Approach 1: Breadth-First Search (BFS) + Backtracking

Intuition

The problem can be correlated with the graph data structure. We can represent the words as the vertices and an edge can be used to connect two words which differ by a single letter.

Before diving further let's see how we can find all the direct connections of a particular word. To find the adjacent words for a particular word, one approach is to traverse all of the other words and add an edge for those that differ by a single letter. This approach requires $O(N \cdot K)$ time where N is the number of words given and K is the maximum length of a word. The observation behind the optimal approach is that the words only consist of lowercase English letters. Hence we can change each character of the word to all other English lowercase characters and check whether or not that word exists in the `wordList` (this particular check operation takes $O(1)$ in C++ while in Java it will take $O(K)$ due to the immutable nature of Strings).This way the number of operations will be $(25 \cdot K * K + 1)$, hence the time complexity will be $O(K^2)$.

Thus we can find all the words that are directly connected. Now, the task is to find all of the shortest paths from `beginWord` to `endWord`.

The naive way to do this is to use backtracking. We will start from `beginWord`, then traverse all the adjacent words until we reach the `endWord`. When we reach the `endWord`, we can compare the path length and find all the paths that have the minimum path length. This method however is extremely inefficient because the number of paths between two vertices can be enormous.

Let's try to optimize our approach. Somehow, we need to reduce the number of traversed paths. Let's say the number of shortest paths that exist between `beginWord` and `endWord` is `x` and the number of paths that we must traverse to find these

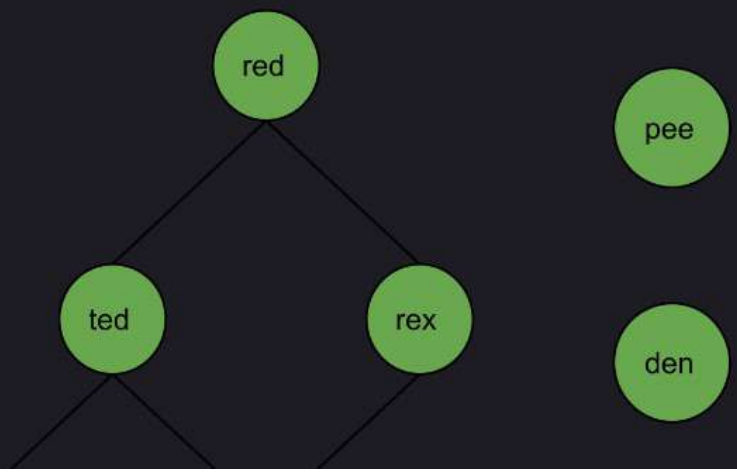
shortest paths is `y`. The closer the value `y` gets to the value `x`, the more efficient our approach will be.

The diagram below shows the graph that represents the connectivity among words. As shown in the diagram we want to go from `red` to `tax`. While backtracking on this graph, we will also cover the edges upwards that is from the `tad` to `ted` similarly from `tex` we will traverse to `ted` as well as `rex`. The key observation here is that going back in the upward direction will never lead us to the shortest path. We should always traverse the edges in the direction of `beginWord` to `endWord`.

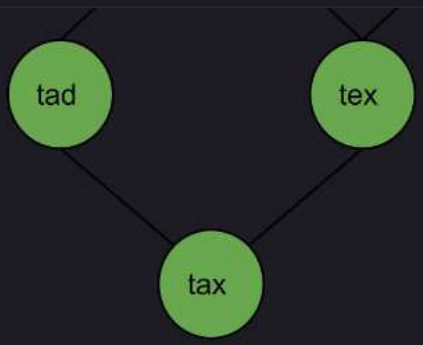
beginWord = red

endWord = tax

wordList = {ted, tex, red, tax, tad, den, rex, pee}



Quick Navigation



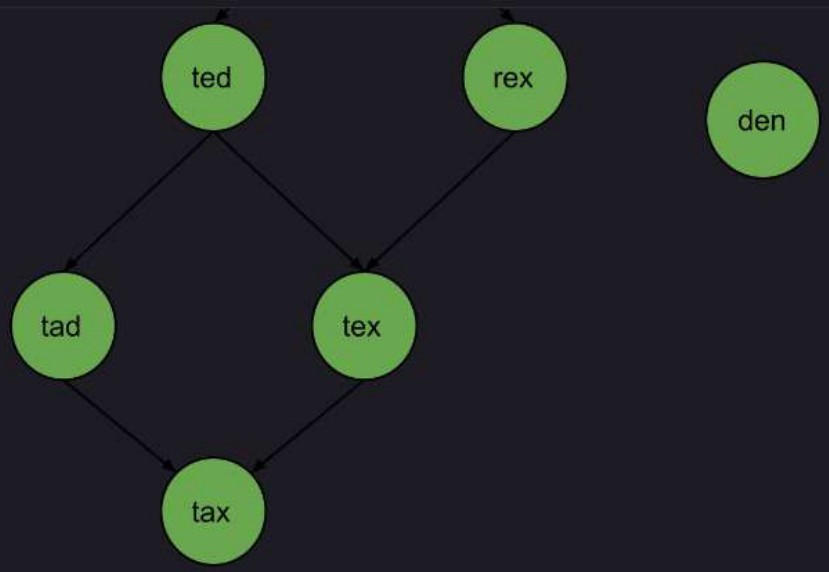
To ensure that we never traverse up the ladder, let's use directed edges to connect the words. The edges in the graph below are all directed towards `endWord`. Also, notice that graphs produced by BFS do not contain cycles. Thus, the graph will be a Directed Acyclic Graph (DAG).

