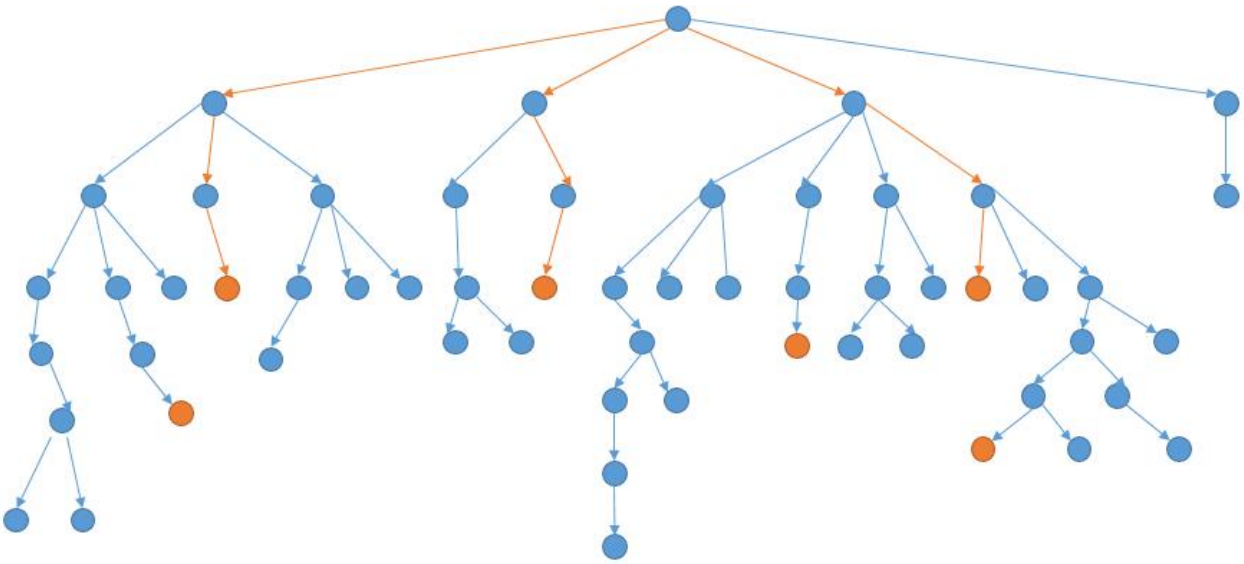
详细通俗的思路分析，多解法

给定两个单词，一个作为开始，一个作为结束，还有一个单词列表。然后依次选择单词，只有当前单词到下一个单词只有一个字母不同才能被选择，然后新的单词再作为当前单词，直到选到结束的单词。输出这个的最短路径，如果有多组，则都输出。

思路分析

如果我们从开始的单词，把与之能够转换的单词连起来，它就会长成下边的样子。



橙色表示结束单词，上图橙色的路线就是我们要找的最短路径。所以我们要做的其实就是遍历上边的树，然后判断当前节点是不是结束单词，找到结束单词后，还要判断当前是不是最短的路径。说到遍历当然就是两种思路了，`DFS` 或者 `BFS`。

解法一 DFS

利用回溯的思想，做一个 DFS。

首先要解决的问题是怎么找到节点的所有孩子节点。这里有两种方案。

第一种，遍历 `wordList` 来判断每个单词和当前单词是否只有一个字母不同。

```
for (int i = 0; i < wordList.size(); i++) {
    String curWord = wordList.get(i);

    if (oneChanged(beginWord, curWord)) {

    }
}

private boolean oneChanged(String beginWord, String curWord) {
    int count = 0;
    for (int i = 0; i < beginWord.length(); i++) {
        if (beginWord.charAt(i) != curWord.charAt(i)) {
            count++;
        }
        if (count == 2) {
            return false;
        }
    }
    return count == 1;
}
```

这种的时间复杂度的话，如果 `wordList` 长度为 `m` ，每个单词的长度为 `n` 。那么就是 `O(mn)` 。
第二种，将要找的节点单词的每个位置换一个字符，然后看更改后的单词在不在 `wordList` 中。

```
private List<String> getNext(String cur, Set<String> dict) {
    List<String> res = new ArrayList<>();
    char[] chars = cur.toCharArray();

    for (int i = 0; i < chars.length; i++) {
        char old = chars[i];

        for (char c = 'a'; c <= 'z'; c++) {
            if (c == old) {
                continue;
            }
            chars[i] = c;
            String next = new String(chars);

            if (dict.contains(next)) {
                res.add(next);
            }
        }
        chars[i] = old;
    }
    return res;
}
```

这种的话，由于用到了 `HashSet` ，所以 `contains` 函数就是 $O(1)O(1)O(1)$。所以整个计算量就是 `26n` ，所以是 $O(n)O(n)O(n)$。

还要解决的一个问题是，因为我们要找的是最短的路径。但是事先我们并不知道最短的路径是多少，我们需要一个全局变量来保存当前找到的路径的长度。如果找到的新的路径的长度比之前的路径短，就把之前的结果清空，重新找，如果是最小的长度，就加入到结果中。

看下一递归出口。

```
if (beginWord.equals(endWord)) {

    if (min > temp.size()) {
        ans.clear();
        min = temp.size();
        ans.add(new ArrayList<String>(temp));

    } else if (min == temp.size()) {
        ans.add(new ArrayList<String>(temp));
    }
    return;
}

if (temp.size() >= min) {
    return;
}
```

