Shortest Word Edit Path

Given two words `source` and `target`, and a list of words `words`, find the length of the shortest series of edits that transforms `source` to `target`.

Each edit must change exactly one letter at a time, and each intermediate word (and the final `target` word) must exist in `words`.

If the task is impossible, return `-1`.

Examples:

```
source = "bit", target = "dog"
words = ["but", "put", "big", "pot", "pog", "dog", "lot"]

output: 5
explanation: bit -> but -> put -> pot -> pog -> dog has 5 transitions.
```

```
source = "no", target = "go"
words = ["to"]

output: -1
```

Constraints:

[time limit] 5000ms

[input] string `source`

1 ≤ source.length ≤ 20

[input] string `target`

1 ≤ target.length ≤ 20

Hints & Tips

What algorithms might be useful for finding a shortest distance?

We should use a breadth-first search.

Typically in a breadth first search, there is some graph, which for each node has some number of neighbors. How might you find which words in the word list are neighbors?

There are two approaches to finding neighbors of a word.

> One approach is for each letter in the original word, change it and check if that word is in the list of words.
> Another approach is for each word in the word list, to check if exactly one letter is different between it and the original word.

Solution

Solution: Breadth First Search

Finding a shortest path is typically done with a breadth first search. Here, we have some underlying graph of words, and two words are connected (neighbors) if they differ by exactly one letter (and have the same length).

The breadth first search explores all nodes distance 0 from the source, then all nodes distance 1, then all nodes distance 2, and so on. This ensure that if we find the target word, we found it at the least possible distance and thus the answer is correct.

One difficulty in this question is, given a `word`, what are all neighboring words? (Words that differ by exactly one letter.)

There are two strategies to enumerating these neighbors:

lowercase letter c , clone word into word2 , replace word2[i] with c , and check whether the resulting word2 is in words .

The decision to use one strategy or another depends on the input parameters. Here, we showcase the second strategy, with the first in the comments.

```
function shortestWordEditPath(source, target, words):
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    wordset = new HashSet(words)
    queue = Queue()
    queue.add((source, 0))
    seen = new HashSet([source])

    while queue:
        word, depth = queue.popfront()
        if word == target: return depth
        for i from 0 to word.length:
            # First Strategy:
            # for word2 in words:
            #     if word2.length == word.length:
            #         diff = 0
            #         for j from 0 to word.length:
            #             if word[j] != word2[j]:
            #                 diff += 1
            #                 if diff == 2: break
            #         if diff == 1 and word2 not in seen:
            #             queue.append((word2, depth+1))
            #             seen.add(word2)

            # Second Strategy:
            for c in alphabet:
                word2 = word.clone()
                word2[i] = c
                if word2 in wordset and word2 not in seen:
                    queue.append((word2, depth+1))
                    seen.add(word2)
    return -1
```

Found a mistake? Think we can do better? Let us know

Space Complexity: `O(NK)` , the space to store the word list.

[Pra]ctice [m]akes [P]erfect!

Go Back