Technical Report: Final Project

EECE 2560: Fundamentals of Engineering Algorithms

Optimizing City-to-City Transportation with Maximum Flow Algorithms

Kennedy Scheimreif, Anna Valades, Matt Geisel

Department of Electrical and Computer Engineering

Northeastern University

scheinmreif.k@northeastern.edu, geisel.m@northeastern.edu, valades.a@northeastern.edu

December 4th, 2024

Contents

1 Project Scope

The objective of this project is to develop a custom system that calculates the maximum number of people that can be transported from one city to another with various transportation methods. It will analyze different routes that have set capacities and optimize the flow for maximum possible transport from the source city to the destination city. It will also calculate the route a user should take based on user preferences such as cost, travel time, and environmental impact.

Project Goals:

- Model a network interconnected where each route represents a specific mode of transportation (airplane, train, bus) with predefined capacity and travel time.
- Use the Ford-Fulkerson algorithm to optimize the flow of people while considering constraints.
- Implement a UI to allow easy input of source and destination cities.

Expected outcomes include a fully functional algorithm and frontend UI, a detailed technical report, and a final presentation summarizing the project's findings.

2 Project Plan

2.1 Timeline:

The overall timeline for the project is divided into phases:

- **Week 1** (October 7th - October 13th): Define project scope, establish team roles, and outline skills/tools.
- **Week 2** (October 14th - October 20th): Make project repository, and design the transportation network mode.
- **Week 3** (October 21 - October 27): Develop the maximum flow algorithm in C++ and begin integrating it with a user interface.
- **Week 4** (October 27th - November 3rd): Complete the backend and integrate frontend elements. Begin PowerPoint presentation.
- **Week 5** (November 4th - November 10th): Finalize the UI system, conduct testing, and continue with the report.
- **Week 6** (November 20th - November 29th): Revise and finalize the technical report and the PowerPoint presentation.
- **Week 7** (November 30th - December 4th): Final presentation, report submission, and project closure.

2.2 Milestones

Key milestones include:

- Project Scope and Plan: October 7th
- Network Model Design and Repository Setup: October 14th
- Algorithm Development and Initial Testing: October 27th
- UI Development: November 3rd
- System Integration and Documentation Draft: November 20th
- Final Testing and Report Draft: November 30th
- Final Presentation and Report Submission: December 4th

3 Team Roles

3.1 Skills Needed:

Team members need the following skills to complete the project:

- **Programming Languages:** C++ for the algorithm and backend processing.
- **Frontend Development:** Python GUI
- **Algorithm Knowledge:** Maximum flow algorithms, such as the Ford-Fulkerson or Edmonds-Karp methods.
- **Collaboration Tools:** GitHub for version control and Overleaf for report writing.

3.2 Team Responsibilities

Each team member will focus on specific areas, but roles may adapt as the project progresses:

- Frontend Development - Matt
- Backend Development - Kennedy
- Technical Documentation + Data Collection - Anna

4 Methodology

4.1 Data Selection and Preprocessing

The cities chosen were **NYC, Boston, Philadelphia, Washington DC, Houston, and Los Angeles.** These cities were chosen because of their large populations, cultural relevance, and variety of mass-transit options. Three modes of transportation were prioritized: **Bus, Train**, and **Airplane**. Data was collected from sources (see **Appendix B**) and displayed in **Table 1**.

Table 1: Data Collected for Travel Options

Source City	**Destination City**	**Mode**	**Capacity (People/Day)**	**Travel Time (Minutes)**	**Cost (USD)**	**Environmental Impact (CO2 kg)**
New York City	Chicago	Airplane	160	162	125	288
New York City	Boston	Airplane	76	83	120	89
New York City	Washington DC	Airplane	76	84	105	92
New York City	Houston	Airplane	160	225	150	195
New York City	Los Angeles	Airplane	160	330	250	450
Boston	Chicago	Airplane	150	182	200	332
Boston	Philadelphia	Airplane	76	105	180	38
Boston	Washington DC	Airplane	76	112	159	291
Boston	Houston	Airplane	160	215	240	350
Boston	Los Angeles	Airplane	160	310	270	470
Philadelphia	Chicago	Airplane	150	151	314	78
Philadelphia	Houston	Airplane	160	220	230	310
Philadelphia	Los Angeles	Airplane	160	320	280	480
Washington DC	Chicago	Airplane	150	134	210	57
Washington DC	Houston	Airplane	160	210	240	330
Washington DC	Los Angeles	Airplane	160	305	260	460
Chicago	Houston	Airplane	160	195	200	275
Chicago	Los Angeles	Airplane	160	270	230	390
Houston	Los Angeles	Airplane	160	285	240	420
Boston	New York City	Train	650	252	60	14
New York City	Philadelphia	Train	650	77	80	6
Philadelphia	Washington DC	Train	650	129	50	50
Boston	Chicago	Train	350	1342	145	52