得到下一个节点刚才讲了两种思路，我们先采用第一种解法，看一下效果。

```
public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
    List<List<String>> ans = new ArrayList<>();
    ArrayList<String> temp = new ArrayList<String>();

    temp.add(beginWord);
    findLaddersHelper(beginWord, endWord, wordList, temp, ans);
    return ans;
}

int min = Integer.MAX_VALUE;

private void findLaddersHelper(String beginWord, String endWord, List<String> wordList,
                               ArrayList<String> temp, List<List<String>> ans) {
    if (beginWord.equals(endWord)) {
        if (min > temp.size()) {
            ans.clear();
            min = temp.size();
            ans.add(new ArrayList<String>(temp));
        } else if (min == temp.size()) {
            ans.add(new ArrayList<String>(temp));
        }
        return;
    }

    if (temp.size() >= min) {
```

```
            return;
        }

        for (int i = 0; i < wordList.size(); i++) {
            String curWord = wordList.get(i);

            if (temp.contains(curWord)) {
                continue;
            }

            if (oneChanged(beginWord, curWord)) {
                temp.add(curWord);
                findLaddersHelper(curWord, endWord, wordList, temp, ans);
                temp.remove(temp.size() - 1);
            }
        }
    }
    private boolean oneChanged(String beginWord, String curWord) {
        int count = 0;
        for (int i = 0; i < beginWord.length(); i++) {
            if (beginWord.charAt(i) != curWord.charAt(i)) {
                count++;
            }
            if (count == 2) {
                return false;
            }
        }
        return count == 1;
    }
```

但是对于普通的输入可以解决，如果 `wordList` 过长的话就会造成超时了。

得到下一个的节点，如果采用第二种解法呢？

```
int min = Integer.MAX_VALUE;
public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
    List<List<String>> ans = new ArrayList<>();
    ArrayList<String> temp = new ArrayList<String>();
    temp.add(beginWord);

    findLaddersHelper(beginWord, endWord, wordList, temp, ans);
    return ans;
}
```

```
private void findLaddersHelper(String beginWord, String endWord, List<String> wordList,
                               ArrayList<String> temp, List<List<String>> ans) {
    if (beginWord.equals(endWord)) {
        if (min > temp.size()) {
            ans.clear();
            min = temp.size();
            ans.add(new ArrayList<String>(temp));
        } else if (min == temp.size()) {
            ans.add(new ArrayList<String>(temp));
        }
        return;
    }

    if (temp.size() >= min) {
        return;
    }
    Set<String> dict = new HashSet<>(wordList);

    ArrayList<String> neighbors = getNeighbors(beginWord, dict);
    for (String neighbor : neighbors) {
        if (temp.contains(neighbor)) {
            continue;
        }
        temp.add(neighbor);
        findLaddersHelper(neighbor, endWord, wordList,  temp, ans);
        temp.remove(temp.size() - 1);
    }
}


private ArrayList<String> getNeighbors(String node, Set<String> dict) {
    ArrayList<String> res = new ArrayList<String>();
    char chs[] = node.toCharArray();

    for (char ch = 'a'; ch <= 'z'; ch++) {
        for (int i = 0; i < chs.length; i++) {
            if (chs[i] == ch)
                continue;
            char old_ch = chs[i];
            chs[i] = ch;
            if (dict.contains(String.valueOf(chs))) {
                res.add(String.valueOf(chs));
            }
            chs[i] = old_ch;
        }

    }
    return res;
}
```

快了一些，但是还是超时。

Submission Detail

21 / 39 test cases passed.

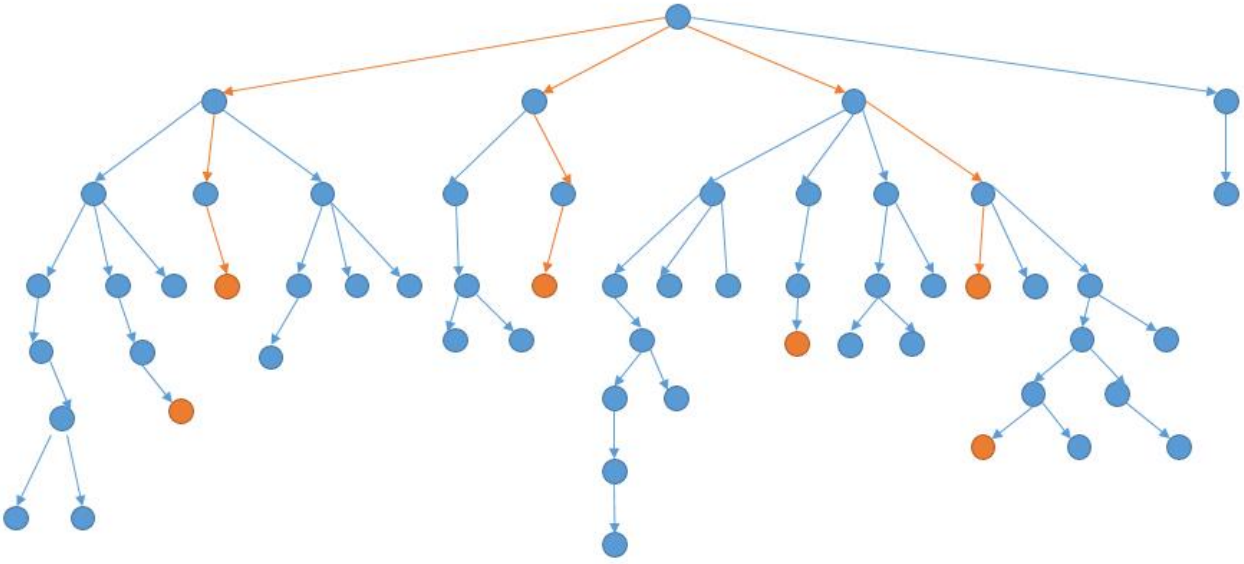
Status: Time Limit Exceeded

Submitted: 0 minutes ago

Last executed input:

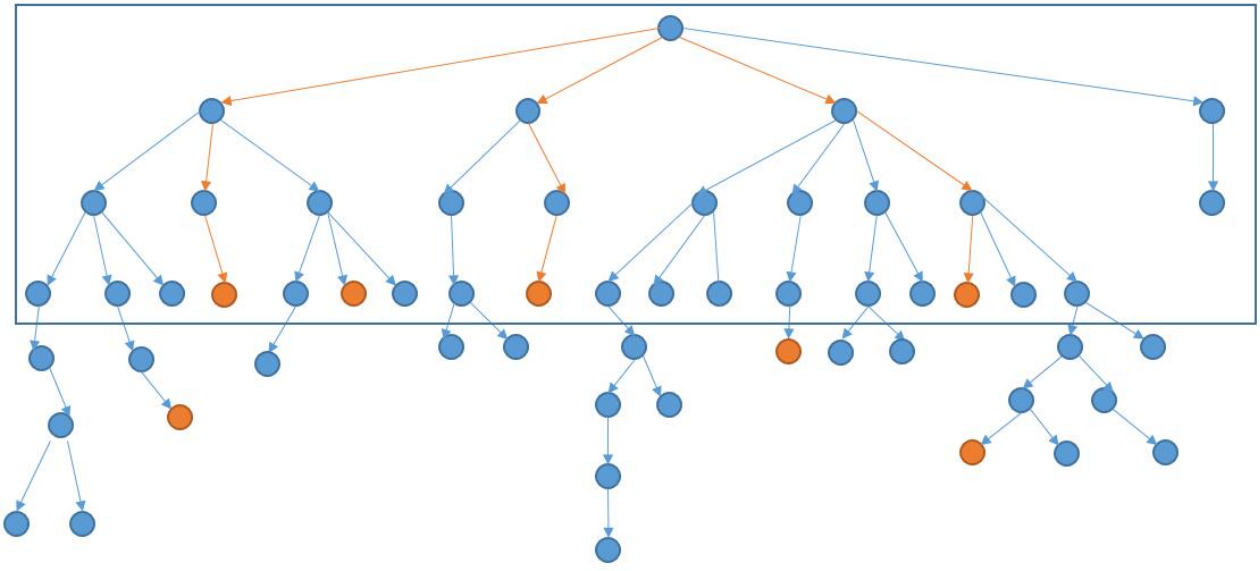
```
"cet"
"ism"
["kid","tag","pup","ail","tun","woo","erg","luz","brr","gay","sip","kay","per","val","mes","ohs","now","boa","ce
t","pal","bar","die","war","hav","eco","pub","lob","rue","fry","lit","rex","ian","cot","bid","ali","pay","co
```

我们继续来优化，首先想一下为什么会超时，看一下之前的图。



DFS 的过程的话，结合上图，就是先考虑了最左边的路径，然后再回溯一下，继续到达底部。然后回溯回溯，终于到了一条含有结束单词的路径，然而事实上这条并不是最短路径。综上，我们会多判断很多无用的路径。

如果我们事先知道了最短路径长度是 4 ，那么我们只需要考虑前 4 层就足够了。



怎么知道结束单词在哪一层呢？只能一层层的找了，也就是 `BFS` 。此外，因为上图需要搜索的树提前是没有的，我们需要边找边更新这个树。而在 `DFS` 中，我们也需要这个树，其实就是需要每个节点的所有相邻节点。

所以我们在 `BFS` 中，就把每个节点的所有相邻节点保存到 `HashMap` 中，就省去了 `DFS` 再去找相邻节点的时间。

此外， `BFS` 的过程中，把最短路径的高度用 `min` 也记录下来，在 `DFS` 的时候到达高度后就可以提前结束。

```
int min = 0;
public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
    List<List<String>> ans = new ArrayList<>();

    if (!wordList.contains(endWord)) {
        return ans;
    }

    HashMap<String, ArrayList<String>> map = bfs(beginWord, endWord, wordList);
    ArrayList<String> temp = new ArrayList<String>();

    temp.add(beginWord);
    findLaddersHelper(beginWord, endWord, map, temp, ans);
    return ans;
}

private void findLaddersHelper(String beginWord, String endWord, HashMap<String,
ArrayList<String>> map,
                               ArrayList<String> temp, List<List<String>> ans) {
    if (beginWord.equals(endWord)) {
        ans.add(new ArrayList<String>(temp));

        return;
    }
    if(temp.size() - 1==  min){
        return;
    }

    ArrayList<String> neighbors = map.getOrDefault(beginWord, new ArrayList<String>());
    for (String neighbor : neighbors) {
        if (temp.contains(neighbor)) {
            continue;
        }
        temp.add(neighbor);
        findLaddersHelper(neighbor, endWord, map, temp, ans);
        temp.remove(temp.size() - 1);
    }
}

public HashMap<String, ArrayList<String>> bfs(String beginWord, String endWord,
List<String> wordList) {
    Queue<String> queue = new LinkedList<>();
```

```
    queue.offer(beginWord);
    HashMap<String, ArrayList<String>> map = new HashMap<>();
    boolean isFound = false;

    Set<String> dict = new HashSet<>(wordList);
    while (!queue.isEmpty()) {
        int size = queue.size();
        min++;
        for (int j = 0; j < size; j++) {
            String temp = queue.poll();

            ArrayList<String> neighbors = getNeighbors(temp, dict);
            map.put(temp, neighbors);
            for (String neighbor : neighbors) {
                if (neighbor.equals(endWord)) {
                    isFound = true;
                }
                queue.offer(neighbor);
            }
        }
        if (isFound) {
            break;
        }
    }
    return map;
}
private ArrayList<String> getNeighbors(String node, Set<String> dict) {
    ArrayList<String> res = new ArrayList<String>();
    char chs[] = node.toCharArray();

    for (char ch = 'a'; ch <= 'z'; ch++) {
        for (int i = 0; i < chs.length; i++) {
            if (chs[i] == ch)
                continue;
            char old_ch = chs[i];
            chs[i] = ch;
            if (dict.contains(String.valueOf(chs))) {
                res.add(String.valueOf(chs));
            }
            chs[i] = old_ch;
        }

    }
    return res;
}
```

