# Optimizing City-to-City Transportation with Ford-Fulkerson and Dijkstra's Algorithm

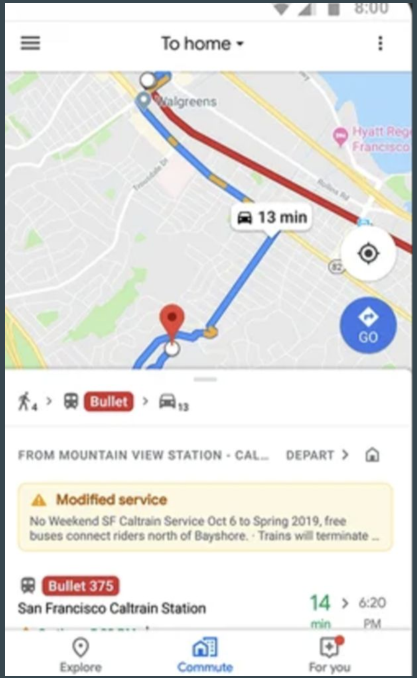
Northeastern University
College of Engineering
EECE 2560: Fundamentals of Engineering Algorithms
Kennedy Scheimreif, Anna Valades, Matthew Geisel

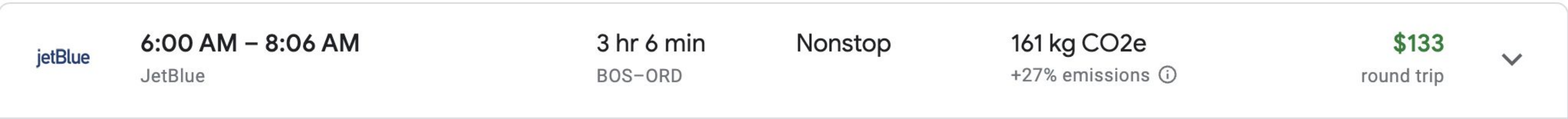
LVX VERITAS VIRTVS
Northeastern University

# Existing Technology

**Travel Booking Industry worth $520 billion**

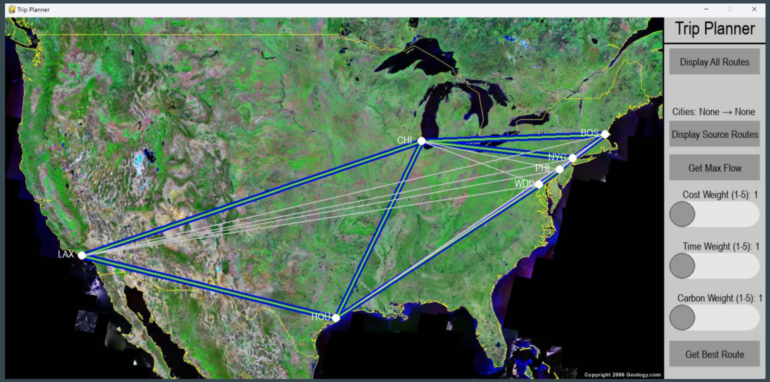
- Many mass-transit systems must be booked independently (i.e. Amtrak, the T)
- Multi-modal transportation systems (Google Maps)
- Most booking sites not optimized for multiple people
  - Booking groups of 10+ difficult & pricey

# Our Solution

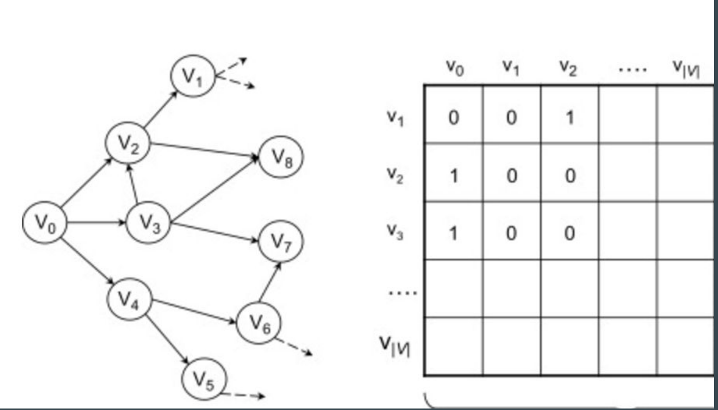
- **Ford-Fulkerson** and **Breadth-First Search** to find maximum flow
- **Dijkstra's Algorithm** to optimize people flow using different modes of transportation and user's traveling preferences
- Real-time data from APIs are expensive
  - **CSV** file as a stand-in
- Backend uses **C++** and the frontend uses **Python** and is integrated with a custom **socket communication protocol**

