Report: Word-Pair Path Finder

1 Introduction

1.1 Problem description

The problem we are solving is finding the shortest path between two words in a dictionary, with the constraint that each consecutive pair of words in the path can only differ by a single character. To tackle this problem, we implemented a non-optimized algorithm that generates an adjacency list representation of the word graph and uses Breadth-First Search (BFS) to find the shortest path between two words. We also use Binary Search to check if the input words exist in the dictionary.

2 Code Organization and Classes

The code is organized into two main classes: `WordGraph` and `WordGraphProcessor`.

2.1 WordGraph

The `WordGraph` class is responsible for representing the word graph, where nodes represent words and edges represent connections between words differing by a single character. The class provides methods for creating the adjacency list, adding edges, and accessing neighbors of a given node.

2.2 WordGraphProcessor

The `WordGraphProcessor` class is responsible for processing the `WordGraph` to find the shortest path between two words. It implements the Breadth-First Search algorithm and provides a method for checking if a given word exists in the dictionary using Binary Search.

3 Runtime Analysis

3.1 WordGraph Runtime Analysis

Let's denote n as the number of words with the same length and l as the length of the words.

- **Generating new words:** $\mathcal{O}(n \cdot l \cdot 26) = \mathcal{O}(n \cdot l)$
- **Dictionary lookups:** $\mathcal{O}(n \cdot l \cdot 26) = \mathcal{O}(n \cdot l)$
- **Index checks and adjacency list updates:** $\mathcal{O}(n \cdot l \cdot 26) = \mathcal{O}(n \cdot l)$

Total runtime complexity for the `WordGraph` class: $\mathcal{O}(n \cdot l) + \mathcal{O}(n \cdot l) + \mathcal{O}(n \cdot l) = \mathcal{O}(n \cdot l)$

3.2 WordPair Processor Runtime Analysis

The runtime complexity of the `WordGraphProcessor` class, which is responsible for processing the `WordGraph` to find the shortest path between two words, is dominated by the Breadth-First Search (BFS) algorithm. The BFS algorithm has a runtime complexity of $\mathcal{O}(V + E)$, where V is the number of vertices (words) in the graph, and E is the number of edges between the vertices.

Since the WordGraph is likely to be sparse in practice, the number of edges, E, would be less than the number of possible edges in a fully connected graph. As a result, the BFS runtime complexity would be closer to $\mathcal{O}(n + m)$, where n is the number of nodes (words), and m is the number of edges.

In the worst-case scenario with a fully connected graph, the runtime complexity would be $\mathcal{O}(n^2)$. However, this situation is improbable and not representative of typical inputs.

3.3 Overall Runtime Complexity

The overall runtime complexity of the program is determined by the combination of the runtime complexities of the individual components:

- **WordGraph class:** $\mathcal{O}(n \cdot l)$
- **WordGraphProcessor class (BFS):** $\mathcal{O}(n + m)$
- **Binary Search:** $\mathcal{O}(\log n)$

Since the WordGraph class and the BFS algorithm have higher runtime complexities than the Binary Search algorithm, the overall runtime complexity of the program is mainly influenced by the WordGraph class and the WordGraphProcessor class (BFS algorithm). Therefore, the overall runtime complexity of the program is $\mathcal{O}(n \cdot l + n + m)$.

4 Conclusion

In this report, we analyzed the design and runtime complexity of a program that determines the minimum number of steps required to transform one word into another, given a list of word pairs and a list of unique words. We introduced the `WordGraph` class for adjacency list creation and the `WordGraphProcessor` class for finding the shortest path between two words using the Breadth-First Search (BFS) algorithm.

We derived the runtime complexity of the `WordGraph` class to be $\mathcal{O}(n \cdot l)$ and the runtime complexity of the `WordGraphProcessor` class to be $\mathcal{O}(n + m)$, where n is the number of nodes (words), l is the length of the words, and m is the number of edges. The overall runtime complexity of the program is $\mathcal{O}(n \cdot l + n + m)$.

This analysis provides insights into the efficiency and scalability of the program, which can be valuable when working with large datasets. The results indicate that the program performs well with sparse graphs and can handle inputs of varying sizes, although the performance may degrade as the number of words and edges increases.

A Algorithm Pseudocode

A.1 Adjacency List Generation

Algorithm 1 Adjacency List Generation

```
function CreateWordLengthGraphs(length_word_dict, length_word_lookup)
    Initialize an empty dictionary, word_length_graphs
    for each word_length, words_with_same_length in length_word_dict.items() do
        Initialize an empty list, adjacency_list
        for each word1 in words_with_same_length do
            Initialize an empty list, neighbors
            for each new_word generated by GenerateNewWords(word1) do
                if new_word in length_word_lookup[word_length] then
                    word2_index ← length_word_lookup[word_length][new_word]
                    word2 ← words_with_same_length[word2_index]
                    if IsAdjacent(word1, word2) then
                        neighbors.append(word2_index)
                    end if
                end if
            end for
            adjacency_list.append(neighbors)
        end for
        word_length_graphs[word_length] ← adjacency_list
    end for
    return word_length_graphs
end function
```