然而这个优化，对于 `leetcode` 的 `tests` 并没有什么影响。

Submission Detail

21 / 39 test cases passed.

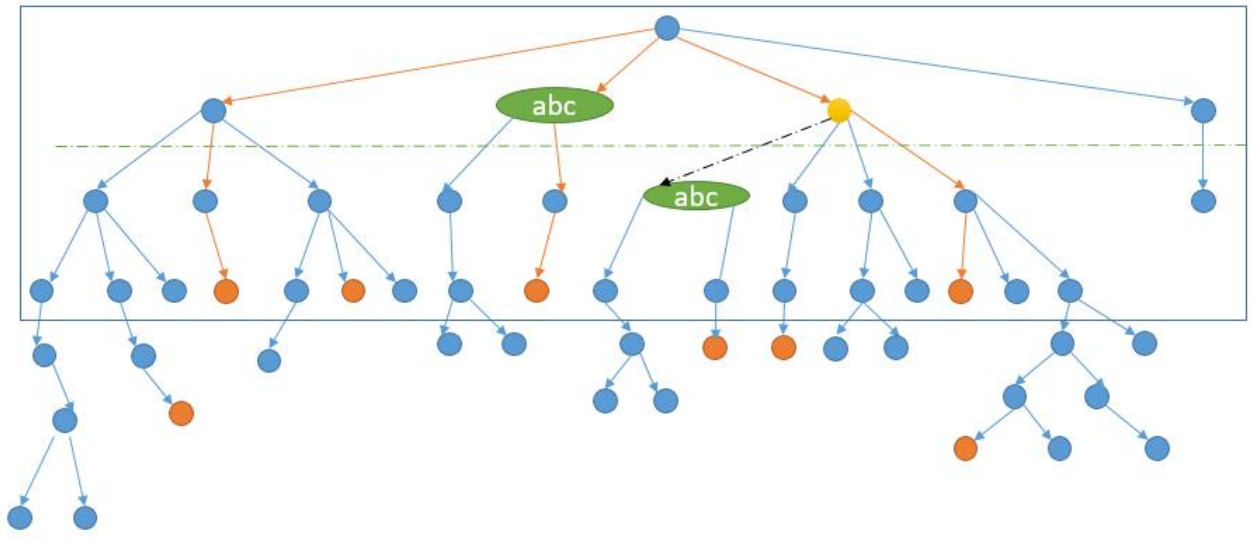
Status: Time Limit Exceeded

Submitted: 0 minutes ago

Last executed input:

```
"cet"
"ism"
["kid","tag","pup","ail","tun","woo","erg","luz","brr","gay","sip","kay","per","val","mes","ohs","now","boa","cet","pal","bar","die","war","hav","eco","pub","lob","rue","fry","lit","rex","jan","cot","bid","ali","pay","co
```

让我们继续考虑优化方案，回到之前的图。



假如我们在考虑上图中黄色节点的相邻节点，发现第三层的 `abc` 在第二层已经考虑过了。所以第三层的 `abc` 其实不用再考虑了，第三层的 `abc` 后边的结构一定和第二层后边的结构一样，因为我们要找最短的路径，所以如果产生了最短路径，一定是第二层的 `abc` 首先达到结束单词。