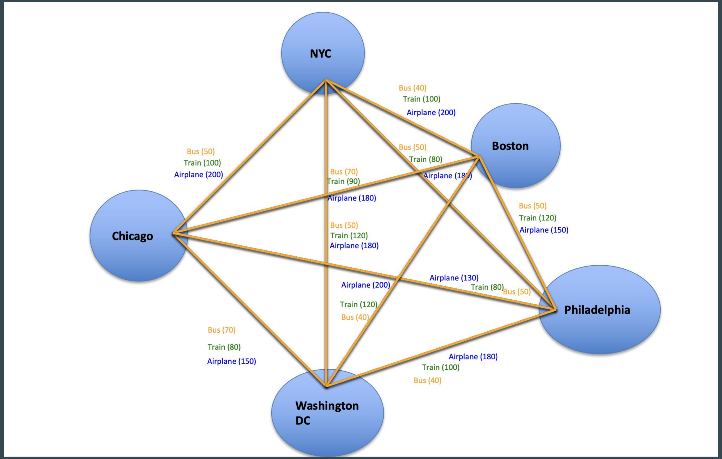



- Efficient
- Scalable
- User-centric

# Methodology: Ford-Fulkerson Algorithm

- **Adjacency Matrix:** 2D matrix where the rows and columns are nodes in the graph.
- **Residual Graph:** a modified version of the original matrix that shows the residual capacity (capacity - current flow).



# Pseudocode + Time Complexity Used Pt 1

```
findAugmentingPath(source, sink, parent, residual):
    initialize visited array to false
    empty queue q
    push source node into q
    mark source as visited
    set parent[source] = -1

    while q not empty:
        curr = q.pop()

        for each node next from 0 to size of residual:
            if next is not visited and residual[curr][next] > 0:
                set next as visited
                set parent[next] = curr
                if next == sink:
                    return True
                add next to the queue
    return False
```

- **Augmenting Path**: a path from the source to the sink where there is a remaining capacity along all edges in the path (bottleneck determines amount of additional flow).
- **BFS:** uses a queue to find the augmenting paths by marking nodes as visited and tracking the path.
- **Time Complexity:** O(n+m)

# Pseudocode + Time Complexity Used Pt 2

## Ford-Fulkerson Algorithm

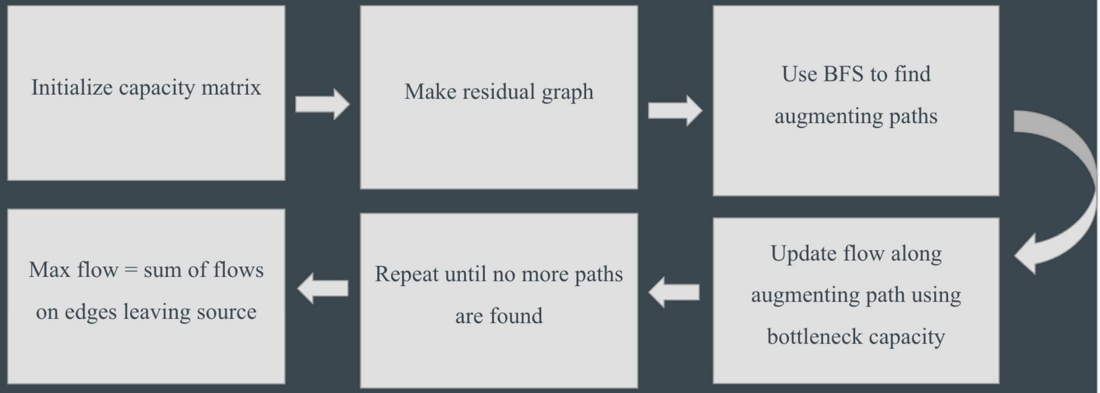
```
calculateMaxFlow(sourceCity, destinationCity):
    if sourceCity or destinationCity is invalid:
        print error message
        return

    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):
        set pathFlow = very large number
        traverse the augmenting path from sink to source:
            for each node v in the path:
                u = parent[v]
                pathFlow = MIN(pathFlow, residual[u][v])

        traverse the augmenting path again to update residual
          graph:
            for each node v in the path:
                u = parent[v]
                residual[u][v] -= pathFlow
                residual[v][u] += pathFlow

        increment maxFlow by pathFlow
    print "Maximum flow from sourceCity to destinationCity is
      maxFlow"
```




