734. Sentence Similarity

Easy Array Hash Table String Leetcode Link

Problem Description

The problem asks to determine if two sentences are similar. Each sentence is represented as an array of words. For example, the sentence "I am happy with leetcode" can be represented as `arr = ["I", "am", "happy", "with", "leetcode"]`. The words within each sentence are considered similar based on a provided list of word pairs `similarPairs`, where each pair `[xi, yi]` indicates that xi and yi are similar words. Two sentences are similar if they have the same number of words and each word in a position in `sentence1` is similar to the corresponding word in the same position in `sentence2`. A word is always considered similar to itself. It is important to note that the similarity between words is not transitive - just because word a is similar to word b and word b is similar to word c, it doesn't mean a is similar to c.

Intuition

To determine if the two sentences are similar, there are two main checks that need to be carried out:

1. **Check Length:** First, we need to check if both sentences have the same number of words. If they don't, we can immediately return `False` as they can't be similar with different lengths.

2. **Check Word Similarity:** Then, we check each pair of corresponding words from `sentence1` and `sentence2`. For each word pair `(a, b)` we check:

   - If `a` is the same as `b`, they are automatically similar.
   - If `a` and `b` are found as a pair in `similarPairs`, they are similar.
   - As similarity is not directed (it is bidirectional), we should also check `(b, a)` in `similarPairs`.

If all word pairs pass these similarity checks, the sentences are similar, and we return `True`. Otherwise, at the first instance of dissimilarity, the overall similarity check fails, and we return `False`.

The solution uses set comprehension to build a set `s` from `similarPairs` for quick lookup, and then utilizes the `all` function combined with list comprehension to efficiently carry out the similarity checks for each word pair from the zipped sentences.

Solution Approach

The implemented solution takes advantage of the following algorithms, data structures, and programming constructs:

- **Set Data Structure:** A set is used to store the `similarPairs` for efficient O(1) average time complexity on lookups. This is done by creating a set `s` using set comprehension to hold all given similar word pairs in both directions (e.g., (a, b) and (b, a)). The nature of a set provides constant time membership testing, which is much faster compared to a list or array for this purpose.

- **Zip Function:** The `zip` function is a built-in Python function that aggregates elements from two or more iterables (arrays, lists, etc.) and returns an iterator of tuples. In this solution, `zip` is used to create pairs of corresponding words from `sentence1` and `sentence2`.

- **List Comprehension with `all` Function:** A list comprehension is used in conjunction with the `all` function to check if all corresponding pairs of words between `sentence1` and `sentence2` are similar. The `all` function returns True if all elements of the given iterable are true (or if the iterable is empty). The list comprehension iterates through each word pair and uses a logical or `||` to evaluate if the words are identical, or if they exist as similar word pairs in the set `s`.

Here is the step-by-step approach of the solution process:

1. **Check Length Equality:**

   - Before iterating through the words, the solution checks if `sentence1` and `sentence2` have the same length using `len(sentence1) != len(sentence2)`. If they don't match, it returns `False`, thereby optimizing the solution by eliminating non-matching sentences early on.
2. **Iterative Comparison with Early Exit:**

   - The solution iterates over each pair of words `(a, b)` from the zipped sentences using list comprehension and checks:
     - If `a == b`, then the words are the same.
     - If `(a, b) in s`, the pair exists in the similarity set, so they are similar.
     - If `(b, a) in s`, since similarity is not transitive but is bidirectional, this pair is also checked in case the reverse was provided in `similarPairs`.
   - The `all` function wraps the list comprehension to ensure that every pair meets at least one of these conditions. If even one pair does not meet the similarity condition, `all` returns `False`.