Source City	Destination City	Mode	Capacity (People/Day)	Travel Time (Minutes)	Cost (USD)	Environmental Impact (CO2 kg)
New York City	Chicago	Train	350	1342	145	52
Washington DC	Houston	Train	350	1100	130	100
Chicago	Houston	Train	350	1000	140	110
Chicago	Los Angeles	Train	350	1500	170	120
Houston	Los Angeles	Train	350	1600	180	130
Boston	New York City	Bus	50	270	40	35
New York City	Philadelphia	Bus	50	130	35	15
Philadelphia	Washington DC	Bus	50	170	30	114
Boston	Chicago	Bus	50	1850	130	130
New York City	Chicago	Bus	50	1260	125	113
Washington DC	Houston	Bus	50	1450	140	120
Chicago	Houston	Bus	50	1400	150	125
Chicago	Los Angeles	Bus	50	2100	180	140
Houston	Los Angeles	Bus	50	2300	200	150

4.2 Literature review

To design an effective system, we researched existing research and applications in transportation networks, optimization algorithms, and user-preference-based routing. The Ford-Fulkerson algorithm has been widely used in logistics and network flow optimization. There are existing systems like Google Maps that suggest routes using different modes of transportation and tools like IATA's flight environmental impact metrics, however, there is not a system that incorporates this.

Our project aims at combining these methodologies by optimizing for maximum flow and user-defined preferences simultaneously. Current systems don't optimize city-to-city flows with set constraints like route capacities. Unlike general routing systems, our project explicitly incorporates CO2 impact as a factor along with different modes of transportation, and travel cost.

4.3 Object-Oriented Overview of Code

Object-Oriented Design was prioritized in the development of the project. Modularity and abstraction were prioritized in the design to ensure that the code was portable to other datasets and frontends. Therefore, the data was stored on an external **.csv** file, which is a common template for spreadsheets. The backend and frontend are independent of each other.

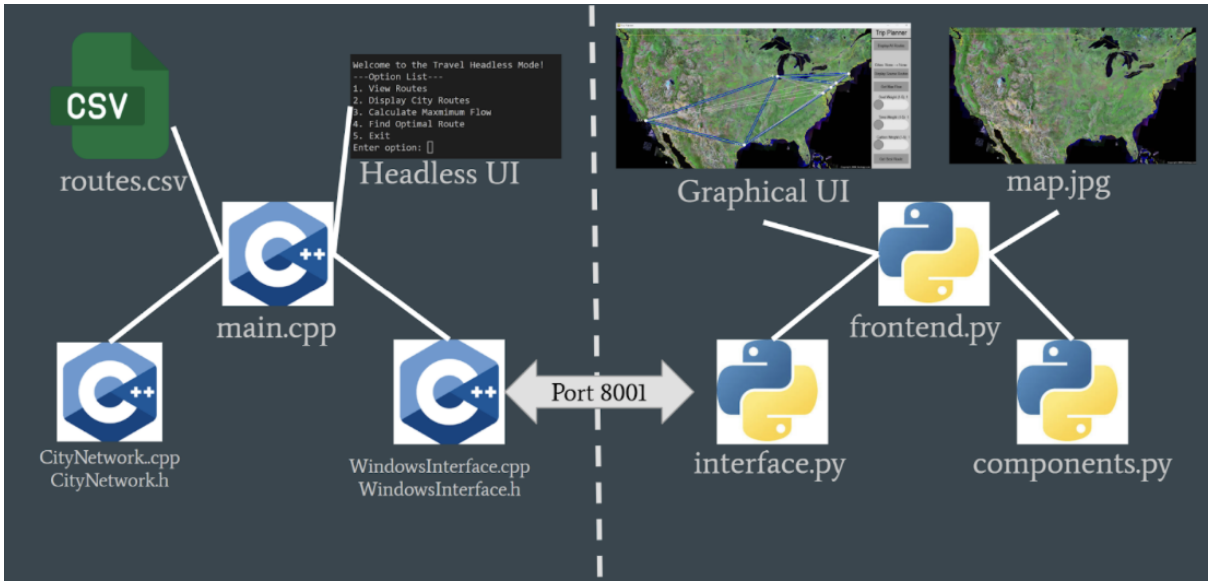

Figure 1: File Overview of Project

5 Classes and Data Structures used in the Backend

The backend was programmed in C++ due to its execution speed. This C++ backend consisted of three files:

- *main.cpp*: the file that holds the data reading and the headless UI
- *CityNetwork.cpp/CityNetwork.h*: the file that holds the **Route , CityNetwork**, and **FlowResult** data structures.
- *WindowsInterface.cpp/WindowsInterface.h*: the file that handles basic socket communications in Windows.

The following classes were used in the *CityNetwork.cpp* file.