所以其实我们在考虑第 `k` 层的某一个单词，如果这个单词在第 `1` 到 `k-1` 层已经出现过，我们其实就不过继续向下探索了。

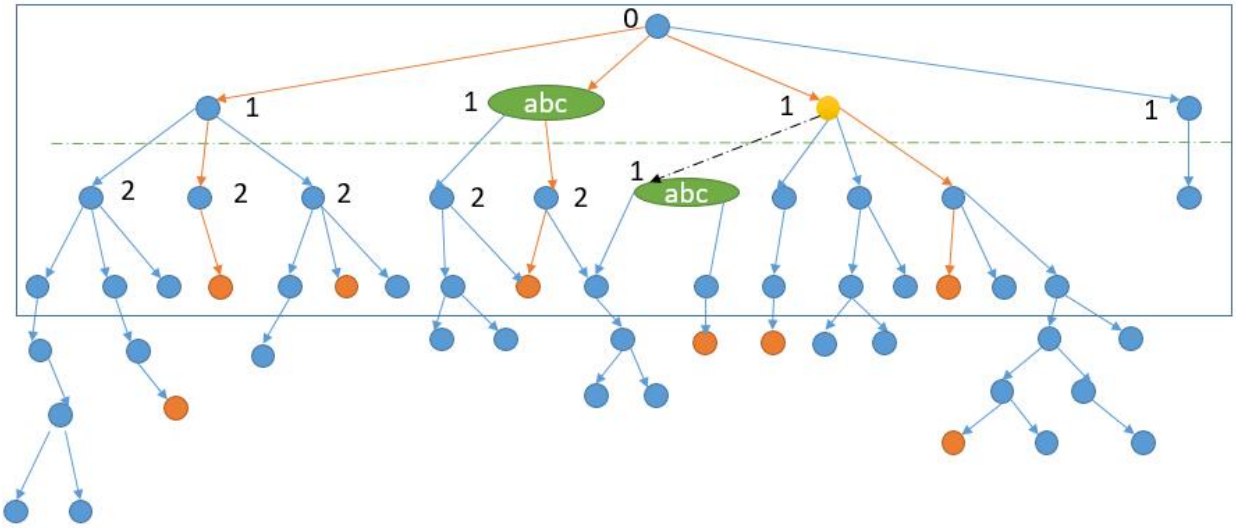
在之前的代码中，我们其实已经考虑了部分这个问题。

```
if (temp.contains(neighbor)) {
    continue;
}
```

但我们只考虑了当前路径是否含有该单词，而就像上图表示的，其他路径之前已经考虑过了当前单词，我们也是可以跳过的。

根据这个优化思路，有两种解决方案。

第一种，再利用一个 `HashMap` ，记为 `distance` 变量。在 `BFS` 的过程中，把第一次遇到的单词当前的层数存起来。之后遇到也不进行更新，就会是下边的效果。



这样我们就可以在 `DFS` 的时候来判断当前黄色的节点的 `distance` 是不是比邻接节点的小 `1` 。上图中 `distance` 都是 `1` ，所以不符合，就可以跳过。

此外，在 `DFS` 中，因为我们每次都根据节点的层数来进行深搜，所以之前保存最短路径的全局变量 `min` 在这里也就不需要了。

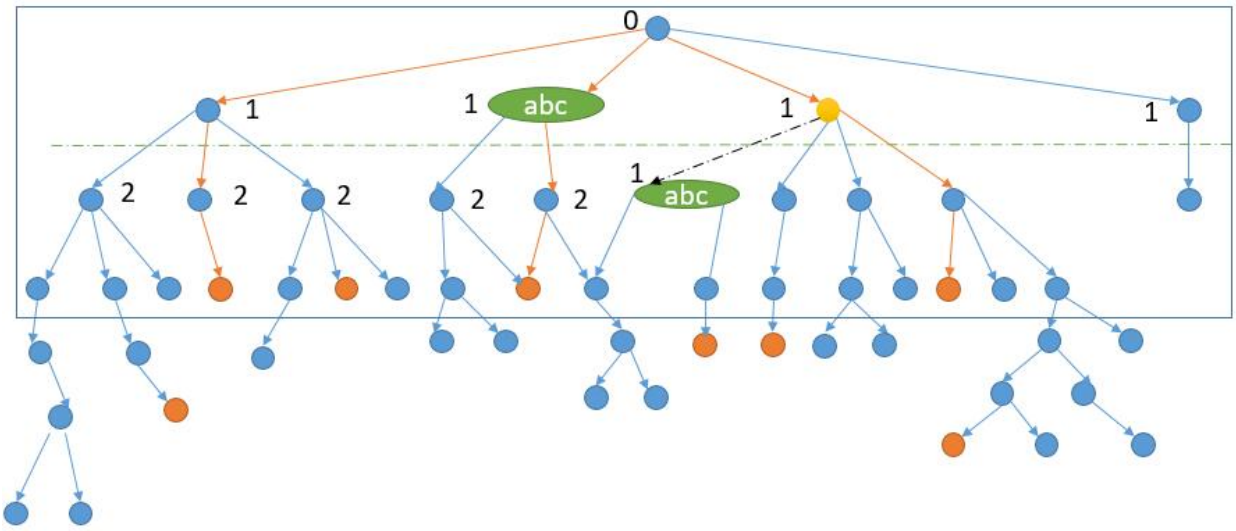
```
public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
    List<List<String>> ans = new ArrayList<>();

    if (!wordList.contains(endWord)) {
        return ans;
    }

    HashMap<String, Integer> distance = new HashMap<>();
    HashMap<String, ArrayList<String>> map = new HashMap<>();
    bfs(beginWord, endWord, wordList, map, distance);
    ArrayList<String> temp = new ArrayList<String>();

    temp.add(beginWord);
    findLaddersHelper(beginWord, endWord, map, distance, temp, ans);
    return ans;
}

private void findLaddersHelper(String beginWord, String endWord, HashMap<String,
ArrayList<String>> map,
                               HashMap<String, Integer> distance, ArrayList<String> temp,
List<List<String>> ans) {
    if (beginWord.equals(endWord)) {
        ans.add(new ArrayList<String>(temp));
        return;
    }



    ArrayList<String> neighbors = map.getOrDefault(beginWord, new ArrayList<String>());
    for (String neighbor : neighbors) {
```

```
        if (distance.get(beginWord) + 1 == distance.get(neighbor)) {
            temp.add(neighbor);
            findLaddersHelper(neighbor, endWord, map, distance, temp, ans);
            temp.remove(temp.size() - 1);
        }
    }
}

public void bfs(String beginWord, String endWord, List<String> wordList, HashMap<String,
ArrayList<String>> map,
                HashMap<String, Integer> distance) {
    Queue<String> queue = new LinkedList<>();
    queue.offer(beginWord);
    distance.put(beginWord, 0);
    boolean isFound = false;
    int depth = 0;
    Set<String> dict = new HashSet<>(wordList);
    while (!queue.isEmpty()) {
        int size = queue.size();
        depth++;
        for (int j = 0; j < size; j++) {
            String temp = queue.poll();

            ArrayList<String> neighbors = getNeighbors(temp, dict);
            map.put(temp, neighbors);
            for (String neighbor : neighbors) {
                if (!distance.containsKey(neighbor)) {
                    distance.put(neighbor, depth);
                    if (neighbor.equals(endWord)) {
                        isFound = true;
                    }
                    queue.offer(neighbor);
                }

            }
        }
        if (isFound) {
            break;
        }
    }
}

private ArrayList<String> getNeighbors(String node, Set<String> dict) {
    ArrayList<String> res = new ArrayList<String>();
    char chs[] = node.toCharArray();

    for (char ch = 'a'; ch <= 'z'; ch++) {
        for (int i = 0; i < chs.length; i++) {
            if (chs[i] == ch)
                continue;
            char old_ch = chs[i];
            chs[i] = ch;
             if (dict.contains(String.valueOf(chs))) {
```

```
                res.add(String.valueOf(chs));
            }
            chs[i] = old_ch;
        }

    }
    return res;
}
```

