Articles ›

Previous (/articles/sentence-similarity/) Next (/articles/bulb-switcher-ii/)

737. Sentence Similarity II (/problems/sentence-similarity-ii/)

ırn=/articles/sentence-similarity-ii/) (/ratings/107/301/?return=/articles/sentence-similarity-ii/) (/ratings/107/301/?return=/articles/sentence-similarity-ii/)

Average Rating: 5 (1 votes)

Nov. 25, 2017 | 3K views

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.

737. Sentence Similarity II ▾

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]`.

Approach #1: Depth-First Search [Accepted]

Intuition

Two words are similar if they are the same, or there is a path connecting them from edges represented by `pairs`.

We can check whether this path exists by performing a depth-first search from a word and seeing if we reach the other word. The search is performed on the underlying graph specified by the edges in `pairs`.

Algorithm

We start by building our `graph` from the edges in `pairs`.

The specific algorithm we go for is an iterative depth-first search. The implementation we go for is a typical "visitor pattern": when searching whether there is a path from `w1 = words1[i]` to `w2 = words2[i]`, `stack` will contain all the nodes that are queued up for processing, while `seen` will be all the nodes that have been queued for processing (whether they have been processed or not).

Articles ›

Java Python Copy

```
class Solution {
    public boolean areSentencesSimilarTwo(
            String[] words1, String[] words2, String[][] pairs) {
        if (words1.length != words2.length) return false;
        Map<String, List<String>> graph = new HashMap();
        for (String[] pair: pairs) {
            for (String p: pair) if (!graph.containsKey(p)) {
                graph.put(p, new ArrayList());
            }
            graph.get(pair[0]).add(pair[1]);
            graph.get(pair[1]).add(pair[0]);
        }

        for (int i = 0; i < words1.length; ++i) {
            String w1 = words1[i], w2 = words2[i];
            Stack<String> stack = new Stack();
            Set<String> seen = new HashSet();
            stack.push(w1);
            seen.add(w1);
            search: {
                while (!stack.isEmpty()) {
                    String word = stack.pop();
                    if (word.equals(w2)) break search;
                    if (graph.containsKey(word)) {
                        for (String nei: graph.get(word)) {
                            if (!seen.contains(nei)) {
                                stack.push(nei);
```

Complexity Analysis

- Time Complexity: $O(NP)$, where N is the maximum length of `words1` and `words2`, and P is the length of `pairs`. Each of N searches could search the entire graph.
- Space Complexity: $O(P)$, the size of `pairs`.

Approach #2: Union-Find [Accepted]

Intuition

As in *Approach #1*, we want to know if there is path connecting two words from edges represented by `pairs`.

Our problem comes down to finding the connected components of a graph. This is a natural fit for a *Disjoint Set Union* (DSU) structure.

Algorithm

Draw edges between words if they are similar. For easier interoperability between our DSU template, we will map each `word` to some integer `ix = index[word]`. Then, `dsu.find(ix)` will tell us a unique id representing what component that word is in.

For more information on DSU, please look at *Approach #2* in the article here (https://leetcode.com/articles/redundant-connection/). For brevity, the solutions showcased below do not use *union-by-rank*.

After putting each word in `pairs` into our DSU template, we check successive pairs of words `w1, w2 = words1[i], words2[i]`. We require that `w1 == w2`, or `w1` and `w2` are in the same component. This is easily checked using `dsu.find`.

Articles

Java Python Copy

```java
class Solution {
    public boolean areSentencesSimilarTwo(String[] words1, String[] words2, String[][] pairs) {
        if (words1.length != words2.length) return false;
        Map<String, Integer> index = new HashMap();
        int count = 0;
        DSU dsu = new DSU(2 * pairs.length);
        for (String[] pair: pairs) {
            for (String p: pair) if (!index.containsKey(p)) {
                index.put(p, count++);
            }
            dsu.union(index.get(pair[0]), index.get(pair[1]));
        }

        for (int i = 0; i < words1.length; ++i) {
            String w1 = words1[i], w2 = words2[i];
            if (w1.equals(w2)) continue;
            if (!index.containsKey(w1) || !index.containsKey(w2) ||
                    dsu.find(index.get(w1)) != dsu.find(index.get(w2)))
                return false;
        }
        return true;
    }
}

class DSU {
    int[] parent;
    public DSU(int N) {
```

Complexity Analysis

- Time Complexity: $O(N \log P + P)$, where N is the maximum length of `words1` and `words2`, and P is the length of `pairs`. If we used union-by-rank, this complexity improves to $O(N * \alpha(P) + P) \approx O(N + P)$, where α is the *Inverse-Ackermann* function.
- Space Complexity: $O(P)$, the size of `pairs`.

Analysis written by: @awice (https://leetcode.com/awice).

Rate this article:

(/ratings/107/301/?return=/articles/sentence-similarity-ii/) (/ratings/107/301/?return=/articles/sen

Previous (/articles/sentence-similarity/)

Next (/articles/bulb-switcher-ii/)

Join the conversation

Login to Reply

W (https://discuss.leetcode.com/user/wang1993)

Wang1993 commented 9 hours ago

@wwan (https://discuss.leetcode.com/uid/169929) I agree with you

E (https://discuss.leetcode.com/user/euiow21391)

euiow21391 commented last week

This union operation is not weighted, so the time complexity won't be logP, it could be P?

(https://discuss.leetcode.com/user/awice)

awice commented last month

@IWantToPass (https://discuss.leetcode.com/uid/754) In each DFS, we visit at most P+1 nodes. The "|V|" is not N necessarily.
@wqmbisheng (https://discuss.leetcode.com/uid/228659) There are N DFS's, each DFS is O(P).

☰ Articles ›

W

(https://discuss.leetcode.com/user/wqmbisheng)

wqmbisheng commented last month

Hi, I agree with the time complexity of DFS method is O(N + P), what do you think? @awice (https://discuss.leetcode.com/uid/71269)

I

(https://discuss.leetcode.com/user/iwanttopass)

IWantToPass commented 2 months ago

I think the time complexity of DFS method is not right
The time complexity of DFS is O(|V| + |E|), and here, |V| = N, and |E| = P. The DFS is executed "N" times, so then shouldn't the time complexity be O(N(N + P)) ?

W

(https://discuss.leetcode.com/user/wwan)

wwan commented 2 months ago

I think the time complexity for the union-find not by rank method is not right. the first loop to construct dsu calls union O(P) times and union runs in O(logP) time because it called find. So the overall should be O((N+P)log P)?

View original thread (https://discuss.leetcode.com/topic/112017)

Contact Us (/support/) | Frequently Asked Questions (/faq/) | Terms of Service (/terms/) | Privacy Policy (/privacy/)

🇺🇸 United States (/region/)