5.0.1 Route

A route class was used to represent a route between two cities. It stores the following attributes:

- sourceCity: the starting city
- destinationCity: end point
- capacity: max amount of people that can travel on this route
- time: the travel time of the route in hours
- mode: type of transportation (airplane, train, bus)
- cost: The route cost in dollars
- environmentalImpact: environmental cost (CO2 emitted in kg)

5.0.2 CityNetwork

A second class called CityNetwork is used to manage cities, routes, and the network they form. It is used in the main operations such as finding the maximum flow and optimizing routes. It has the following members:

- routes: a vector of Route objects that stores the details about the routes in the network
- validCities: a set of city names to check if the city is in the network

- capacity: a 2D matrix that represents connections between cities, capacity[i][j] which shows the max amount of people that can travel from city i to city j.

5.0.3 Flow Result

This is the data structure that is returned by the maximum flow algorithm. It has the following datatypes:

- flow: the total flow between the two cities
- connections: the routes between the cities
- capacities: the flow in each connection

5.0.4 Vector

Vectors are used in the following:

- routes: stores all the routes in the network
- capacity: adjacent matrix to store capacity between cities
- parent: tracks the parent of each node during path finding

5.0.5 Adjacency Matrix

The Adjacency Matrix is a data structure used in the Ford-Fulkerson algorithm to determine the viability of connections between nodes.

5.0.6 Capacity Graph

The adjacent matrix is a 2D table that shows the connections between cities (nodes) in the network. Each cell in the matrix, capacity[i][j] shows the max number of people that can travel directly from city i to city j. If there is no direct connection it will have a value of 0. Each row in the matrix corresponds to a source city and each column corresponds to a destination city. The cityToIndex and indexToCity functions are used to map the city names to indices or back to city names so that the matrix can be used. When a new city is added, a new row is added to represent connections from the new city and a new column is added to represent connections to the new city. For example, if you start with one city: NYC the matrix will look like this:

$$\text{capacity} = \begin{bmatrix} 0 \end{bmatrix}$$

After adding the city **Boston**, the matrix updates to:

$$\text{capacity} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (\text{NYC} \rightarrow [\text{NYC, Boston}], \text{Boston} \rightarrow [\text{NYC, Chicago}])$$

After adding the city **Chicago**, the matrix updates again to:

$$\text{capacity} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(NYC → [NYC, Boston, Chicago], Boston → [NYC, Boston, Chicago], Chicago → [NYC, Boston, Chicago])

When you add a route using the addRoute function the capacity between the two cities is updated. For example, if a route from NYC to Boston is added with a capacity of 100 and a route from Boston to Chicago with a capacity of 30:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(NYC → [NYC, Boston, Chicago], Boston → [NYC, Boston, Chicago], Chicago → [NYC, Boston, Chicago])

5.0.7 Residual Graph

The residual graph is a copy of the capacity matrix at the beginning of the Ford-Fulkerson algorithm. The adjacency matrix is used as the residual graph for the algorithm where the residual graph shows the remaining capacity on each route after a flow has been assigned. If a cell in the matrix, residual[i][j] is greater than 0 it means there is still capacity to send more people from city i to city j. For example if the residual graph is the following:

$$\text{residual} = \begin{bmatrix} 0 & 100 & 50 \\ 0 & 0 & 30 \\ 10 & 20 & 0 \end{bmatrix}$$

(NYC → [NYC, Boston, Chicago], Boston → [NYC, Boston, Chicago], Chicago → [NYC, Boston, Chicago])

If the algorithm finds the path NYC -¿ Boston -¿ Chicago, the minimum capacity along the path is 30 which is from Boston to Chicago. It updates the forward edges by reducing it by 30, and the reverse edges are increased by 30.

$$\text{residual} = \begin{bmatrix} 0 & 70 & 50 \\ 30 & 0 & 30 \\ 10 & 50 & 0 \end{bmatrix}$$

(NYC → [NYC, Boston, Chicago], Boston → [NYC, Boston, Chicago], Chicago → [NYC, Boston, Chicago])

The residual from NYC to Boston changed where the original capacity was 100, but then 30 people were sent so the remaining capacity is 70. From Boston to Chicago, the original capacity was 30 but then 30 people were sent so the remaining capacity is zero. The reverse edges, Boston to NYC, and Chicago to Boston, were increased by 30.

If another flow is sent along a new path, the residual graph will change again. This continues until no more augmenting path exists.

5.0.8 Adding Cities and Routes

5.0.9 Adding a City

The function addCity checks if a city is already in the network by using cityToIndex.find(city) where cityToIndex is a map that maps city names (strings) with unique indices (integers). If the city isn't found it adds the city. It assigns a unique index where the index is equal to the current size of the indexToCity vector. It then adds the city to maps to map the city name to its city and then stores the city name at the index position in the vector. The adjacency matrix (capacity) is then expanded by adding a new column to every existing row and a new row for the new city.

5.0.10 Adding a Route

The add route function first ensures both cities are in the network, if they do not exist it calls addCity for the source and destination city. It uses the cityToIndex map to convert the city names to their indices. It then updates the capacity of the adjacency matrix between the source and destination city and then adds the route to the routes list.