**Time Complexity:**

- O(n^2*m)

# Pseudocode + Time Complexity Pt 3

## Get City Connections:

**Algorithm getCityConnections(city)**
**begin**
  Print "Cities directly connected to city:"
  Initialize *connectionFound* as false
  **for** *each route in routes* **do**
    **if** *route.sourceCity == city* **then**
      Set *connectionFound* to true
      Print "Destination: *route.destinationCity* — Mode: *route.mode* — Travel Time: *route.time* hours — Cost: $*route.cost*"
    **end**
  **end**
  **if** *connectionFound is false* **then**
    Print "No direct connections found for city."
  **end**
**end**

---

Time Complexity:

- Iterate through the routes to find connections: O(R) where R is number of routes
- Print results O(1)
- Total Time Complexity: O(R)

## Get Busiest Routes:

**Algorithm getBusiestRoutes()**
**begin**
  **if** *routes is empty* **then**
    Print "No routes available in the network."
    **return**
  **end**
  Sort *routes* in descending order of capacity:
  Compare capacity of route *a* and route *b*
  **if** *a.capacity ¿ b.capacity* **then**
    Return *a.capacity ¿ b.capacity*
  **end**
  Print "Busiest Routes by Capacity:"
  Set *limit* to the smaller of 5 or *routes.size()*
  **for** *i from 0 to limit-1* **do**
    Let *route* = *routes[i]*
    Print "*route.sourceCity* to *route.destinationCity* — Mode: *route.mode* — Capacity: *route.capacity* people — Travel Time: *route.time* hours"
  **end**
**end**

---

Time Complexity:

- Check if routes is empty: O(1)
- Sort routes by capacity using introsort which is a combination of quicksort and heap sort: O(RlogR)
- Print top 5 routes: O(5)
- Total Time Complexity: O(RlogR)

# Pseudocode + Time Complexity Pt 4

**Dijkstra's Algorithm**

1. Create **vectors** of weights & routes (weights set to very large)
2. Go through each valid weight city with **2 nested for loops**
    a. Go through every **route** that includes the city
        i. Calculate weight of route (cost / scalar) * weight + curr_weight
            1. Time: 600 mins, Carbon: 0.5 tons, Cost: $300
        ii. If the other city's weight is **more**, set its route & weight
3. Start at destination city
    a. Add its route to **return vector**
    b. Go to the other city in route

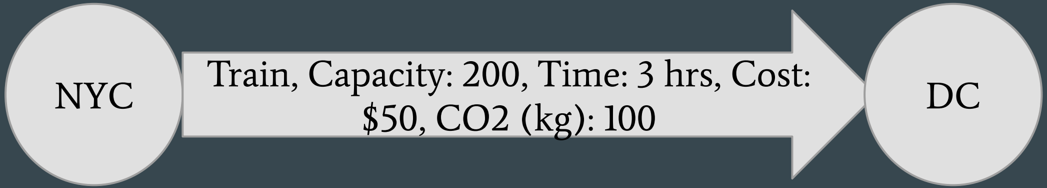
**Input:** src, dst(string), timeWeight, costWeight, impactWeight(int)

**Returns:** vector<Route>

**Time Complexity:** O(n^2*m) (n is # cities, m is # routes per city)

# Data Structures Pt 1

- **Route Class:** represents individual connections between cities with attributes and is stored as a **vector** of **objects** with the following data types:
  - Source City: **string**
  - Destination City: **string**
  - Capacity: **int**
  - Time: **double**
  - Mode: **string**
  - Cost: **double**
  - Environmental Impact: **double**




- **CityNetwork Class:** manages the overall network of cities and routes.
- **Parent Array:** tracks the parent of each node during BFS for path construction

