Word Ladder II

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

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the word list

For example,

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

Return

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

Note:

- All words have the same length.
- All words contain only lowercase alphabetic characters.

Solution 1

The solution contains two steps 1 Use BFS to construct a graph. 2. Use DFS to construct the paths from end to start.Both solutions got AC within 1s.

The first step BFS is quite important. I summarized three tricks

1) Using a **MAP** to store the min ladder of each word, or use a **SET** to store the words visited in current ladder, when the current ladder was completed, delete the visited words from unvisited. That's why I have two similar solutions.

2) Use **Character iteration** to find all possible paths. Do not compare one word to all the other words and check if they only differ by one character.

3) One word is allowed to be inserted into the queue only **ONCE**. See my comments.

```
public class Solution {
    Map<String,List<String>> map;
    List<List<String>> results;
    public List<List<String>> findLadders(String start, String end, Set<String> d
ict) {
        results= new ArrayList<List<String>>();
        if (dict.size() == 0)
            return results;

        int min=Integer.MAX_VALUE;

        Queue<String> queue= new ArrayDeque<String>();
        queue.add(start);

        map = new HashMap<String,List<String>>();

        Map<String,Integer> ladder = new HashMap<String,Integer>();
        for (String string:dict)
            ladder.put(string, Integer.MAX_VALUE);
        ladder.put(start, 0);

        dict.add(end);
        //BFS: Dijisktra search
        while (!queue.isEmpty()) {

            String word = queue.poll();

            int step = ladder.get(word)+1;//'step' indicates how many steps are n
eeded to travel to one word.

            if (step>min) break;

            for (int i = 0; i < word.length(); i++){
               StringBuilder builder = new StringBuilder(word);
                for (char ch='a';  ch <= 'z'; ch++){
                    builder.setCharAt(i,ch);
                    String new_word=builder.toString();
                    if (ladder.containsKey(new_word)) {

                        if (step>ladder.get(new_word))//Check if it is the shorte
st path to one word
```

```
st path to one word.
                            continue;
                        else if (step<ladder.get(new_word)){
                            queue.add(new_word);
                            ladder.put(new_word, step);
                        }else;// It is a KEY line. If one word already appeared i
n one ladder,
                              // Do not insert the same word inside the queue twi
ce. Otherwise it gets TLE.

                        if (map.containsKey(new_word)) //Build adjacent Graph
                            map.get(new_word).add(word);
                        else{
                            List<String> list= new LinkedList<String>();
                            list.add(word);
                            map.put(new_word,list);
                            //It is possible to write three lines in one:
                            //map.put(new_word,new LinkedList<String>(Arrays.asLi
st(new String[]{word})));
                            //Which one is better?
                        }

                        if (new_word.equals(end))
                            min=step;

                    }//End if dict contains new_word
                }//End:Iteration from 'a' to 'z'
            }//End:Iteration from the first to the last
        }//End While

        //BackTracking
        LinkedList<String> result = new LinkedList<String>();
        backTrace(end,start,result);

        return results;
    }
    private void backTrace(String word,String start,List<String> list){
        if (word.equals(start)){
            list.add(0,start);
            results.add(new ArrayList<String>(list));
            list.remove(0);
            return;
        }
        list.add(0,word);
        if (map.get(word)!=null)
            for (String s:map.get(word))
                backTrace(s,start,list);
        list.remove(0);
    }
}
```

Another solution using two sets. This is similar to the answer in the most viewed thread. While I found my solution more readable and efficient.

```
public class Solution {
    List<List<String>> results;
```

```
    List<String> list;
    Map<String,List<String>> map;
        public List<List<String>> findLadders(String start, String end, Set<Strin
g> dict) {
            results= new ArrayList<List<String>>();
            if (dict.size() == 0)
                return results;

            int curr=1,next=0;
            boolean found=false;
            list = new LinkedList<String>();
            map = new HashMap<String,List<String>>();

            Queue<String> queue= new ArrayDeque<String>();
            Set<String> unvisited = new HashSet<String>(dict);
            Set<String> visited = new HashSet<String>();

            queue.add(start);
            unvisited.add(end);
            unvisited.remove(start);
            //BFS
            while (!queue.isEmpty()) {

                String word = queue.poll();
                curr--;
                for (int i = 0; i < word.length(); i++){
                   StringBuilder builder = new StringBuilder(word);
                    for (char ch='a';  ch <= 'z'; ch++){
                        builder.setCharAt(i,ch);
                        String new_word=builder.toString();
                        if (unvisited.contains(new_word)){
                            //Handle queue
                            if (visited.add(new_word)){//Key statement,Avoid Dupl
icate queue insertion
                                next++;
                                queue.add(new_word);
                            }

                            if (map.containsKey(new_word))//Build Adjacent Graph
                                map.get(new_word).add(word);
                            else{
                                List<String> l= new LinkedList<String>();
                                l.add(word);
                                map.put(new_word, l);
                            }

                            if (new_word.equals(end)&&!found) found=true;

                        }

                    }//End:Iteration from 'a' to 'z'
                }//End:Iteration from the first to the last
                if (curr==0){
                    if (found) break;
                    curr=next;
                    next=0;
                    unvisited.removeAll(visited);
```

```
                visited.clear();
            }
        }//End While

        backTrace(end,start);

        return results;
    }
    private void backTrace(String word,String start){
        if (word.equals(start)){
            list.add(0,start);
            results.add(new ArrayList<String>(list));
            list.remove(0);
            return;
        }
        list.add(0,word);
        if (map.get(word)!=null)
            for (String s:map.get(word))
                backTrace(s,start);
        list.remove(0);
    }
}
```

written by reeclapple original link here

Solution 2

In order to reduce the running time, we should use two-end BFS to slove the problem.