5.1 Backend - Frontend Middleware Implementation

Communications between the frontend and backend were handled with sockets. Sockets are a way that Operating Systems communicate between processes. In C++, socket implementation is Operating System-specific. Therefore, the WindowsInterface only works on the Windows operating system. If a user does not have the Windows OS, they can use the Headless UI. The C++ implementation uses the *winsock* library with basic functionality (see **Appendix A**). The functions are as follows:

- void wait_for_client(): Wait until a client connects to the socket
- void send_msg(string message): Send message to the client
- string rcv_msg(): Returns whatever is in the buffer from the client

This message is then parsed using a custom communication protocol. Commands are sent from clients in the form *Command Parameters*. The client then responds. The protocol is shown below:

Table 2: Command Protocol Documentation

Command	**Parameters**	**Example**	**Response**	**Example Response**
`"Display"`	City Code — `"All"`	`Display BOS`	Src, Dst, Mode;...	`NYC, BOS, Bus; CHI, BOS, Train;`
`"Flow"`	Src Dst	`Flow BOS NYC`	Cap:City, City,...;...	`20: BOS, PHI, NYC;`
`"Route"`	Src Dst Cost Time Impact	`Route BOS NYC 1 5 1`	Src, Dst, Mode;...	`NYC, BOS, Bus;`

The frontend will send a request when a button is pressed. This will be stored in the buffer in the backend until the rcv_msg() function is called. When this is called, the message is read and processed.

5.2 Classes and Data Structures used in the Frontend

The frontend is written in Python because of its ease of use. Because of the usage of sockets, any frontend can be used as long as it connects to the correct port (port 8001) and uses the protocol specified in **Table 2**. PyGame is used for the GUI because of its widespread use in GUI applications. The Python frontend consists of the following files:

- *frontend.py* - contains main function, handles sending and receiving logic as well as route display
- *components.py* - consists of classes created for PyGame
- *interface.py* - handles socket communication with backend
- *map.jpg* - background image loaded by *frontend.py*

5.2.1 Component, Slider, and Button Class

In *components.py*, three custom GUI classes are created: **Component, Slider**, and **Button**. **Slider** and **Button** both override **Component** to allow for polymorphic behavior (all components are stored in a list). They all contain the draw(), on_click(tuple pos), and on_release() functions. These functions have no actual logic; they are simply for display purposes.

5.2.2 Displaying City Position

To make displaying cities easier, a map with a mercator projection was used. This is a map that stretches more northern latitudes so longitude and latitude lines appear as a grid.

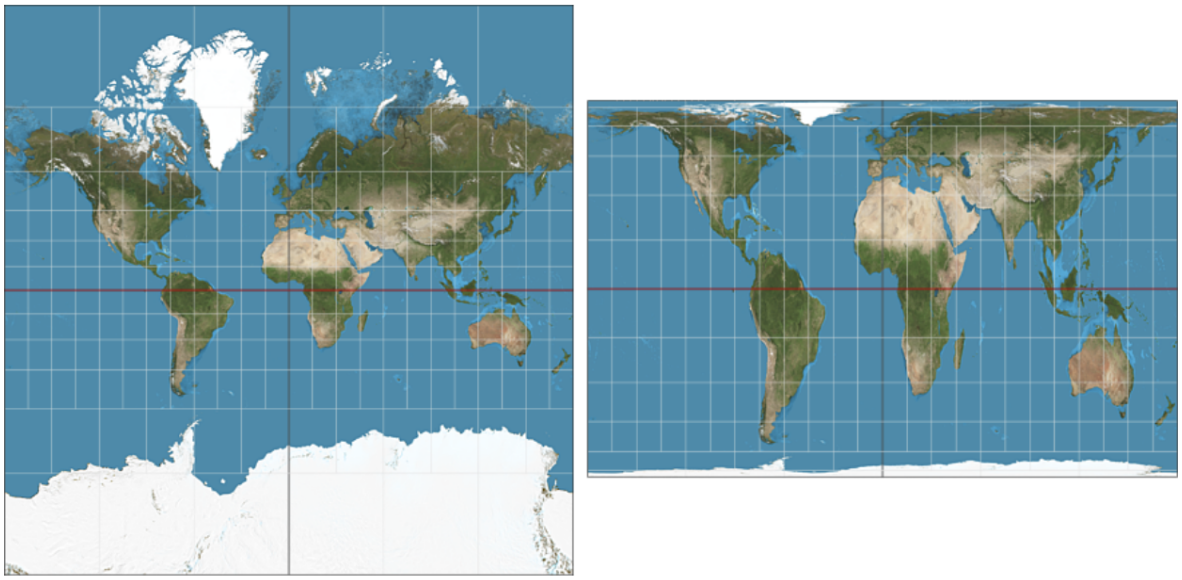

Figure 2: Mercator (left) vs Peters (right) projection of the Earth

Because a Mercator Projection is used, the coordinates of each city are recorded in a vector data structure and translated to $< x, y >$ position with the following equation: $[x, y] = [21.433 * long + 2686.177, -28.185 * lat + 1420$.

