Project 3 Report

Administrative

- Team Name: **J Cubed Flights™**
- Team Members: **Jack Wang, Jan Torruellas, Jason Li**
- GitHub URL: https://github.com/J-Cubed-Flights/FlightPathQuery
- Link to Video: https://www.youtube.com/watch?v=eL4GfqAiDEA

Extended and Refined Proposal [Suggested 2 Pages]

- Problem: What problem are we trying to solve?
  - To find the fastest flight route between two airports while having a reasonable and quick search time. We used a two hour time penalty for each layover. In our project, we were able to solve this problem for US domestic flights because that was what our data contained, but given the proper data, this program we created should work for international flights as well.
- Motivation: Why is this a problem?
  - People want to get to their destination as quickly as possible, and layovers are generally undesirable, so we wanted to use algorithms to find the shortest possible travel time (with layover penalties) between two airports. Although this flight path probably wouldn't be the cheapest, it would give users a rough idea of what flight path they might want to look at when booking tickets.
- Features implemented
  - Two types of test cases:
    - Test Type 1: Small set of test cases that we find the proper flight times from google searching. If the resulting flight returned by the program is +/- an acceptable time difference, then the test case will be considered passed.
    - Test Type 2: Large batch of randomly generated start and end destinations. This is to ensure that our algorithms are able to find a flight path between a large number of random airports.
  - A print flights function print all flights going out of a specific airport (this helps us to view the data in the graph and verify the path calculated is indeed the shortest path)
  - The following are buttons used on the GUI and their functions:
    - A function to get all airport names (which are then displayed on the GUI) .
    - A search button,- in the GUI to search for the flight path from a chosen origin and destination
    - A clear button - clears the information in the results section, clears the elapsed time, and clears the information typed into the start and destination boxes.
    - "Options" in the top left of the screen.
      - "Choose Data Folder" option - Allows you to select the directory from which the data is read from (if it the data was placed in a different folder).
      - A reset option - resets the Floyd Warshall Map that is repeatedly used after each computation.
  - The GUI can be viewed in the youtube video.
- Description of data
  - The flight data we used is from the following link:
    - https://www.kaggle.com/datasets/vishwanathmuthuraman/domestic-flight-data
    - It contains over 200k different US domestic flights, each with multiple data attributes, so we will be well above the 100k data points requirement
    - Data attributes: Origin Airport, Destination Airport, Flight Time

    - We also have a dataset of US airport abbreviations with their corresponding Airport Full Name, the link is here:
        - https://www.kaggle.com/datasets/aravindram11/list-of-us-airports
        - Data Attributes (That are used): IATA Codes, Airport Names
- Tools/Languages/APIs/Libraries used
    - Programming language: C++
    - Tools: CLion IDE, Visual Studio IDE, GitHub Desktop (for version control)
    - Frameworks: .NET Framework
    - Libraries: C++ Standard Library, CLI (Common Language Infrastructure)
- Algorithms implemented
    - Dijkstra's Algorithm (used both the standard version and a priority queue version)
    - Floyd-Warshall Algorithm
- Additional Data Structures/Algorithms used
    - QuickSort - used to sort the Airport Names which are displayed in the GUI for easy reference of the existing Airports.
    - Several STL Data Structures: vector, unordered_map, priority_queue, pair
    - Adjacency List Graph - This was used to store the weighted & directed graph that was generated from the data.
- Distribution of Responsibility and Roles:
    - Jason: cleaned source data, created the Floyd Warshall Algorithm function, built and integrated the GUI, created a function to parse the CSV data files
    - Jan: cleaned source data, created the Djikstra's algorithm function, created the Path Object (FlightPath Class) for returning the shortest flight path, video editor
    - Jack: created the Graph Nodes (Airport Class) and Graph Object (DirectedGraph Class), repo organization and structure
    - Everyone: Bug fixes, maintain code quality, commenting code, and group project deliverables (i.e. this report), creation of the video

Analysis [Suggested 1.5 Pages]

- Any changes the group made after the proposal? The rationale behind the changes.
    - Switched the BFS algorithm to Floyd-Warshall due to an oversight for BFS, since it does not work on weighted graphs. The Floyd-Warshall algorithm was a great choice to use alongside Djikstra's because they both are algorithms designed to handle finding the shortest path in a weighted graph.
    - A GUI was added to enhance the visuals part of our project rather than just having CLI. This makes it easier for users to use the program compared to entering values into a command line and the output displayed is much cleaner.
- Big O worst case time complexity analysis of the major functions/features you implemented
    - **Parsing time complexity:** O(N) time; N = number of flights added to the graph (or number of lines in the flight file).
        - The function had to iterate through each line of data to parse it from the file.
        - Since adding a flight to the graph takes constant time, the overall time complexity was O(N).
    - **Quick sort time complexity:** In the worst case, this would take O(V^2) time (although, the average and best case are both O(Vlog(V)) time). V = number of vertices in the graph.