By using this approach, the solution effectively breaks down the problem into small logical checks, ensuring an efficient and clear way to compare the sentences. This results in a concise and effective code:

```python
def areSentencesSimilar(self, sentence1: List[str], sentence2: List[str], similarPairs: List[List[str]]) -> bool:
    if len(sentence1) != len(sentence2):
        return False
    s = {(a, b) for a, b in similarPairs}
    return all(
        a == b or (a, b) in s or (b, a) in s for a, b in zip(sentence1, sentence2)
    )
```

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we have two sentences represented by the arrays below:

- `sentence1 = ["I", "love", "to", "code"]`
- `sentence2 = ["I", "adore", "to", "program"]`

And let's say we are also given the list of similar word pairs:

- `similarPairs = [["love", "adore"], ["code", "program"]]`

When we apply our solution, this is how it will proceed:

1. **Check Length Equality:**

   - Both `sentence1` and `sentence2` have the same number of words (4 words each), so we move on to the next step. If they had a different number of words, we would return `False` at this point.
2. **Iterative Comparison with Early Exit:**

   - We create an iterator of tuples by pairing up corresponding items from both sentences using `zip(sentence1, sentence2)`, which gives us `[("I", "I"), ("love", "adore"), ("to", "to"), ("code", "program")]`.
   - We create a set from `similarPairs` to store all given similar word pairs, including their bidirectional counterparts - that would give us a set `s` containing `[("love", "adore"), ("adore", "love"), ("code", "program"), ("program", "code")]`.
   - Using list comprehension, we check each pair:
     - First pair is `("I", "I")`: the words are identical, so this pair is similar.
     - Second pair is `("love", "adore")`: the pair exists in set `s`, so this pair is also similar.
     - Third pair is `("to", "to")`: again, the words are identical, similar by default.
     - Fourth pair is `("code", "program")`: the pair exists in set `s`, hence this pair is similar too.
   - Since every pair of words from both sentences have passed the similarity check, the `all` function will ultimately return `True`.

Using this approach with the given `sentence1`, `sentence2`, and `similarPairs`, our function `areSentencesSimilar` would return `True` because all corresponding word pairs are similar as per our rules.

Python Solution

```python
from typing import List

class Solution:
    def areSentencesSimilar(self, sentence1: List[str], sentence2: List[str], similar_pairs: List[List[str]]) -> bool:
        # First, check if both sentences have the same number of words
        # If not, they cannot be similar
        if len(sentence1) != len(sentence2):
            return False

        # Create a set of pairs that are considered similar from the given list of similar pairs
        # This allows for O(1) lookup time
        similar_set = {(first, second) for first, second in similar_pairs}

        # Check each pair of words from both sentences to see if they're the same
        # If not, check if they are considered similar from the previously created set
        # We check for both (a, b) and (b, a) to account for the order in which the similar pair might have been given
        # If all pairs of words are either the same or similar (in any order), sentences are similar
        return all(
            word1 == word2 or (word1, word2) in similar_set or (word2, word1) in similar_set
            for word1, word2 in zip(sentence1, sentence2)
        )

# Below this point would be code that uses the Solution class and its method,
# usually including test cases to demonstrate the implementation.
# Since the class itself should only contain the method definition, usage of the
# class is not included here.