5.2.3 Main Display Loop

The main loop is run every ˜0.01 seconds where the screen is redrawn. The pseudocode is displayed below. This loop handles user input and route and city display. Plane routes are thin and grey, bus routes are blue, and train routes are green.

5.3 Algorithm Design

5.3.1 Ford Fulkerson Algorithm

The Ford-Fulkerson Algorithm is used to calculate the maximum flow between cities. It uses a residual graph that is initialized from an adjacency matrix (capacity). Then it finds an augmenting path using Breadth-First Search (BFS) with the "findAugmentingPath" function. An augmenting path is a path from the source to the sink where there is a remaining capacity along all edges in the path. It updates the residual graph by reducing capacities along the augmenting path and increasing reverse capacities for backwards flow. This repeats until no more augmenting paths are found, and the total flow is the sum of the flows through all the paths found.

5.4 BFS

BFS is used in the Ford-Fulkerson to find an augmenting path.The inputs are the source (the starting city as an index), the sink (the destination city as an index), and the residual which is the residual capacity graph. It starts from source, marks it as visited, and then adds all unvisited neighbors with positive residual capacity to the queue. It keeps track of the path using the parent array. It outputs true if a path is found, and false if it's not and then updates parent to show the path. This process ends when the sink (destination) city is reached or all reachable cities are visited.

5.5 Sorting

The built in sort function is used in finding the busiest routes by capacity and optimizing routes based on user input preferences. It has a time complexity of O(nlogn).

5.6 Route Optimization

The route optimization uses a modified version of Dijkstra's Algorithm to calculate the optimal path between two cities given a series of weights for different attributes. Dijkstra's Algorithm is a greedy algorithm typically used in applications such as networking. This project used a more dynamic approach, which was more accurate but less time-efficient.

Dijkstra's Algorithm works by creating a list of costs associated with every node. Starting with the source node, the cost to get to each node is calculated by adding the cost of the node to the cost of the route. If this cost is less than the other node's current cost (initialized to infinity), the other node's cost is updated. This is done with every node.

The main differences between the standard algorithm and the route optimization algorithm are that we stored the source Route (as a **Route** datatype) and repeated the calculation for each node. Rather than only performing the calculation for each node once, we performed it n times. This allowed for the nodes to check for more less direct routes, which was important given the prevalence of undesirable direct routes. The cost was created by dividing the route's specific cost parameter (time, cost in USD, or environmental impact) by the maximum expected value (300 USD, 600 minutes, 0.5 tons CO2). This was then multiplied by the weight and added together.

5.7 Backend Pseudocode

5.7.1 findAugmentingPath

Algorithm 1: findAugmentingPath

```
Algorithm findAugmentingPath(source, sink, parent, residual)
begin
    Initialize visited array to false
    Initialize an empty queue q
    Push source node into q
    Mark source as visited
    Set parent[source] = -1
    while q is not empty do
        curr = q.pop()
        for each node next from 0 to size of residual do
            if next is not visited and residual[curr][next] > 0 then
                Set next as visited
                Set parent[next] = curr
                if next == sink then
                    return True
                end
                Add next to q
            end
        end
    end
    return False
end
```

5.7.2 Ford-Fulkerson Algorithm

Algorithm 2: calculateMaxFlow

```
Algorithm calculateMaxFlow(sourceCity, destinationCity)
begin
    if sourceCity or destinationCity is invalid then
        Print error message
        return
    end
    Initialize maxFlow = 0
    Convert sourceCity and destinationCity to their index values
    Copy capacity graph into residual graph
    Initialize parent array
    while findAugmentingPath(source, sink, parent, residual) do
        Set pathFlow = very large number
        Traverse the augmenting path from sink to source:
        for each node v in the path do
            u = parent[v]
            pathFlow = MIN(pathFlow, residual[u][v])
        end
        Traverse the augmenting path again to update residual graph:
        for each node v in the path do
            u = parent[v]
            residual[u][v] -= pathFlow
            residual[v][u] += pathFlow
        end
        Increment maxFlow by pathFlow
    end
    Print "Maximum flow from sourceCity to destinationCity is maxFlow"
end
```

5.7.3 Optimized Route Based on User Preferences

Algorithm 3: calculateOptimizedFlow

```
Algorithm calculateOptimizedFlow(sourceCity, destinationCity, timeWeight,
 costWeight, impactWeight)
begin
    Initialize weights array to [1000000]× len(cities)
    Initialize routes array to [empty Route]× len(cities)
    if timeWeight == 0 and costWeight == 0 and impactWeight == 0 then
        Set all elements in weights to 1
    end
    for i from 0 to weights.size do
        for j from 0 to weights.size do
            if weights[j] ≠ 1000000 then
                for each route in routes do
                    Check if city is in route
                    Calculate route weight and add to city weight
                    if route's other city's weight > calculated weight then
                        Set other city's route to this route
                        Set other city's weight to calculated weight
                    end
                end
            end
        end
    end
    Initialize returnVal as vector¡Route¿
    Set currCity = destinationCity
    while currCity ≠ sourceCity do
        Add currCity's route to returnVal
        Set currCity to other city in route
    end
    return returnVal
end
```