        - This worst case arises if each pivot just happens to be the smallest or largest value IATA code. Thus, the partitioning at each layer would take the following number of comparisons: V-1, V-2… 1. This is equal to O(V^2).
    - Dijkstra's Algorithm (Standard): $O(V^2)$. Note that unordered map operation searching operation takes O(1) time every time it is called.
        - First, it takes O(V) time to initialize the predecessor, shortest time, and visited maps.
        - Then there is a while loop that will operate V times (once for each vertex). Within each iteration it performs two operations:
            - Search the unvisited set to find the vertex with the shortest path. (this looks through V vertices for the first iteration, then V-1, V-2, …, and 1 vertex the last time). This means that this section of the while loop will take O(V^2) time to execute overall, when considering the total time of each iteration of the while loop for this operation.
            - Perform edge relaxation for all outbound edges of current vertex: This operation could be summarized as O(E) - since it will go through every edge over all the while-loop iterations.
        - The last step is to construct the path, which takes at most V operations (if the shortest path somehow traverses all vertices.
        - Thus, the overall time complexity can be summarized as O(V + V^2 + E + V), which is O(V^2). This is because we know (E is less than or equal to V^2 at most).
    - Dijkstra's Algorithm (with MinHeap): O(E*log(E)) in the worst case (**Note: Our solution was not the most optimal O(E*Log(V)) solution since we used the STD priority queue, which does not have a function to update the priority of existing elements in the heap, unlike with fibonacci heap**). E = number of total edges, V = number of vertices.
        - First, it takes O(V) time to initialize the costs vector, predecessor vector, and visited vector.
        - There is a while loop that will run at most E times (if all edges somehow were added to the minHeap in the worse case, and in the best case, it will contain up to V elements). This has two major (non-constant) operations in each iteration:
            - Popping the top element from the minHeap, which takes log(E) time at worst (if all the edges are in the heap). Since this portion occurs up to E times (for the E iterations), this comes out to be O(E*log(E))
            - Edge Relaxation of all edges for the selected vertex. When considering all the existing edges that are examined throughout all the iterations of the while loop, this occurs E times, and for each edge, it will add a new value to the minHeap if the path cost is less than the previously calculated path cost. Thus, this comes out to be O(E*log(E)), assuming each edge is added to the minHeap
                - Note: In actually, the above two operations will be closer to O(V*log(V)) and O(E*log(V)) respectively. This is because relatively few duplicate vertices are added for flights (since fewer layovers tend to result in shorter travel time).
        - Last, it takes O(V) time (in the worst case that the shorted path somehow traverses over all vertices) to construct the path.
        - Overall this can be summarized to O(V + E*log(E) + E*log(E) + V), which is equal to O(E*log(E)) time.
    - Floyd-Warshall Algorithm: $O(V^3)$. V = number of vertices.
        - Initialization the adjacency matrix that is used by Floyd-Warshall Algorithm takes O(V^2) time (making the V-by-V matrix).

        - Executing the Floyd-Warshall Algorithm has 3 nested loops iterating through all vertices each, which gives a time complexity of O(V^3).
        - Thus, it has an overall time complexity of O(V^3).
            - Note: Each call after the first takes O(V) time in the worst case (if the shortest path traverses all the vertices for some reason), since the matrix that is generated by the first call of the algorithm can be used afterwards.

Reflection [Suggested 1-1.5 Page]

- As a group, how was the overall experience for the project?
    - Overall we had a very wholesome, fun, and collaborative team for this project. Any problems were resolved fairly quickly and it was a great learning experience for everyone involved. Everyone was open to each other's ideas and suggestions, which allowed us to solve problems quickly. The group was very responsive which led to a good open channel for communication.
- Did you have any challenges? If so, describe.
    - Due to us planning the project out in detail before beginning the project, there weren't too many large-scale problems (and actually not too many bugs either) during actual development. The major changes were already mentioned earlier, which were to implement a user-friendly GUI and changing the BFS algorithm to Floyd-Warshall.
    - There were some problems that stumped us for a bit related to memory access L-value errors, but a change in how a few member variables were stored and called upon fixed this issue.
- If you were to start once again as a group, any changes you would make to the project and/or workflow?
    - Besides the GUI and algorithm change, one change would be to make the GitHub repository public to start out with so that we can make use of the branch protection rules (since we don't have GitHub Enterprise). This way we can protect the main branch from changes and only work on non-main branches. Since it was just the 3 of us we were lucky to not run into version mismatch issues due to not using branches, but if we were to start over, branches would be a crucial and necessary change. We messaged each other whenever commits were made, which likely prevented version mismatch from occurring.
- Comment on what each of the members learned through this process.
    - Jason: This was the first time that I have worked on a collaborative project and used GitHub for version control, so this was a new experience overall for me. Additionally, this was an excellent learning experience for me in working with developing GUI's using C++, since it was my first time and I spent quite a bit of time learning how to use the .NET framework.
    - Jan: My last experience with a group project was not the greatest as I was put into a group who remained absent for all but the few days leading up to due date. This led to me having to essentially do everything on my own. I have learned a great deal about the dynamic of team work when it comes to projects. I learned a bit about various C++ implementations and thanks to the amazing partners I had, learned a different style to code in. I also learned a lot about how expansive the C++ library really is and the usefulness of having a GitHub repository. Biggest of all, if ever stumped, not to hesitate asking for help. Overall, practice and a look at other ways to implement essentially the same code definitely helps for future reference.
    - Jack: I was project manager for a large Python project in the past, so I was familiar with version control and project management already, but not as knowledgeable with the C++ language as I am with Python. Through this project, I learned a fair bit about code organization and styling in C++, and also became more familiar with memory-related problems, as well as the C++ language itself. Also, I haven't worked on many group projects in C++ before, so this was a good

chance for me to see more of my peers' coding styles and learn how to better collaborate in C++.

References

- Data Structures and Algorithm Analysis in C++
- https://www.kaggle.com/datasets/vishwanathmuthuraman/domestic-flight-data
- https://www.kaggle.com/datasets/aravindram11/list-of-us-airports
- https://en.wikipedia.org/wiki/Graph_(abstract_data_type)
- https://en.wikipedia.org/wiki/Hash_table
- https://en.wikipedia.org/wiki/Directed_graph
- https://en.wikipedia.org/wiki/Dijkstra's_algorithm
- https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm
- https://www.geeksforgeeks.org/finding-shortest-path-between-any-two-nodes-using-floyd-warshall-algorithm/
- https://www.simplilearn.com/tutorials/cpp-tutorial/cpp-gui