```

Java Solution

```java
class Solution {
    // Method to determine whether two sentences are similar based on the provided similar word pairs.
    public boolean areSentencesSimilar(
        String[] sentence1, String[] sentence2, List<List<String>> similarPairs) {

        // If the sentences have different lengths, they cannot be similar.
        if (sentence1.length != sentence2.length) {
            return false;
        }

        // Create a set to store the unique similar word pairs.
        Set<String> similarPairSet = new HashSet<>();
        for (List<String> pair : similarPairs) {
            // Concatenate the two words with a delimiter and add to the set.
            similarPairSet.add(pair.get(0) + "." + pair.get(1));
        }

        // Iterate through the words in both sentences.
        for (int i = 0; i < sentence1.length; i++) {
            String word1 = sentence1[i];
            String word2 = sentence2[i];

            // If the current words are not the same and the pair (both combinations)
            // does not exist in the set, the sentences are not similar.
            if (!word1.equals(word2) && !similarPairSet.contains(word1 + "." + word2)
                    && !similarPairSet.contains(word2 + "." + word1)) {
                return false;
            }
        }

        // If all word pairs are similar, return true.
        return true;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_set>

class Solution {
public:
    // Function to check if two sentences are similar based on the provided similar word pairs.
    bool areSentencesSimilar(std::vector<std::string>& sentence1, std::vector<std::string>& sentence2, std::vector<std::vector<std::s
        int sentenceLength = sentence1.size(); // Storing the length of the first sentence.
        // Check if the two sentences are of different lengths; if so, they can't be similar.
        if (sentenceLength != sentence2.size()) return false;

        // Create a set to hold the pairs as concatenated strings for quick lookup.
        std::unordered_set<std::string> pairSet;
        // Insert each pair of similar words into the set as 'word1.word2'.
        for (const auto& pair : similarPairs) {
            pairSet.insert(pair[0] + "." + pair[1]);
        }

        // Check every word in both sentences for similarity.
        for (int i = 0; i < sentenceLength; ++i) {
            const std::string& word1 = sentence1[i]; // Word from the first sentence.
            const std::string& word2 = sentence2[i]; // Corresponding word from the second sentence.

            // If words are not the same and neither composition of words exists in the similarity set, return false.
            if (word1 != word2 && pairSet.find(word1 + "." + word2) == pairSet.end() && pairSet.find(word2 + "." + word1) == pairSet.
                return false;
            }
        }

        // If we make it through all word comparisons without returning false, the sentences are similar.
        return true;
    }
};

```

Typescript Solution

```typescript
type StringPair = [string, string];

// Global variables and function definitions in TypeScript

// Function to check if two sentences are similar based on the provided similar word pairs.
function areSentencesSimilar(sentence1: string[], sentence2: string[], similarPairs: StringPair[]): boolean {
    // Storing the length of the first sentence.
    const sentenceLength: number = sentence1.length;

    // Check if the two sentences are of different lengths; if so, they can't be similar.
    if (sentenceLength !== sentence2.length) return false;

    // Create a set to hold the pairs as concatenated strings for quick lookup.
    const pairSet: Set<string> = new Set();

    // Insert each pair of similar words into the set as 'word1.word2'.
    for (const pair of similarPairs) {
        pairSet.add(pair[0] + "." + pair[1]);
    }

    // Check every word in both sentences for similarity.
    for (let i = 0; i < sentenceLength; i++) {
        const word1: string = sentence1[i]; // Word from the first sentence.
        const word2: string = sentence2[i]; // Corresponding word from the second sentence.

        // If words are not the same and neither composition of words exists in the similarity set, return false.
        if (word1 !== word2 && !pairSet.has(word1 + "." + word2) && !pairSet.has(word2 + "." + word1)) {
            return false;
        }
    }

    // If we make it through all word comparisons without returning false, the sentences are similar.
    return true;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be broken down as follows:

1. Checking if `len(sentence1)` is equal to `len(sentence2)` takes `O(1)` time, as it is simply comparing two integers.
2. Creating a set `s` from the `similarPairs` takes `O(p)` time, where `p` is the number of similar pairs.
3. The list comprehension iterates over each pair of words from `sentence1` and `sentence2`, which occurs in `O(n)` time, where `n` is the number of words in each sentence (since we've previously established the sentences are of equal length to reach this point).
4. Each comparison within the list comprehension is `O(1)` because checking equality of strings and checking for existence in a set are both constant-time operations.

Thus, the time complexity is `O(n + p)`, where `n` is the length of the sentences and `p` is the number of similar word pairs.

Space Complexity

The space complexity can be analyzed as follows:

1. The additional set `s` which stores the similar word pairs requires `O(p)` space, where `p` is the number of similar pairs.
2. The space for input variables `sentence1`, `sentence2`, and `similarPairs` is not counted as extra space as per convention since they are part of the input.

As a result, the auxiliary space complexity of the code is `O(p)`.