Accepted 68ms c++ solution for Word Ladder.

```
class Solution {
public:
    int ladderLength(std::string beginWord, std::string endWord, std::unordered_s
et<std::string> &dict) {
        if (beginWord == endWord)
            return 1;
        std::unordered_set<std::string> words1, words2;
        words1.insert(beginWord);
        words2.insert(endWord);
        dict.erase(beginWord);
        dict.erase(endWord);
        return ladderLengthHelper(words1, words2, dict, 1);
    }

private:
    int ladderLengthHelper(std::unordered_set<std::string> &words1, std::unordere
d_set<std::string> &words2, std::unordered_set<std::string> &dict, int level) {
        if (words1.empty())
            return 0;
        if (words1.size() > words2.size())
            return ladderLengthHelper(words2, words1, dict, level);
        std::unordered_set<std::string> words3;
        for (auto it = words1.begin(); it != words1.end(); ++it) {
            std::string word = *it;
            for (auto ch = word.begin(); ch != word.end(); ++ch) {
                char tmp = *ch;
                for (*ch = 'a'; *ch <= 'z'; ++(*ch))
                    if (*ch != tmp)
                        if (words2.find(word) != words2.end())
                            return level + 1;
                        else if (dict.find(word) != dict.end()) {
                            dict.erase(word);
                            words3.insert(word);
                        }
                *ch = tmp;
            }
        }
        return ladderLengthHelper(words2, words3, dict, level + 1);
    }
};
```

Accepted 88ms c++ solution for Word Ladder II.

```
class Solution {
public:
    std::vector<std::vector<std::string> > findLadders(std::string beginWord, std
::string endWord, std::unordered_set<std::string> &dict) {
        std::vector<std::vector<std::string> > paths;
        std::vector<std::string> path(1, beginWord);
```

```
        std::vector<std::string> path(1, beginWord);
        if (beginWord == endWord) {
            paths.push_back(path);
            return paths;
        }
        std::unordered_set<std::string> words1, words2;
        words1.insert(beginWord);
        words2.insert(endWord);
        std::unordered_map<std::string, std::vector<std::string> > nexts;
        bool words1IsBegin = false;
        if (findLaddersHelper(words1, words2, dict, nexts, words1IsBegin))
            getPath(beginWord, endWord, nexts, path, paths);
        return paths;
    }
private:
    bool findLaddersHelper(
        std::unordered_set<std::string> &words1,
        std::unordered_set<std::string> &words2,
        std::unordered_set<std::string> &dict,
        std::unordered_map<std::string, std::vector<std::string> > &nexts,
        bool &words1IsBegin) {
        words1IsBegin = !words1IsBegin;
        if (words1.empty())
            return false;
        if (words1.size() > words2.size())
            return findLaddersHelper(words2, words1, dict, nexts, words1IsBegin);
        for (auto it = words1.begin(); it != words1.end(); ++it)
            dict.erase(*it);
        for (auto it = words2.begin(); it != words2.end(); ++it)
            dict.erase(*it);
        std::unordered_set<std::string> words3;
        bool reach = false;
        for (auto it = words1.begin(); it != words1.end(); ++it) {
            std::string word = *it;
            for (auto ch = word.begin(); ch != word.end(); ++ch) {
                char tmp = *ch;
                for (*ch = 'a'; *ch <= 'z'; ++(*ch))
                    if (*ch != tmp)
                        if (words2.find(word) != words2.end()) {
                            reach = true;
                            words1IsBegin ? nexts[*it].push_back(word) : nexts[wo
rd].push_back(*it);
                        }
                        else if (!reach && dict.find(word) != dict.end()) {
                            words3.insert(word);
                            words1IsBegin ? nexts[*it].push_back(word) : nexts[wo
rd].push_back(*it);
                        }
                *ch = tmp;
            }
        }
        return reach || findLaddersHelper(words2, words3, dict, nexts, words1IsBe
gin);
    }
    void getPath(
        std::string beginWord,
        std::string &endWord,
```

```
        std::unordered_map<std::string, std::vector<std::string> > &nexts,
        std::vector<std::string> &path,
        std::vector<std::vector<std::string> > &paths) {
        if (beginWord == endWord)
            paths.push_back(path);
        else
            for (auto it = nexts[beginWord].begin(); it != nexts[beginWord].end()
; ++it) {
                path.push_back(*it);
                getPath(*it, endWord, nexts, path, paths);
                path.pop_back();
            }
    }
};
```

written by prime_tang original link here

Solution 3

```
class Solution:
# @param start, a string
# @param end, a string
# @param dict, a set of string
# @return a list of lists of string
def findLadders(self, start, end, dic):
    dic.add(end)
    level = {start}
    parents = collections.defaultdict(set)
    while level and end not in parents:
        next_level = collections.defaultdict(set)
        for node in level:
            for char in string.ascii_lowercase:
                for i in range(len(start)):
                    n = node[:i]+char+node[i+1:]
                    if n in dic and n not in parents:
                        next_level[n].add(node)
        level = next_level
        parents.update(next_level)
    res = [[end]]
    while res and res[0][0] != start:
        res = [[p]+r for r in res for p in parents[r[0]]]
    return res
```

Every level we use the defaultdict to get rid of the duplicates

written by tusizi original link here