LeetCode 737

https://leetcode.com/problems/sentence-similarity-ii/description/

Yifeng Zeng

Description

737. Sentence Similarity II

Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two sentences are similar.

For example, words1 = ["great", "acting", "skills"] and words2 = ["fine", "drama", "talent"] are similar, if the similar word pairs are pairs = [["great", "good"], ["fine", "good"], ["acting","drama"], ["skills","talent"]].

Note that the similarity relation is transitive. For example, if "great" and "good" are similar, and "fine" and "good" are similar, then "great" and "fine" are similar.

Similarity is also symmetric. For example, "great" and "fine" being similar is the same as "fine" and "great" being similar.

Also, a word is always similar with itself. For example, the sentences words1 = ["great"], words2 = ["great"], pairs = [] are similar, even though there are no specified similar word pairs.

Finally, sentences can only be similar if they have the same number of words. So a sentence like words1 = ["great"] can never be similar to words2 = ["doubleplus","good"].

Note:

- The length of words1 and words2 will not exceed 1000.
- The length of pairs will not exceed 2000.
- The length of each pairs[i] will be 2.
- The length of each words[i] and pairs[i][j] will be in the range [1, 20].

Idea Report

This problem can be represented as a search on undirected graph problem. For each pair of words in the two sentences, we can do a graph search to see if these two words are connected. The input paris contains all the edge information between all the nodes. We can build up a adjacency list (String to Set map) based on the input pairs array. Then we traverse the two sentences, and pick up the words in the same index of the senetences (word1, word2). If words1 is equals word2, we don't have to do any search and continue to the next index. Otherwise we see if word1 and word2 are connected in the graph. If they are, we continue to the next index, otherwise we return false meaning these two words are not similar, which means the sentences are not similar. We can do both BFS or DFS for the search in the graph.

Code:

```
class Solution {
    // BFS AC
    public boolean areSentencesSimilarTwo(String[] words1, String[] words2,
                                          String[][] pairs) {
        if (words1.length != words2.length) {
            return false;
        }

        Map<String, Set<String>> map = initMap(pairs);

        for (int i = 0; i < words1.length; i++) {
            if (!isSimilar(words1[i], words2[i], map)) {
                return false;
            }
        }
        return true;
    }

    private boolean isSimilar(String str1, String str2,
                              Map<String, Set<String>> map) {
        if (str1.equals(str2)) {
            return true;
        }
        if (!map.containsKey(str1) || !map.containsKey(str2)) {
            return false;
        }

        Set<String> visited = new HashSet<>();
        Deque<String> q = new LinkedList<>();
        visited.add(str1);
        q.offer(str1);
        while (!q.isEmpty()) {
            String cur = q.poll();
            for (String next : map.get(cur)) {
                if (next.equals(str2)) {
                    return true;
                }
                if (!visited.contains(next)) {
                    q.offer(next);
                    visited.add(next);
                }
            }
        }
        return false;
    }

    private Map<String, Set<String>> initMap(String[][] pairs) {
        Map<String, Set<String>> map = new HashMap<>();
        for (String[] p : pairs) {
            map.put(p[0], map.getOrDefault(p[0], new HashSet<>()));
            map.get(p[0]).add(p[1]);
            map.put(p[1], map.getOrDefault(p[1], new HashSet<>()));
            map.get(p[1]).add(p[0]);
        }

        return map;
    }
}
```

Code:

```
class Solution {
    // DFS AC
    public boolean areSentencesSimilarTwo(String[] words1, String[] words2,
                                          String[][] pairs) {
      if (words1.length != words2.length) {
          return false;
      }

      Map<String, Set<String>> map = initMap(pairs);
      for (int i = 0; i < words1.length; i++) {
          if (!isSimilar(words1[i], words2[i], map)) {
              return false;
          }
      }

      return true;
    }
    private boolean isSimilar(String source, String target, Map<String,
                              Set<String>> map) {
      if (source.equals(target)) {
          return true;
      }
      if (!map.containsKey(source) || !map.containsKey(target)) {
          return false;
      }

      Set<String> visited = new HashSet<>();
      visited.add(source);
      return dfsHelper(source, target, map, visited);
    }

    private boolean dfsHelper(String cur, String target,
                              Map<String, Set<String>> map,
                              Set<String> visited) {
      if (cur.equals(target)) {
          return true;
      }

      for (String next : map.get(cur)) {
          if (!visited.contains(next)) {
              visited.add(next);
              if (dfsHelper(next, target, map, visited)) {
                  return true;
              }
          }
      }

      return false;
    }

    private Map<String, Set<String>> initMap(String[][] pairs) {
      Map<String, Set<String>> map = new HashMap<>();
      for (String[] p : pairs) {
          map.put(p[0], map.getOrDefault(p[0], new HashSet<>()));
          map.get(p[0]).add(p[1]);
          map.put(p[1], map.getOrDefault(p[1], new HashSet<>()));
          map.get(p[1]).add(p[0]);
      }

      return map;
    }
}
```

In an undirected graph, we can also use union find to store the information about which nodes are connected together instead of using BFS/DFS to do the search to see if aforementioned word1 and word2 are connected in the graph. We use a String to String map to store the String itself and the parent String. We first initialize the map, connect all the nodes in the graph based on the pairs input. Then when traversing the words in the two sentences, we compare word1 and word2 is equal or has the same parent, if not we return false meaning they are not similar, otherwise continue to the next index.

Code:

```
class Solution {
    // union find AC
    public boolean areSentencesSimilarTwo(String[] words1, String[] words2,
                                          String[][] pairs) {
        if (words1.length != words2.length) {
            return false;
        }

        Map<String, String> parents = initMap(pairs);
        for (int i = 0; i < words1.length; i++) {
            if (!isSimilar(words1[i], words2[i], parents)) {
                return false;
            }
        }

        return true;
    }

    private boolean isSimilar(String source, String target,
                              Map<String, String> parents) {
        if (source.equals(target)) {
            return true;
        }
        if (!parents.containsKey(source) || !parents.containsKey(target)) {
            return false;
        }
        String pa = find(parents, source);
        String pb = find(parents, target);
        return pa.equals(pb);
    }

    private Map<String, String> initMap(String[][] pairs) {
        Map<String, String> parents = new HashMap<>();
        for (String[] p : pairs) {
            if (!parents.containsKey(p[0])) {
                parents.put(p[0], p[0]);
            }
            if (!parents.containsKey(p[1])) {
                parents.put(p[1], p[1]);
            }
            String pa = find(parents, p[0]);
            String pb = find(parents, p[1]);
            // union
            if (!pa.equals(pb)) {
                parents.put(pa, pb);
            }
        }

        return parents;
    }

    private String find(Map<String, String> parents, String str) {
        while (!parents.get(str).equals(str)) {
            parents.put(str, parents.get(parents.get(str)));
            str = parents.get(str);
        }
        return str;
    }
}
```

Summary

- The relationship between one thing (word) and the other (word) are represented as a connection (pair), then this can be represented as graph search problem.
- If the graph is undirected, we can consider using union find instead of BFS/DFS search.