COMPSCI2026 Algorithmics

Assessed Exercise – Status and Implementation Reports

Esmeralda Jardine
2391491

April 11, 2025

Status report

The program for both wordladder and Dijkstra's algorithm compile and appear to run correctly uin the test cases provided.

Implementation report

(a) Each word in the dictionary is a vertex, an edge exists between two vertices if their words differ by exactly one letter. The graph is represented using adjacency lists for storage and traversal.

- **Graph Initialisation:** The dictionary file is read and each word is added as a vertex in the graph. Each word is compared with every other word in the dictionary and if two words differ by exactly one letter, an edge is added between them in the form of adding them to each others in the adjacency list.
- **Shortest Path Search:** A Breadth-First Search algorithm is used to determine the shortest word ladder. BFS was used as it is ideal for unweighted graphs as it guarantees the shortest path in terms of the number of edges. The algorithm starts from the vertex belonging to the start word and marks it as visited, then adds it to a queue. It continues traversing until the end word is reached or all possibilities are used up.
- **Path Construction:** Each vertex stores its predecessor during the BFS. The path is reconstructed by backtracking from the end word to the start word using the predecessors. If a path is not possible, then the program reports that no ladder is possible.
- **Efficiency Choices:**

    * A HashMap is used to map words to their indices in the graph, allowing for a O(1) lookup for vertex indices.
    * The adjacency list representation minimizes memory usage compared to an adjacency matrix because it only stores edges that exist.
    * BFS is efficient because it explores all vertices at the current depth level before moving on. This means that the shortest path is found without unnecessary computations.

(b) The graph is constructed in the same way as in part (a). The only difference is that the graph is weighted.

- **Graph Initialisation:** The graph is constructed in the same way as in part (a). The only difference is that the graph is weighted. The weight is stored in the adjacency list of the corresponding vertex.
- **Dijkstra's Algorithm:** Dijkstra's algorithm is used to find the shortest path in the weighted graph. It starts with an array of distances initialized to "infinity" for all vertices except the starting vertex - that is set to 0. A (linked list) queue is used to choose the vertex with the smallest distance that has not been visited yet. For every selected vertex, its adjacent vertices are updated if a shorter path is found going through the current vertex.
- **Efficiency Choices:**
    * The adjacency list representation minimizes memory usage by storing only existing edges.
    * A helper function calculates the edge weight efficiently by determining the absolute difference in positions of the mismatched letter.
    * The algorithm avoids repeating calculations by tracking the visited vertices and skipping them in subsequent iterations.
- **Debugging issues:** Initially, this algorithm was only working for some of the test cases which only involved a couple letter transformations. For larger transformations, the total path weight seemed too low. It was easy to trace that the problem was in the way the edge weights were calculated. My Implementation was using signed integers to calculate the difference rather than unsigned integers. This caused the algorithm to return negative weights for some edges, making the total path weights lower than expected.

Empirical results

Word Ladder Times:

- `print` → `paint`: 91 ms, path length 1

- `forty` → `fifty`: 87 ms, path length 4
- `cheat` → `solve`: 92 ms, path length 13
- `worry` → `happy`: 74 ms, path length 12
- `smile` → `frown`: 107 ms, path length 7
- `small` → `large`: 91 ms, path length 16
- `black` → `white`: 95 ms, path length 8
- `greed` → `money`: 85 ms, no path

Dijkstra's Times:

The execution times for Dijkstra's algorithm are slightly higher than wordladder because of the added overhead of maintaining and updating distances and weights.

- `blare` → `blase`: 115 ms, weight 1
- `blond` → `blood`: 131 ms, weight 1
- `allow` → `alloy`: 125 ms, weight 2
- `cheat` → `solve`: 120 ms, weight 96
- `worry` → `happy`: 113 ms, no path
- `print` → `paint`: 113 ms, weight 17
- `small` → `large`: 112 ms, weight 11
- `black` → `white`: 117 ms, weight 56
- `greed` → `money`: 113 ms, no path