COP3530 Final Project

Team Name: Group 10

Team Members + GitHub Username:

- Junhao Li – JunhaoLiXD
- Yuan Yao – diegoyao13
- Xinye Li – carali6 (you may also see “carali66” in Contributors, which is another account of mine)

Link to GitHub repo: https://github.com/JunhaoLiXD/FinalProject_COP3530

Link to Video demo: https://youtu.be/ItizqHnUP2o

PART I: Extended and Refined Proposal

Nowadays, we use navigation apps almost every day. Navigation apps always give us the most efficient route from the starting point to our destination in real-time. Finding the most efficient route is crucial for reducing travel time and fuel costs. In this project, we are trying to explore the algorithm of finding the most efficient route behind navigation apps by comparing Dijkstra’s algorithm and Bellman-Ford algorithm in the context of real-world navigation problems to determine which one is better suited for different scenarios. Comparing different shortest path algorithms is crucial because each algorithm has unique strengths and weaknesses depending on graph structure, edge weights, efficiency requirements, and real-world constraints.

The entire project is implemented in C++ using Qt framework for UI development, and CMake for build configuration. The following libraries and modules are used:

Qt Core, GUI, Widgets modules
CMake
C++ Standard library (STL)

We compare Dijkstra’s algorithm and Bellman-Ford algorithm for finding the shortest path. For both algorithms, the real-life problem of finding the shortest path between the starting point and the destination can be simplified to a graph with vertices and edges – a vertex representing a location, and an edge representing the road between locations.

- **Dijkstra's Algorithm:** to find the shortest distance between the source node and the destination node, it iteratively picks unvisited node with the minimum distance from the source node, and update the distance of its neighboring nodes if the new path gives a shorter distance than current distance. We use min-heap priority queue to implement Dijkstra's Algorithm to efficiently select unvisited nodes with the minimum distance from the source node each time.
- **Bellman-Ford Algorithm:** to find the shortest paths from a single source node to all other nodes in a weighted graph, it repeatedly relaxes all edges for V - 1 iterations, where V is the number of vertices.

Instead of downloading a public dataset, we choose to randomly generate data. Two types of directed graphs with 100,000 data points were generated: one general directed graph used for both algorithms, and one directed acyclic graph (DAG) used specifically for testing the Bellman-Ford algorithm. This is because each algorithm has unique constraints on graph structure. Dijkstra's Algorithm does not work for negative weights, and Bellman-Ford Algorithm does not work on graphs containing negative weight cycles.

Working as a team, we are each responsible for our own content. Xinye Li is responsible for data generation and implementation of Dijkstra's Algorithm. Yuan Yao is responsible for data generation and implementation of the Bellman-Ford Algorithm. Junhao Li is responsible for graph visualization of both algorithms.

PART II: ANALYSIS

We didn't make any changes after making the proposal.

Time Complexity Analysis

① vector<vector<Edge>> generateDirectedGraph(const int num_nodes, const int avg_degree, const int min_weight, const int max_weight)

- **Time Complexity: O(D * N)**
    - N: number of nodes, D: average number of outgoing edges per node
- **Analysis**

The algorithm assigns D outgoing edges for each node, so it runs in O(D * N) on average. If D is close to N, the graph becomes dense and many edge attempts are rejected due to duplicates, which can increase complexity to O(N^2). In our project, since D is usually much smaller than N, the expected time remains O(D * N).

② vector<vector<Edge>> generateDAGraph(int num_nodes, int avg_degree, int min_weight, int max_weight)

- **Time Complexity: O(D * N)**
    - N: number of nodes, D: average number of outgoing edges per node
- **Analysis**
    - For num_nodes <= 10: The algorithm generates all valid DAG edges (from < to), shuffles them, and selects a subset. This takes about O(N^2) time because there can be up to N^2 possible edges.
    - For num_nodes > 10: The algorithm first ensures each node has at least one incoming edge, which takes O(N) time. Then, it adds around D * N edges to reach the target average degree D. This runs in O(D * N) time. Only when D approaches N, the complexity approaches O(N^2) due to

increased duplicate edge checks. In our project, D is generally much smaller than N, so the expected runtime remains closer to O(D * N).

③ pair<vector<int>, int> Dijkstra(const int num_nodes, const int source, const int target, const vector<vector<Edge>>& adjacencyList)

- **Time Complexity: O((V+E) * log V)**
  - V = number of nodes, E = number of edges
- **Analysis**
  - Extracting the node with minimum value from the min heap takes O(log V), and every node needed to be extracted from the min heap, so extracting all nodes takes O(V * log V).
  - The process of relaxing edges needs to update the distance of every edge in the worst case, which takes O(log V) for each update because it inserts to the min heap. To update all edges in the worst case, it takes O(E * log V) in total.
  - So, the worst-case time complexity when using min-heap priority queue for Dijkstra's algorithm is O((V+E) * log V).

④ pair<vector<int>, int> BellmanFord(const int num_nodes, const int source, const int target, const vector<vector<Edge>>& adjacencyList)

- **Time Complexity: O(V * E)**
  - V = number of nodes, E = number of edges
- **Analysis**
  - The Bellman-Ford algorithm computes the shortest path from a single source node by iteratively relaxing all edges, In the worst case, it performs (V -1) full iterations.
  - In each iteration, the algorithm traverses every edge in the graph. For E edges, one full pass of edge relaxation takes O(E) time. Therefore, (V - 1) iterations result in a total time complexity of O(V * E).

PART III: REFLECTION

As a group, the overall experience for the project is great. All group members are responsible for their part and there are a lot of group discussions. Every group member learns a lot through the project. The whole process goes very smoothly, but if we were to start once again as a group, we may want to spend more time on trying more algorithms that could be applied to compute the shortest path between the starting point and the destination and do more comprehensive comparisons of their logic, constraints, and performance.

Junhao Li: I learned how to use Qt to build a simple and clear user interface for our project. During the process, I got more familiar with how Qt layouts work and how to connect buttons and actions. It also helped me understand how to link the UI with the backend code.

Yuan Yao: I learned a new algorithm to compute the shortest path: the Bellman-Ford Algorithm, and practiced its implementation, which helped me gain a deeper understanding of this essential problem. I have also gained insights into how different ways solving the same problem may have different priorities and pros.

Xinye Li: I have a better understanding of min heap by implementing Dijkstra’s algorithm with min heap to choose the node with shortest distance. I also learned about how the similar idea of Dijkstra’s Algorithm can be transferable to the Bellman-Ford Algorithm.

PART IV: REFERENCES

[1] Geeksforgeeks. (2025, Apr. 11). *How to find Shortest Paths from Source to all Vertices using Dijkstra’s Algorithm* [Online]. Available: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7

[2] Geeksforgeeks. (2025, Apr. 15). *Bellman–Ford Algorithm* [Online]. Available: https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23