Python | Copy

```python
from collections import defaultdict
class Solution(object):
    def __init__(self):
        self.length = 0
        # Dictionary to hold combination of words that can be formed,
        # from any given word. By changing one letter at a time.
        self.all_combo_dict = defaultdict(list)

    def visitWordNode(self, queue, visited, others_visited):
        current_word, level = queue.popleft()
        for i in range(self.length):
            # Intermediate words for current word
            intermediate_word = current_word[:i] + "*" + current_word[i+1:]

            # Next states are all the words which share the same intermediate state.
            for word in self.all_combo_dict[intermediate_word]:
                # If the intermediate state/word has already been visited from the
                # other parallel traversal this means we have found the answer.
                if word in others_visited:
                    return level + others_visited[word]
                if word not in visited:
                    # Save the level as the value of the dictionary, to save number of hops.
                    visited[word] = level + 1
                    queue.append((word, level + 1))
        return None

    def ladderLength(self, beginWord, endWord, wordList):
```

Complexity Analysis

- Time Complexity: $O(M^2 \times N)$, where M is the length of words and N is the total number of words in the input word list. Similar to one directional, bidirectional also takes $O(M^2 \times N)$ 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: $O(M^2 \times N)$, to store all M transformations for each of the 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.

Rate this article: ★★★★★

Previous Next

Comments: 72 Sort By

Type comment here... (Markdown is supported)

Preview Post

littletonconnor ★346 July 17, 2019 7:56 PM

This does not seem like a "medium" problem to me, more like hard. :(

271 Share Reply

SHOW 5 REPLIES

CoderJoe ★687 March 1, 2019 8:11 AM

Try my concise and intuitive solution here:

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

Read More

69 Share Reply

SHOW 12 REPLIES

enthu ★24 April 22, 2019 5:02 AM

For 1st approach I don't quite get the time complexity M X N: how? Won't it be something like O(N X M X N X M)? As you have one main loop of Q which can go N times -> next you have inner loop of transforming each letter of each word containing M letters so max it can go for M times -> next you iterate through all_combo_dict which can go close to N times -> next you compare M letters of the word with endWord each time while iterating within allcombodict . So won't the complexity be O(M^2

Read More

23 Share Reply

SHOW 8 REPLIES

sellerdoor ★22 March 19, 2019 9:11 PM

The first Python solution should be updated to use `collections.deque` or `queue.Queue` for the BFS queue. Popping from the front of a list is a linear time operation in Python.

```
adj = collections.defaultdict(list)
```

Read More

22 Share Reply

SHOW 3 REPLIES

RodneyShag ★1058 August 9, 2019 11:47 AM

In Solution #1, it's $O(M^2N)$ space complexity just to store words in `allComboDict`. For example, for the word "food", where M=4, we would store `*ood, f*od, fo*d, foo*`, which is 16 characters = 4^2 = M^2. So it definitely can't be O(MN) space complexity.

The time complexity is also $O(M^2N)$ for this part of the code as well.

Read More

15 Share Reply

SHOW 3 REPLIES

499655087 ★21 April 28, 2019 12:18 AM

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

why it expect result is 5 for this?

10 Share Reply

SHOW 11 REPLIES

th015 ★238 April 24, 2019 12:44 PM

pop(0) is O(n), very slow.

14 Share Reply

SHOW 2 REPLIES

21S ★10 April 20, 2019 6:31 AM

shouldn't dot and lot also be connected in this graph ?

8 Share Reply

SHOW 1 REPLY

majinRamesh ★9 January 31, 2020 8:51 AM

beginWord = "a"
endWord = "c"
wordlist = ["a", "b", "c"]
Why is the expected output 2?

7 Share Reply

SHOW 1 REPLY

ping_pong ★811 July 12, 2019 8:24 AM

I think the time complexity in both the solution is $O(M N^2)$, correct me if I am wrong.

12 Share Reply

SHOW 2 REPLIES