PathFinding

Overall Description

To represent the map, we created a custom adjacency list class named `Graph`. To represent an edge between two nodes, we created an `Edge` class that stores the **start** node, **end** node and the **weight**. The `Edge` can also be used to represent the distance from an **origin** node to a specific node.

Task A: Implement the Dijkstra' Algorithm

We implemented the algorithm using a min-heap `PriorityQueue` (**Q**) to keep track of which nodes the algorithm should visit next. At start, it initializes the distance estimate for the origin to be **0** and all other nodes to be "infinity" (we used `Integer.MAX_VALUE`), then it adds the distance estimate for origin to the **Q**. Then while the **Q** is not empty **and** the destination is not found, it dequeues one node and if that node is not in the `settledNodes` list, the algorithm adds it to the list and updates the distance estimates for its neighbors. Once the destination node is found, the loop stops and the algorithm will backtrace from the destination to the origin and add all nodes along the way to the `shortestPath` list.

Task B: Handle Terrain Costs

The terrain cost information is stored in each `Coordinate` (node). For example, node **A** has a terrain cost of **2** and node **B 3**. The weight of `Edge` from **A** to **B** is then **3**, because that's the cost of entering node **B**. There is no modification needed to the algorithm mentioned above as before all the weights were defaulted to **1**.

Task C: Consider Multiple Origins and Multiple Destinations

For multiple origins, we slightly modified the algorithm so that it initializes the distance estimates for **all** the origins to be **0** and adds them to the **Q**. After that it runs the same steps as Task A. Once a destination is found, the loops stops, and that destination is part of the shortest path and the algorithm just backtraces from the found destination back to one of the origins.

Task D: Go Through Waypoints

With waypoints, this problem becomes **NP-Hard** and we have to try all possible paths to find the shortest. We used the **Heap's Algorithm** to recursively generate all permutations of the waypoints and each one represents a **visiting order** of those waypoints. Whenever a new **order** is found we run the same algorithm we implemented to find the shortest path for each path segment, i.e., from one origin to the 1st waypoint, then from the 1st waypoint to the 2nd waypoint and all the way to the destination. Then we sum up the costs of those segments to get the total cost of the current path and compare it with the previous path cost. If the new cost is lower, make the current path the new candidate, otherwise, discard the path. Since the time complexity of this approach is `O(n!)`, where `n` is the number of waypoints, the algorithm can only find the optimal solution within a reasonable time, for small number of waypoints.