5.7.4 Frontend Pseudocode

Algorithm 4: main

```
Algorithm main()
begin
    if user pressed the mouse then
        Run on_click(mouse_pos) for each component
        if mouse press was on a city then
            Append the city to cities_clicked list
        end
    end
    if user released the mouse then
        Run on_release() for each component
    end
    Draw background
    for each city in route_list do
        Display the route with the correct color and weight
    end
    Draw each city
    Draw labels
    Draw components
    Update display
end
```

5.7.5 Processing Non-Flow Responses

Algorithm 5: ProcessBackendMessage

```
begin
    Clear route_list
    Receive message from backend
    Split message based on ';' and store in splitMessage
    Create empty vectors: busses, trains, airplanes
    for each substring in splitMessage do
        Split substring by ',' into routeInfo
        if routeInfo is invalid then
            Skip to the next iteration
        end
        Add the route to the appropriate vector (busses, trains, or airplanes)
    end
    Extend busses, trains, and airplanes to route_list in that order
end
```

5.7.6 Processing Flow Responses

Algorithm 6: UpdateRouteListWithCapacity

```
begin
    Clear route_list
    Receive message from backend
    Split message based on ';' and store in splitMessage
    for each substring in splitMessage do
        Split substring by ':' and record capacity
        for each route in route_list do
            Initialize addCity to True
            for each route in route_list do
                if city is in route then
                    Set addCity to False
                    Increment the route's capacity by recorded capacity
                end
            end
            if addCity then
                Add the city to route_list
            end
        end
    end
end
```

6 Results

6.1 Time Spent

Time spent on each major aspect of the project is documented below.

- Backend: 1.5 hr/week
- Frontend: 1 hr/week
- Testing and Optimization: 1 hr/week
- Reports and Presentation: .5 hr/week
- Total: ˜32 hours over the past 2 months per person

6.2 Time Complexity

6.2.1 findAugmentingPath

- First the visited array and queue is initialized which is O(N) where N is the number of nodes (cities).
- Then the BFS loops where for every node in the graph it checks its neighbors. In the worst case, all nodes and edges are visited once.
- Total Time Complexity: O(N+E) where E is the number of edges (routes).

6.2.2 Ford-Fulkerson Algorithm

- The capacity graph is copied to the residual graph: O(N^2)
- The parent array is initialized: O(N)
- The while loop runs at most F times where F is the maximum possible flow in the network.
- Each iteration updates the residual graph which means it traverses the path found by BFS: O(N).
- Total Time Complexity: O($N^2 * E$) where each BFS takes O(N+E).

6.2.3 Route Optimization Algorithm

- Each city is iterated through in two nested for loops: O(n^2)
- Each route is iterated through: O(m)
- The cost is calculated and compared: O(1)
- Total Time Complexity: O($n^2 * m$) where n is number of cities and m is number of routes

6.2.4 Processing Non-Flow Responses

- Splits the string: O(n)
- Iterates through each route: O(n)
- Adds the route to the list: O(1)
- Total Time Complexity: O(n)

6.2.5 Processing Flow Responses

- Splits the string: O(n)
- Iterates through each route: O(n)
- Iterates through the route_list to check if route is accounted for: O(n)
- Total Time Complexity: O(n^2)

7 Discussion

The program successfully optimizes the flow of people between 7 cities in the United States. In the front end, we integrated a visual map of the country with different line thickness to differentiate taking an air travel (airplane) and land travel (train and bus). These results can be used to help people choose the most optimal trip based on capacity, cost, and environmental impact constraints in a time effective way. It also used scalable algorithms that could be applied to additional routes.

The frontend and headless UI both work effectively. The Ford-Fulkerson Algorithm provides results that make sense, as it sends large amounts of people through direct train and bus routes that are high capacity and small amounts of people through low-capacity flights. An example of this is shown in **Fig. 3**.

