Assignment 9

Course: COMPSCI 2C03
Date: 12/04/2025
Author: Sulagna Nandi
MacID: nandis
Student Number: 400575118
Professor: Dr. Vincent Maccio

Problem 1

P1.1

To model the given situation as a graph problem, I treat every road crossing as a node in a weighted graph, and every direct road segment between two consecutive crossings as an edge. Each edge carries a weight corresponding to the travel duration along that section of road. Because traffic danger is highest at crossings, the goal is not only to find routes of minimal total cost but, among those equally minimal-cost routes, to prefer the ones that pass through the fewest crossings. In graph-theoretic terms, the problem reduces to finding, from a starting crossing A, all shortest paths with respect to the sum of edge weights, and then selecting among them those with the minimum number of edges. This means our target is a lexicographically minimal path: first minimizing total weight, and then minimizing the hop count.

P1.2

To find such a best route, I define the algorithm FindBestRoute, which is a modified version of Dijkstra's algorithm. Instead of recording only the shortest distance to every node, the algorithm keeps a pair (d, k) for each node, where d is the minimal total cost discovered so far and k is the number of crossings (edges) on that route. During relaxation, a new path to a node v is considered superior if it has a strictly smaller cost d, or if it has the same cost but uses fewer edges k. The algorithm proceeds as follows:

Algorithm 1 FindBestRoute(G, A)

For each node v, initialize dist$[v] = (\infty, \infty)$.
Set dist$[A] = (0, 0)$.
Insert $(0, 0, A)$ into a priority queue ordered lexicographically.
while the queue is not empty **do**
    Extract the node u with smallest pair (d, k).
    for each neighbor v of u with edge weight w **do**
        Let $(d', k') = (d + w, k + 1)$.
        if $(d', k') <_{\text{lex}}$ dist$[v]$ **then**
            Update dist$[v] = (d', k')$.
            Insert (d', k', v) into the queue.
        end if
    end for
end while
return dist

This produces, for each crossing, the best safe shortest route according to the desired lexicographic ordering.

P1.3

For this algorithm, I use an adjacency list representation, since the graph represents a realistic road network where most nodes tend to have only a small number of neighbors. With adjacency lists, each relaxation step inspects only the edges that actually exist, so the running time follows the classical Dijkstra bound of $O((V + E) \log V)$ when using a binary heap and lexicographic keys. Because the only modification is the tuple stored in the priority queue, the asymptotic cost remains the same.

P1.4

If instead the graph were stored as an adjacency matrix, enumerating the neighbors of a node would require scanning an entire row of length V every time, regardless of how sparse the graph is. In the worst case, this leads to a total running time of $O(V^2 \log V)$, because each extraction from the priority queue still costs $\log V$, but now each relaxation step triggers V neighbor checks. This represents the typical drop in efficiency when using adjacency matrices for large sparse road networks.

Problem 2

P2.1

To describe the energy-efficiency problem for electric cars as a graph problem, I again treat every crossing or relevant point as a node and each direct road segment as a directed edge. This time, each edge is assigned a weight representing energy consumption, which may even be negative to capture regenerative braking on downhill segments. The task is to compute the most energy-efficient path from a starting point to all other nodes, which means determining the path with minimum total energy cost, even if that cost involves some negative edges. The central difficulty is that downhill edges can yield energy, so a naïve shortest-path algorithm cannot be assumed safe.

P2.2

It is important to consider whether the system would allow a car to drive in circles indefinitely while gaining energy. Mathematically, this would correspond to the existence of a directed cycle whose total weight is negative. If such a cycle exists, one could loop around it repeatedly and accumulate unlimited energy. Therefore, it is essential to determine whether negative cycles are present. In a physically realistic road system, negative cycles are unlikely because any gain from descending must eventually be offset by climbing, but the graph model itself does allow for the possibility. Thus, the graph-theoretic answer is that if a negative cycle is reachable from the starting point, then unbounded energy gain is possible; otherwise, it is not.

P2.3

To find the most energy-efficient routes, including dealing properly with possible negative weights, I define the algorithm FindEfficientRoutes, which is simply the Bellman–Ford algorithm enhanced to detect negative cycles. The algorithm proceeds by initializing all distances to infinity, setting the source's distance to 0, and then performing $|V| - 1$ rounds of edge relaxation. After this, it performs a final pass to check whether any edge can still be relaxed; if so, a negative cycle must exist.

Algorithm 2 FindEfficientRoutes(G, A)

```
For each node v, set dist[v] = ∞.
Set dist[A] = 0.
for i = 1 to |V| − 1 do
    for each edge (u, v) with weight w do
        if dist[u] + w < dist[v] then
            Update dist[v] = dist[u] + w.
        end if
    end for
end for
for each edge (u, v) with weight w do
    if dist[u] + w < dist[v] then
        return "Negative cycle reachable from A"
    end if
end for
return dist
```

This algorithm properly handles negative weights and reports if the problem is ill-defined due to the possibility of infinite energy gain.

P2.4

In implementing this algorithm, I use an adjacency list representation, because Bellman–Ford iterates over all edges in the graph, and enumerating them is far more efficient with adjacency lists. Each full relaxation pass costs $O(E)$ time, and the algorithm performs $V - 1$ such passes, leading to a total running time of $O(VE)$.

P2.5

If the graph were instead stored as an adjacency matrix, each edge relaxation step would require checking every possible ordered pair of nodes to determine whether an edge exists, which means that each pass costs $O(V^2)$. Because Bellman–Ford requires V passes, the worst-case running time becomes $O(V^3)$ when using an adjacency matrix representation. This is a significant overhead compared to adjacency lists, especially for sparse road networks where E is much smaller than V^2.