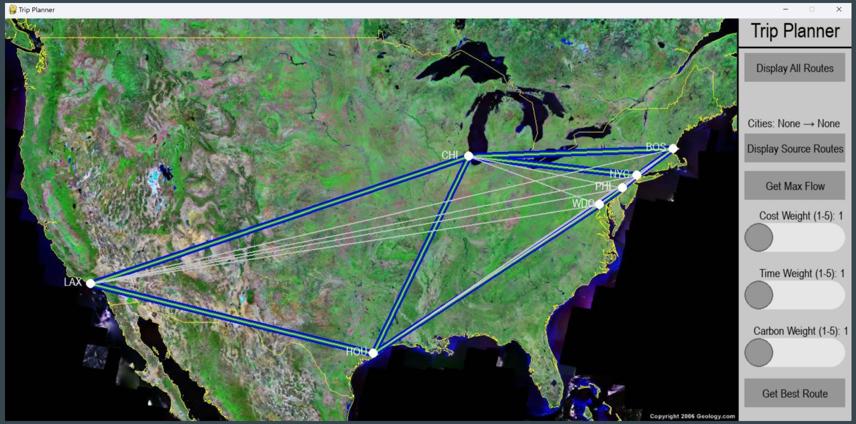
# Data Structures Pt 2

- **Queue:** used in the BFS to find augmenting paths where there is a queue of integers corresponding to cities
- **Vectors:** used to store a dynamic list of data elements such as cities, routes, and graph edges.
- **FlowResult struct:** stores data that gives result of Ford-Fulkerson algorithm
  - Flow: **int**
    - Total max flow
  - Connections: **vector<vector<string>>**
    - Stores vectors of city names
  - Capacities: **vector<int>**
    - Stores the number of people sent through each connection

# Backend - Frontend Integration

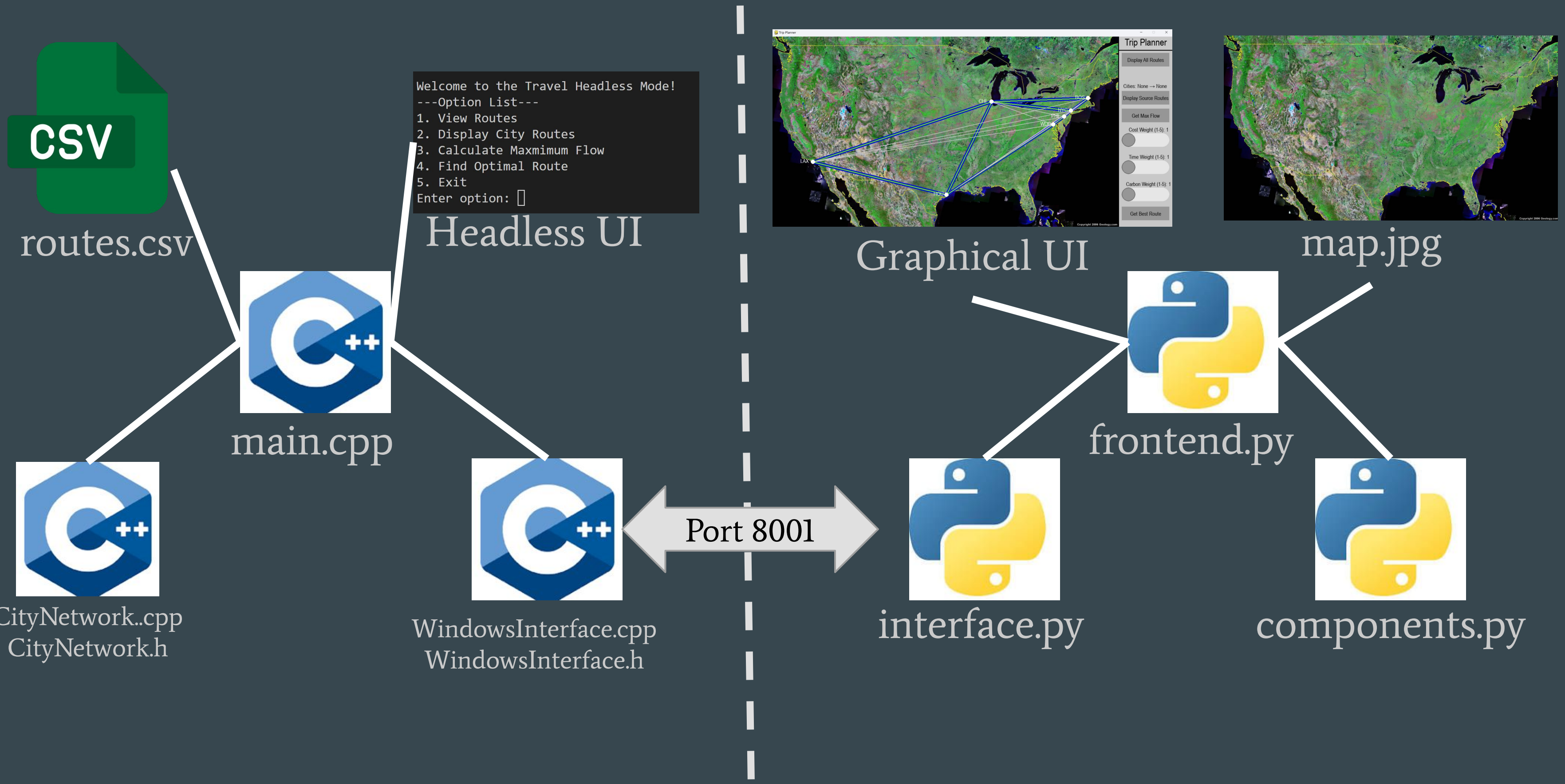
- **Socket:** Way that two processes can communicate (port 8001 used)
  - **C++: OS-Specific** (only Windows driver developed)
  - **Python: OS-Independent** (advantage of Python)
  - **Frontend-Independent** (easily integratable)
- Developed **custom communication protocol** of strings
  - Frontend: *Command Parameters*
    - "Display *src*" (set to All for all)
    - "Flow *src dst*"
    - "Route *src dst cost time impact*" (last 3 are **1-5**)
  - Backend: Response
    - Display/Route: "*src,dst,mode;…*"
    - Flow: "*capacity:city,city,…; …*"