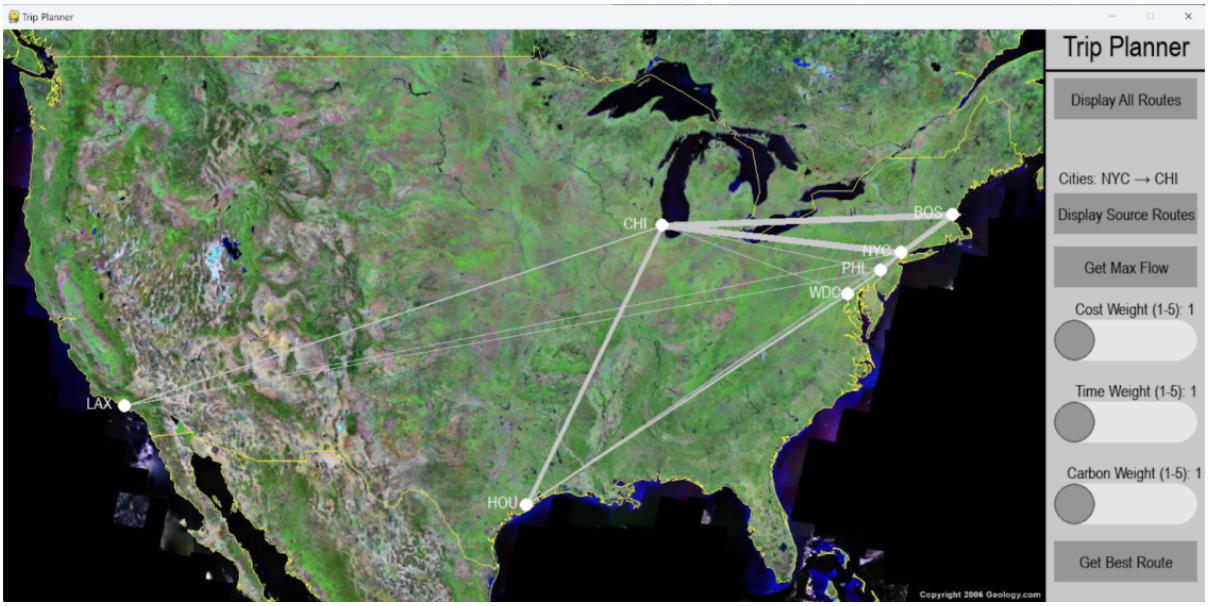

Figure 3: Result of Ford-Fulkerson Algorithm from NYC to Chicago

The optimized route algorithm also works as expected, as the selected modes of transportation depend on the inputs the user puts in. An example of this is shown in **Figure 4**.

Figure 4: Boston to DC route optimization for cost(left), time(center), and carbon footprint(right).

As can be seen, the algorithm chose connecting bus and train routes for cost, a direct flight for time, and connecting train routes for carbon emissions. The routes were also able to be displayed as seen in **Fig. 5**.

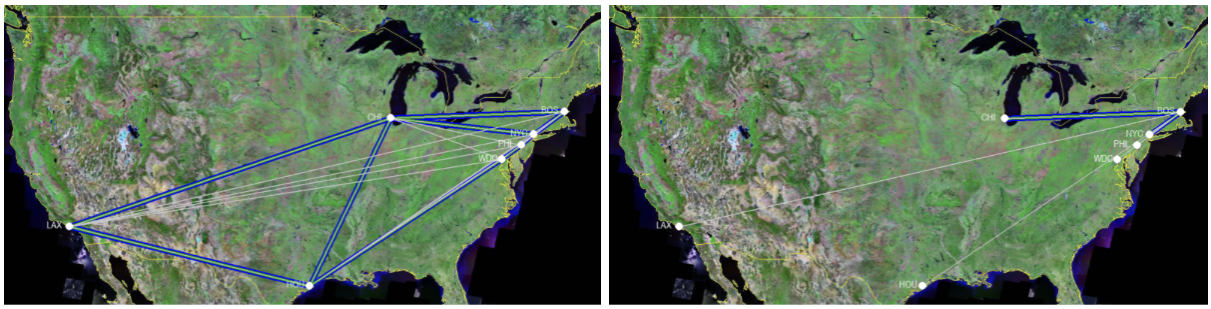

Figure 5: Display of all routes(left) and routes from Boston(right)

Edge cases were also designed around, such as when the user specifies that all weights are at "1", or "low

priority". If the user does not use the Windows OS, a headless UI can be used (see **Fig. 6**).

```
Welcome to the Travel Headless Mode!
---Option List---
1. View Routes
2. Display City Routes
3. Calculate Maxmimum Flow
4. Find Optimal Route
5. Exit
Enter option: 3

Enter the source city: BOS
Enter the destination city: NYC
Maximum flow: 200
Paths used:
80 sent via NYC -> BOS
80 sent via NYC -> CHI -> BOS
10 sent via NYC -> WDC -> BOS
10 sent via NYC -> HOU -> BOS
10 sent via NYC -> LAX -> BOS
10 sent via NYC -> PHI -> BOS
```

Figure 6: Flow Algorithm Run in Headless Mode

The project developed further than our initial objectives. We added the factor of environmental impact, created a more visually appealing map, added additional cities, and allowed more user interaction.

8 Conclusion

This project shows the different routes a person can take between the cities of New York City, Boston, Philadelphia, Chicago, Washington DC, Houston, and Los Angeles using three different modes of transportation: airplane, train, and bus. The program displays the cost, capacities of the vehicles, environmental impact, and optimizes the flow of people between these cities using the Ford Fulkerson Algorithm.

Some limitations in this project are the amount of cities, fixed capacities, fixed travel times, fixed costs, and limited modes of transportation. We don't consider people traveling in private vehicles like cars or RVs.

For future improvement, we could further this project by adding more cities like Seattle, San Francisco, etc. and even add cities in other countries. If we were to do this we'd need to consider the limitations of geographical boundaries. For example, if you're traveling from Boston to London, it can only be done on airplanes or boats because of the Atlantic Ocean. We could do real time data integration to account for price fluctuations that happen over weekends and holidays. We could also consider and differentiate the cost for services like business class tickets on airplanes.

