737. Sentence Similarity II

Nov. 25, 2017 | 19.3K views

Average Rating: 3.90 (10 votes)

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]`.

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).

Java | Python — Copy

```
class Solution(object):
    def areSentencesSimilarTwo(self, words1, words2, pairs):
        if len(words1) != len(words2): return False
        graph = collections.defaultdict(list)
        for w1, w2 in pairs:
            graph[w1].append(w2)
            graph[w2].append(w1)

        for w1, w2 in zip(words1, words2):
            stack, seen = [w1], {w1}
            while stack:
                word = stack.pop()
                if word == w2: break
                for nei in graph[word]:
                    if nei not in seen:
                        seen.add(nei)
                        stack.append(nei)
            else:
                return False
        return True
```

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. 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`.

Java | Python — Copy

```
class DSU:
    def __init__(self, N):
        self.par = range(N)
    def find(self, x):
        if self.par[x] != x:
            self.par[x] = self.find(self.par[x])
        return self.par[x]
    def union(self, x, y):
        self.par[self.find(x)] = self.find(y)

class Solution(object):
    def areSentencesSimilarTwo(self, words1, words2, pairs):
        if len(words1) != len(words2): return False

        index = {}
        count = itertools.count()
        dsu = DSU(2 * len(pairs))
        for pair in pairs:
            for p in pair:
                if p not in index:
                    index[p] = next(count)
            dsu.union(index[pair[0]], index[pair[1]])

        return all(w1 == w2 or
                   w1 in index and w2 in index and
                   dsu.find(index[w1]) == dsu.find(index[w2])
                   for w1, w2 in zip(words1, words2))
```

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.

Rate this article:

Previous Next

Comments: 15

Sort By

Type comment here... (Markdown is supported)

Preview Post

xinyun ★45 August 9, 2018 1:25 AM

Time complexity for naive Union-Find is incorrect.

11 Share Reply

SHOW 3 REPLIES

leetcode_deleted_user ★247 March 4, 2018 1:46 AM — Report

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

6 Share Reply

leduykhanh ★35 July 28, 2018 7:01 AM

I used union-by-rank [Weighted quick-union], this complexity improves to O(N+P)

```
class Solution {
    public boolean areSentencesSimilarTwo(String[] words1, String[] words2, Strin
```

Read More

5 Share Reply

SHOW 1 REPLY

wwan ★2 December 28, 2017 5:15 AM

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)?

2 Share Reply

SHOW 3 REPLIES

awice ★4246 January 30, 2018 4:53 AM

@IWantToPass In each DFS, we visit at most P+1 nodes. The "|V|" is not N necessarily. @wqmbisheng There are N DFS's, each DFS is O(P).

1 Share Reply

IWantToPass ★282 December 29, 2017 12:55 AM — Report

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)) ?

1 Share Reply

SHOW 1 REPLY

chinanathan ★91 June 23, 2018 11:23 PM

Hi, could you pleas illustrate the time complexity of UNION-FIND method. How can we determine find function time complexity ? are we using amortized analysis here ?? thankss so much

1 Share Reply

Aaron121138 ★0 May 1, 2018 4:08 AM

In find method. why return x would give me stack over flow? Doesn't it return when parent[x] == x? Returning x is the same as returning parent[x]?

0 Share Reply

SHOW 1 REPLY

wqmbisheng ★8 January 30, 2018 4:47 AM — Report

Hi, I agree with the time complexity of DFS method is O(N + P). what do you think? @awice

0 Share Reply

barabum ★6 2 days ago

The first approach can be optimized. You don't have to traverse the graph for each word. Instead, you can normalize word1 and word2 before comparing them. You can build a mapping Word => NormalizedWord and use it for normalization. In that case you need to traverse the graph only once to build the normalization dictionary.

0 Share Reply