# Frontend Implementation



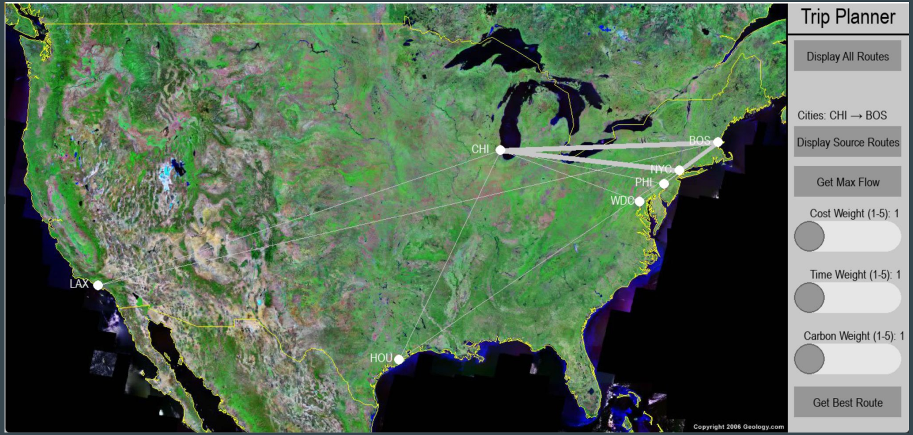

- Uses **PyGame** library (common for GUIs)
  - Loops every ~0.01 seconds
  - Displays **airplane**, **bus**, **train**
    - Except in max flow
- On press: send, recv, process, display
  - **Vector** (Python list) stores each route [src, dst, mode] & selected cities
  - For **display & optimize:** just processes message (**O(n)**)
  - **Flow:** has to sum weights of sub-parts (**O(n^2*m)**)
    - For each route, check if it already exists. If so, add the capacity.
- **Mercator Projection** used: easy to translate coordinates to position
  - [x, y] = [21.433 * long + 2686.177, -28.185 * lat  + 1420.191]