9 A Appendix A: Repository

Project GitHub Repository

10 B Appendix B: References

- Flight Time: american airlines -`https://www.aa.com/homePage.do`
- Flight Cost: kayak - `https://www.kayak.com` and american airlines - `https://www.aa.com/homePage.do`
- Flight passengers: american airlines - `https://www.aa.com/homePage.do`
- Flight Environmental Impact: `https://www.iata.org` and `https://www.epa.gov`
- Bus Time: greyhound - `https://www.greyhound.com`

- Bus Cost: wanderu - https://www.wanderu.com and greyhound - https://www.greyhound.com
- Bus passenger: greyhound - https://www.greyhound.com
- Bus Environmental Impact: https://www.epa.gov and https://www.ucsusa.org
- Train Time: amtrak - https://www.amtrak.com/home.html
- Train Cost: wanderu - https://www.wanderu.com and amtrak - https://www.amtrak.com/home.html
- Train passenger: amtrak - https://www.amtrak.com/home.html
- Train Environmental Impact: https://www.energy.gov and https://railroads.dot.gov
- https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/
- https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

11 C Appendix C: Additional Figures

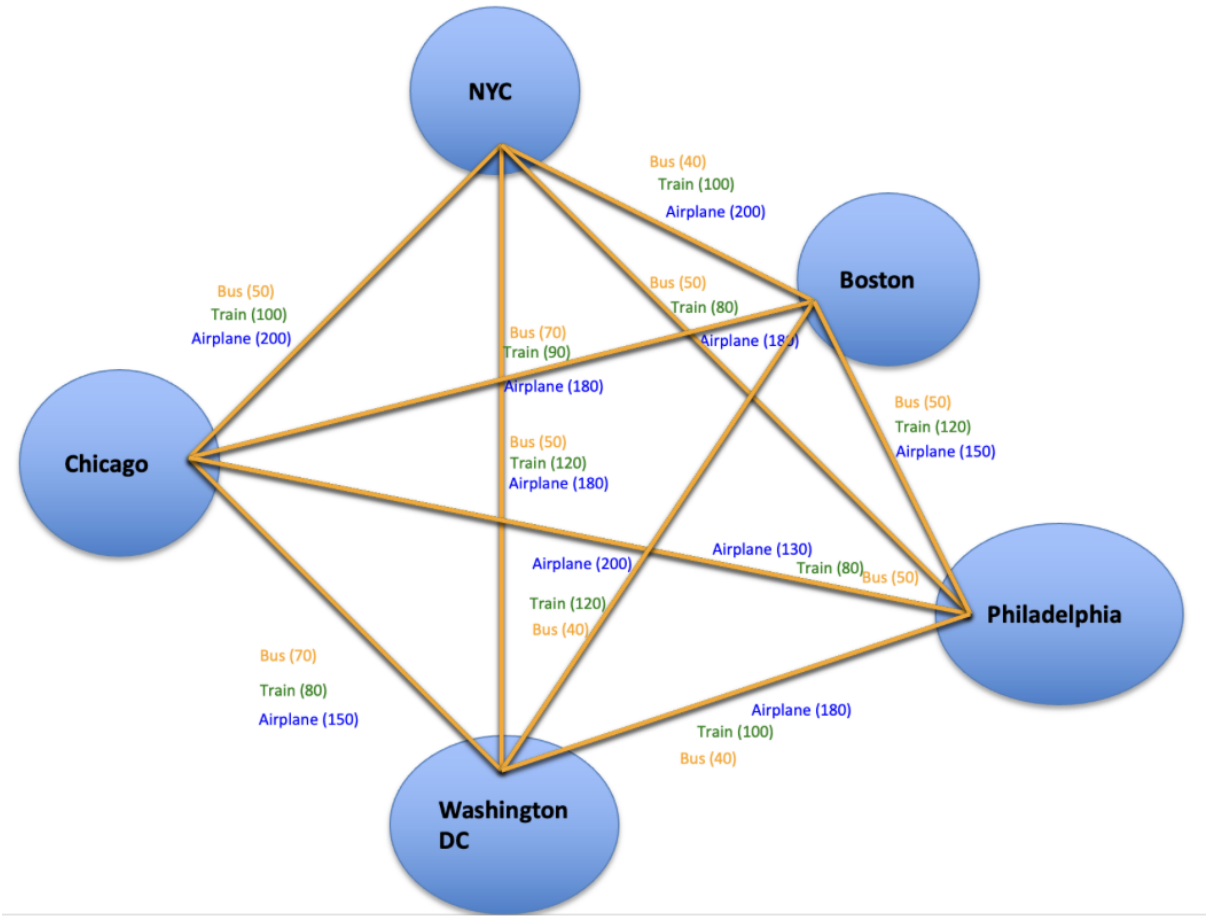

Figure 7: Visual of Route Network