终于，上边的算法 AC 了。上边讲到我们提前存储了 distance ，方便在 DFS 中来判断我们是否继续深搜。

这里再讲一下另一种思路，再回顾一下这个要进行优化的图。



我们就是减少了第三层的 abc 的情况的判断。我们其实可以不用 distance ，在 BFS 中，如果发现有邻接节点在之前已经出现过了，我们直接把这个邻接节点删除不去。这样的话，在 DFS 中就不用再判断了，直接取邻居节点就可以了。

判断之前是否已经处理过，可以用一个 HashSet 来把之前的节点存起来进行判断。

这里删除邻接节点需要用到一个语言特性， java 中遍历 List 过程中，不能对 List 元素进行删除。如果想边遍历边删除，可以借助迭代器。

```
Iterator<String> it = neighbors.iterator();
while (it.hasNext()) {
    String neighbor = it.next();
    if (!visited.contains(neighbor)) {
        if (neighbor.equals(endWord)) {
            isFound = true;
        }
        queue.offer(neighbor);
        subVisited.add(neighbor);
    }else{
        it.remove();
    }
}
```

此外我们要判断的是当前节点在之前层有没有出现过，当前层正在遍历的节点先加到 `subVisited` 中。

```
public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
    List<List<String>> ans = new ArrayList<>();
    if (!wordList.contains(endWord)) {
        return ans;
    }

    HashMap<String, ArrayList<String>> map = new HashMap<>();
    bfs(beginWord, endWord, wordList, map);
    ArrayList<String> temp = new ArrayList<String>();

    temp.add(beginWord);
    findLaddersHelper(beginWord, endWord, map, temp, ans);
    return ans;
}

private void findLaddersHelper(String beginWord, String endWord, HashMap<String,
ArrayList<String>> map,
                               ArrayList<String> temp, List<List<String>> ans) {
    if (beginWord.equals(endWord)) {
        ans.add(new ArrayList<String>(temp));
        return;
    }

    ArrayList<String> neighbors = map.getOrDefault(beginWord, new ArrayList<String>());
    for (String neighbor : neighbors) {
        temp.add(neighbor);
        findLaddersHelper(neighbor, endWord, map, temp, ans);
        temp.remove(temp.size() - 1);

    }
}

public void bfs(String beginWord, String endWord, List<String> wordList, HashMap<String,
ArrayList<String>> map) {
    Queue<String> queue = new LinkedList<>();
    queue.offer(beginWord);
```

```
    boolean isFound = false;
    int depth = 0;
    Set<String> dict = new HashSet<>(wordList);
    Set<String> visited = new HashSet<>();
    visited.add(beginWord);
    while (!queue.isEmpty()) {
        int size = queue.size();
        depth++;
        Set<String> subVisited = new HashSet<>();
        for (int j = 0; j < size; j++) {
            String temp = queue.poll();

            ArrayList<String> neighbors = getNeighbors(temp, dict);
            Iterator<String> it = neighbors.iterator();
            while (it.hasNext()) {
                String neighbor = it.next();
                if (!visited.contains(neighbor)) {
                    if (neighbor.equals(endWord)) {
                        isFound = true;
                    }
                    queue.offer(neighbor);
                    subVisited.add(neighbor);
                }else{
                    it.remove();
                }
            }
            map.put(temp, neighbors);
        }
        visited.addAll(subVisited);
        if (isFound) {
            break;
        }
    }
}

private ArrayList<String> getNeighbors(String node, Set<String> dict) {
    ArrayList<String> res = new ArrayList<String>();
    char chs[] = node.toCharArray();

    for (char ch = 'a'; ch <= 'z'; ch++) {
        for (int i = 0; i < chs.length; i++) {
            if (chs[i] == ch)
                continue;
            char old_ch = chs[i];
            chs[i] = ch;
            if (dict.contains(String.valueOf(chs))) {
                res.add(String.valueOf(chs));
            }
            chs[i] = old_ch;
        }

    }
    return res;
}
```