# Overall Code Structure



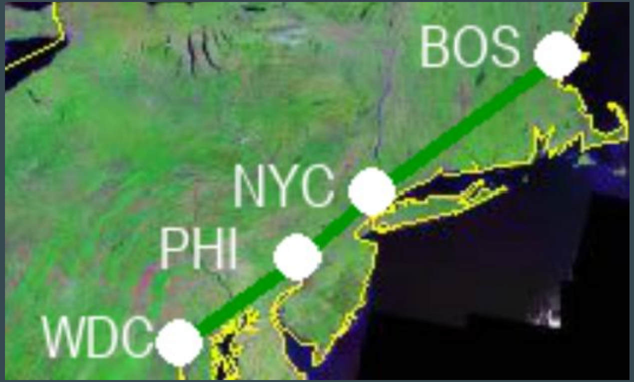
# Functionality Examples

## BOS -> CHI Max Flow




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

## BOS -> WDC Impact-Optimized Route




```
Enter the source city: BOS
Enter the destination city: WDC
Enter the time weight: 1
Enter the cost weight: 1
Enter the environmental weight: 5

Result:
PHI -> WDC (Train): cost- 50, time- 120, tons of carbon- 0
NYC -> PHI (Train): cost- 25, time- 60, tons of carbon- 0
BOS -> NYC (Train): cost- 30, time- 240, tons of carbon- 0
```

# Live Demonstration

# Analysis and Results: Time Spent on Project Per Person

- **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

| Date | Week | Coding Progress | Technical Report Progress | PowerPoint Progress | Team member |
|---|---|---|---|---|---|
| 2024-10-03 00:00:00 | Week 1 | Not Started | Started | Not Started | Kennedy |
| 2024-10-04 00:00:00 | Week 1 | Not Started | Started | Not Started | Anna |
| 2024-10-05 00:00:00 | Week 1 | Not Started | Started | Not Started | Matt |
| 2024-10-06 00:00:00 | Week 1 | Not Started | Started | Not Started | Kennedy |
| 2024-10-07 00:00:00 | Week 1 | Not Started | Started | Not Started | Anna |
| 2024-10-08 00:00:00 | Week 1 | Not Started | Started | Not Started | Matt |
| 2024-10-09 00:00:00 | Week 1 | Not Started | Started | Not Started | |
| 2024-10-10 00:00:00 | Week 2 | Not Started | Started | Not Started | Kennedy |
| 2024-10-11 00:00:00 | Week 2 | Not Started | Started | Not Started | Anna |
| 2024-10-12 00:00:00 | Week 2 | Not Started | Started | Not Started | Matt |
| 2024-10-13 00:00:00 | Week 2 | Not Started | Started | Not Started | Kennedy |
| 2024-10-14 00:00:00 | Week 2 | Not Started | Started | Not Started | Anna |
| 2024-10-15 00:00:00 | Week 2 | Not Started | Started | Not Started | Matt |
| 2024-10-16 00:00:00 | Week 2 | Not Started | Not Started | Not Started | |
| 2024-10-17 00:00:00 | Week 3 | Started | Started | Not Started | Kennedy |
| 2024-10-18 00:00:00 | Week 3 | Started | Started | Not Started | Anna |
| 2024-10-19 00:00:00 | Week 3 | Started | Started | Not Started | Matt |
| 2024-10-20 00:00:00 | Week 3 | Started | Started | Not Started | Kennedy |
| 2024-10-21 00:00:00 | Week 3 | Started | Started | Not Started | Anna |

# Analysis and Results: Key Findings and Interpretation of Results

- **Customer Use**
  - The system could be integrated in an app, helping users plan affordable, time-efficient, and environmentally sustainable trips
- **Industry Use**
  - City planners can use this tool to simulate and optimize transit networks, identify bottlenecks, and allocate resources more effectively
- **Combines Theory and Practical Applicability:**
  - Maps user preferences to real-world scenarios
  - Adaptable for both small-scale and large-scale use cases

# Discussion

**Implications of Findings:**

- Time-effective process for determining the most optimal trip
- Scalable algorithms able to be used for more routes

**Project Limitations:**

- Number of cities
- Fixed capacities, travel times, and cost
- Doesn’t include private modes of transportation (cars)

# Conclusion

- **Recommendations for Future Work:**
- Real time data integration with API
    - Consider price fluctuations over weekends and holidays
- Add more cities
- Consider how business class and other travel options affect price

# References


- https://www.aa.com/homePage.do
- https://www.greyhound.com/
- https://www.amtrak.com/home.html
- https://www.kayak.com/
- https://www.wanderu.com/
- https://www.iata.org/
- https://www.epa.gov/
- https://www.ucsusa.org/
- https://www.energy.gov/
- 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/


- https://www.grandviewresearch.com/industry-analysis/online-travel-booking-service-market-report
- https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php