beginWord = red

endWord = tax

wordList = {ted, tex, red, tax, tad, den, rex, pee}

Quick Navigation



Now for the easy part, think of the previous graph as a bunch of layers and observe that once we reach a particular layer we don't want the future words to have the connection back to this layer. We will build our DAG using BFS. We will then add all the directed edges from the words present in the current layer and once all words in this layer have been traversed, we will remove them from the `wordList`. This way we will avoid adding any edges that point towards `beginWord`.

After constructing the graph, we can use our same backtracking approach to find the shortest paths between `beginWord` and `endWord`. Also, note that in the graph all paths between `beginWord` and `endWord`, obtained through BFS, will be the shortest possible. This is because all the edges in the graph will be directed in the direction of `beginWord` to `endWord`. Furthermore, there will not be any edge between the words that are on the same level. Therefore, iterating over any edge will bring us one step closer to the `endWord`, thus there is no need to compare the length of the path each time we reach the `endWord`.

Algorithm

Algorithm

1. Store the words present in `wordList` in an unordered set so that the words can be efficiently removed during the breadth-first search.
2. Perform the BFS, and add the edges to the adjacency list `adjList`. Also once a level is finished remove the `visited` words from the `wordList`.
3. Start from `beginWord` and while keep tracking of the current path as `currPath` traverse all the possible paths, whenever the path leads to the `endWord` store the path in `shortestPaths`.

Implementation

> **NOTE:**
>
> In the following implementation, for convineince, instead for go from `beginWord` to `endWord`, we go from `endWord` to `beginWord`.

C++ | Java

Copy

```cpp
class Solution {
public:
    unordered_map<string, vector<string>> adjList;
    vector<string> currPath;
    vector<vector<string>> shortestPaths;

    vector<string> findNeighbors(string &word, unordered_set<string> &wordList) {
```

Quick Navigation

Complexity Analysis

- Time complexity: $O(NK^2 + \alpha)$.

  Here N is the number of words in `wordList`, K is the maximum length of a word, α is the number of possible paths from `beginWord` to `endWord` in the directed graph we have.

  Copying the `wordList` into the set will take $O(N)$.

  In BFS, every word will be traversed and for each word, we will find the neighbors using the function `findNeighbors` which has a time complexity of $O(K^2)$. Therefore the total complexity for all the `N` words will be $O(NK^2)$. Also, each word will be enqueued and will be removed from the set hence it will take $O(N)$. The total time complexity of BFS will therefore be equal to $O(NK^2)$.

  While backtracking, we will essentially be finding all the paths from `beginWord` to `endWord`. Thus the time complexity will be equal to $O(\alpha)$.

  We can estimate the upper bound for α by assuming that every layer except the first and the last layer in the DAG has x number of words and is fully connected to the next layer. Let h represent the height of the DAG, so the total number of paths will be x^h (because we can choose any one word out of x words in each layer and each choice will be part of a valid shortest path that leads to the `endWord`). Here, h equals $(N-2)/x$. This would result in $x^{(N-2)/x}$ total paths, which is maximized when $x = 2.718$, which we will round to 3 because x must be an integer. Thus the upper bound for α is $3^{(N/3)}$, however, this is a very loose bound because the nature of this problem precludes the possibility of a DAG where every layer is fully connected to the next layer.

  The total time complexity is therefore equal to $O(NK^2 + \alpha)$.

- Space complexity: $O(NK)$.

  Here N is the Number of words in `wordList`, K is the Maximum length of a word.

  Storing the words in a set will take $O(NK)$ space.

  To build the adjacency list $O(N)$ space is required as the BFS will produce a directed graph and hence there will be at max $(N-1)$ edges.

  In backtracking, stack space will be consumed which will be equal to the maximum number of active functions in the stack which is equal to the N as the path can have all the words in the `wordList`. Hence space required is $O(N)$.

  The total space complexity is therefore equal to $O(NK)$.