解法二 BFS

如果理解了上边的 `DFS` 过程，接下来就很好讲了。上边 `DFS` 借助了 `BFS` 把所有的邻接关系保存了起来，再用 `DFS` 进行深度搜索。

我们可不可以只用 `BFS` ，一边进行层次遍历，一边就保存结果。当到达结束单词的时候，就把结果存储。省去再进行 `DFS` 的过程。

是完全可以的， `BFS` 的队列就不去存储 `String` 了，直接去存到目前为止的路径，也就是一个 `List` 。

```
public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
    List<List<String>> ans = new ArrayList<>();

    if (!wordList.contains(endWord)) {
        return ans;
    }
    bfs(beginWord, endWord, wordList, ans);
    return ans;
}

public void bfs(String beginWord, String endWord, List<String> wordList, List<List<String>>
ans) {
    Queue<List<String>> queue = new LinkedList<>();
    List<String> path = new ArrayList<>();
    path.add(beginWord);
    queue.offer(path);
    boolean isFound = false;
    Set<String> dict = new HashSet<>(wordList);
    Set<String> visited = new HashSet<>();
    visited.add(beginWord);
    while (!queue.isEmpty()) {
        int size = queue.size();
        Set<String> subVisited = new HashSet<>();
        for (int j = 0; j < size; j++) {
            List<String> p = queue.poll();

            String temp = p.get(p.size() - 1);

            ArrayList<String> neighbors = getNeighbors(temp, dict);
            for (String neighbor : neighbors) {

                if (!visited.contains(neighbor)) {

                    if (neighbor.equals(endWord)) {
                        isFound = true;
                        p.add(neighbor);
                        ans.add(new ArrayList<String>(p));
                        p.remove(p.size() - 1);
                    }

                    p.add(neighbor);
```

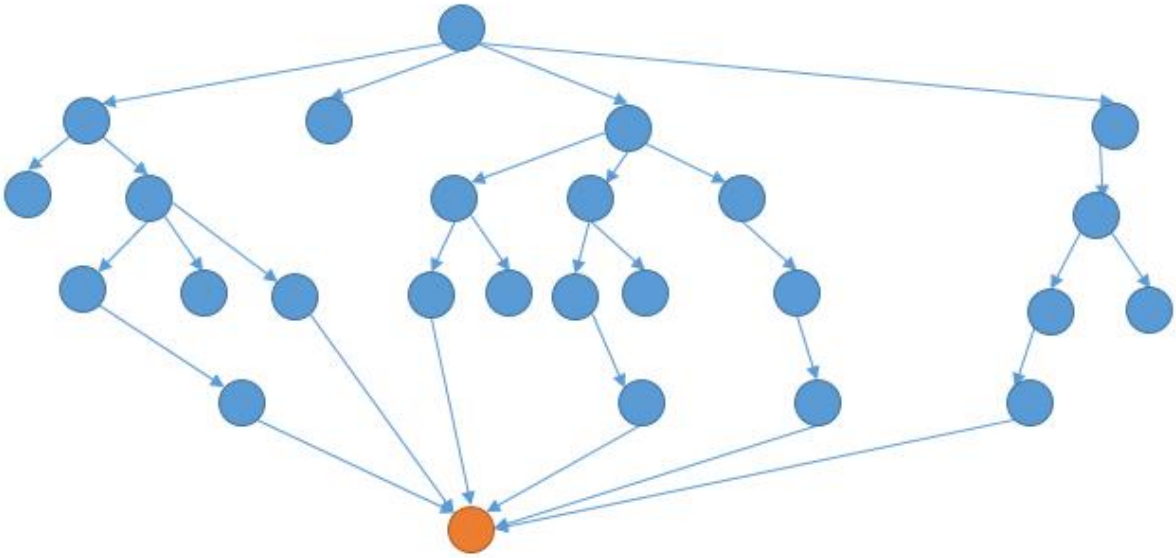
```
                queue.offer(new ArrayList<String>(p));
                p.remove(p.size() - 1);
                subVisited.add(neighbor);
            }
        }
    }
    visited.addAll(subVisited);
    if (isFound) {
        break;
    }
  }
}

private ArrayList<String> getNeighbors(String node, Set<String> dict) {
  ArrayList<String> res = new ArrayList<String>();
  char chs[] = node.toCharArray();
  for (char ch = 'a'; ch <= 'z'; ch++) {
    for (int i = 0; i < chs.length; i++) {
        if (chs[i] == ch)
            continue;
        char old_ch = chs[i];
        chs[i] = ch;
        if (dict.contains(String.valueOf(chs))) {
            res.add(String.valueOf(chs));
        }
        chs[i] = old_ch;
    }

  }
  return res;
}
```

