Word ladders report

Adam Tovatt

April 13, 2025

1 Results

All my solutions were marked as correct by the provided `check_solution.sh` script.

The input file `5large1.in` takes the longest to process, with a runtime of approximately 12 seconds. Interestingly, `6large2.in`, which is of similar size, only takes around 6 seconds.

`5large1.in` seems to have 181554 edges. `6large2.in` only has 5334 edges.

This difference is likely due to the amount of edges in the graph from `5large1.in`. A larger amount of edges lead to more edge-checking during graph construction, which takes more time. Most time that is used by the code is used in the graph construction so if that takes more time the whole program will also take more time to run.

2 Implementation details

The solution is structured as a graph where each node represents a five-letter word. There is an edge from word u to word v if all of the last four letters in u appear in v, with at least the same number of each letter.

Each node is represented by a `Node` class, containing its word and a list of neighboring nodes. All nodes are stored in a `Dictionary<string, Node>` that is a class level field in the Graph-class. Edges are constructed during initialization by iterating over all pairs of words and checking the inclusion rule using character count dictionaries.

Character counts are implemented using `Dictionary<char, int>` and encapsulated in string extension methods for reusability and clarity.

For shortest path queries, I use a standard breadth-first search (BFS), ensuring a time complexity of $\mathcal{O}(n + m)$ per query, where n is the number of nodes and m is the number of edges. A custom `SearchState` struct is used to track BFS state cleanly.

Overall time complexity:

- Graph construction: $\mathcal{O}(n^2 \cdot k)$, where n is the number of words and $k = 4$ is the suffix length. The constant factor is small due to the limited alphabet.
- Queries: Each query is answered in $\mathcal{O}(n + m)$ using BFS.