代码看起来简洁了很多。

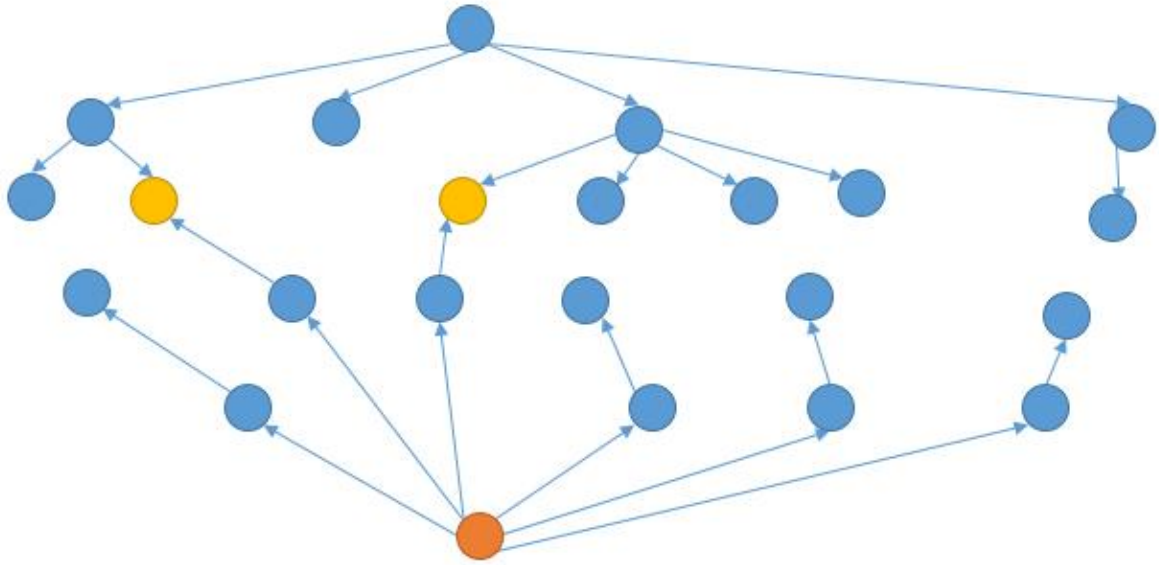
解法三 DFS + BFS 双向搜索（two-end BFS）

在解法一的思路上，我们还能够继续优化。

解法一中，我们利用了 `BFS` 建立了每个节点的邻居节点。在之前的示意图中，我们把同一个字符串也画在了不同节点。这里把同一个节点画在一起，再看一下。



我们可以从结束单词反向进行 `BFS` 。



这样的话，当两个方向产生了共同的节点，就是我们的最短路径了。

至于每次从哪个方向扩展，我们可以每次选择需要扩展的节点数少的方向进行扩展。

例如上图中，一开始需要向下扩展的个数是 `1` 个，需要向上扩展的个数是 `1` 个。个数相等，我们就向下扩展。然后需要向下扩展的个数就变成了 `4` 个，而需要向上扩展的个数是 `1` 个，所以此时我们向上扩展。接着，需要向上扩展的个数变成了 `6` 个，需要向下扩展的个数是 `4` 个，我们就向下扩展……直到相遇。

双向扩展的好处，我们粗略的估计一下时间复杂度。

假设 `beginword` 和 `endword` 之间的距离是 `d` 。每个节点可以扩展出 `k` 个节点。

那么正常的时间复杂就是 kdk^dkd。

双向搜索的时间复杂度就是 kd/2+kd/2k^{d/2} + k^{d/2}kd/2+kd/2。

```
public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
    List<List<String>> ans = new ArrayList<>();
    if (!wordList.contains(endWord)) {
        return ans;
    }

    HashMap<String, ArrayList<String>> map = new HashMap<>();
    bfs(beginWord, endWord, wordList, map);
    ArrayList<String> temp = new ArrayList<String>();

    temp.add(beginWord);
    findLaddersHelper(beginWord, endWord, map, temp, ans);
    return ans;
}

private void findLaddersHelper(String beginWord, String endWord, HashMap<String,
ArrayList<String>> map,
                               ArrayList<String> temp, List<List<String>> ans) {
    if (beginWord.equals(endWord)) {
        ans.add(new ArrayList<String>(temp));
        return;
    }

    ArrayList<String> neighbors = map.getOrDefault(beginWord, new ArrayList<String>());
    for (String neighbor : neighbors) {
        temp.add(neighbor);
        findLaddersHelper(neighbor, endWord, map, temp, ans);
        temp.remove(temp.size() - 1);
    }
}


private void bfs(String beginWord, String endWord, List<String> wordList, HashMap<String,
ArrayList<String>> map) {
    Set<String> set1 = new HashSet<String>();
    set1.add(beginWord);
    Set<String> set2 = new HashSet<String>();
    set2.add(endWord);
    Set<String> wordSet = new HashSet<String>(wordList);
    bfsHelper(set1, set2, wordSet, true, map);
}


private boolean bfsHelper(Set<String> set1, Set<String> set2, Set<String> wordSet, boolean
direction,
                          HashMap<String, ArrayList<String>> map) {



    if(set1.isEmpty()){
        return false;
```

```
    }

    if (set1.size() > set2.size()) {
        return bfsHelper(set2, set1, wordSet, !direction, map);
    }

    wordSet.removeAll(set1);
    wordSet.removeAll(set2);

    boolean done = false;


    Set<String> set = new HashSet<String>();

    for (String str : set1) {

        for (int i = 0; i < str.length(); i++) {
            char[] chars = str.toCharArray();


            for (char ch = 'a'; ch <= 'z'; ch++) {
                if(chars[i] == ch){
                    continue;
                }
                chars[i] = ch;

                String word = new String(chars);


                String key = direction ? str : word;
                String val = direction ? word : str;

                ArrayList<String> list = map.containsKey(key) ? map.get(key) : new
ArrayList<String>();


                if (set2.contains(word)) {
                    done = true;
                    list.add(val);
                    map.put(key, list);
                }


                if (!done && wordSet.contains(word)) {
                    set.add(word);
                    list.add(val);
                    map.put(key, list);
                }
            }
        }
    }


    return done || bfsHelper(set2, set, wordSet, !direction, map);
```

```
}
```

总

最近事情比较多，这道题每天想一想，陆陆续续拖了好几天了。这道题本质上就是在正常的遍历的基础上，去将一些分支剪去，从而提高速度。至于方法的话，除了我上边介绍的实现方式，应该也会有很多其它的方式，但其实本质上是为了实现一样的东西。另外，双向搜索的方法，自己第一次遇到，网上搜了。