Technical Report: Final Project
EECE 2560: Fundamentals of Engineering Algorithms

William Jappe, Johnathon Walton and Ryder Robbins
Department of Electrical and Computer Engineering
Northeastern University
jappe.w@northeastern.edu
walton.j@northeastern.edu
robbins.r@northeastern.edu

December 4, 2024

Contents

1 Project Scope

The scope of this project was to design and develop a real-time traffic navigation system capable of determining the shortest route for vehicles in a state network while incorporating traffic conditions based on the time of day. Our goal is to utilize DIjkstra's algorithm as the primary method of determining the shortest route between two locations, while modifying it to be able to support the consideration and calculation of traffic conditions.

1.1 Objectives

- Track a network of cities and roads with data classes.
- Use time-of-day data to calculate traffic conditions for roads.
- Implement Dijkstra's algorithm to compute the shortest path accurately.
- Visualize the network and routes using graphing libraries for users.

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

2 Project Plan

2.1 Timeline

The project is divided into phases, each with specific deliverables:

- **Week 1 (October 7 - October 13)**: Define project scope, establish team roles, set up project repository, and outline skills/tools.
- **Week 2 (October 14 - October 20)**: Begin development of system basics and begin testing researching and testing algorithms
- **Week 3 (October 21 - October 27)**: Continue coding system and work on backend integration with algorithm.
- **Week 4 (October 28 - November 3)**: Complete the backend, integrate frontend elements for the interface and start writing documentation.
- **Week 5 (November 4 - November 10)**: Finalize the system, conduct testing, begin the presentation and continue with the report.
- **Week 6 (November 11 - November 17)**: Refine frontend elements for interface and include screenshots in documentation and presentation.
- **Week 7 (November 18 - November 24)**: Finalize the technical report and the PowerPoint presentation.
- **Week 8 (November 25 - December 4)**: Final testing, report submission and project closure

2.2 Milestones

Key milestones include:

- Develop a Project Scope and Plan (October 7).
- GitHub Repository Setup and Initial Development (October 9).
- Determine steps of algorithm to find shortest route in network (October 20)
- Integrate Dikstra's Algorithm for the Backend (November 3
- Frontend Interface Completion (November 8).
- Final System Testing and Documentation (November 12).
- Complete Report and Presentation Drafts (November 20)
- Final Presentation and Report Submission (December 4).

3 Team Roles

- **Ryder Robbins**: Graph representation and traffic calculations.
- **William Jappe**: Geographic routing and presentation preparation.
- **Johnathon Walton**: Algorithm implementation and debugging.

4 Methodology

4.1 Graph Representation

The state network was represented as a graph with:

- **Edges (Roads):** Represented using the Edge structure with attributes for base time, actual travel time (considering traffic), and variation.

```
structure Edge:
    end: integer  // Destination city index
    Basetime: integer  // Base travel time
    time: integer  // Actual travel time considering traffic
    variation: integer  // Variation factor for traffic

    method ChangeTraffic():
        time = Basetime + (1 + trafficCalc(hour) * variation)
```

- **Nodes (Cities):** Represented using the City class, each containing a list of edges connecting it to other cities.

```
class City:
    edges: vector of Edge  // vector of edges (roads) from this city
    name: string  // Name of the city

    method add_edge(end, time, variation):
        create new Edge instance with given end, time, and variation
        call ChangeTraffic() on the new Edge
        add the Edge to edges vector
```

- **Graph (World):** Managed by the World class, which maintains a list of cities and supports methods for adding cities, roads, and running algorithms.

```
class World:
    cities: list of City  // List of cities
    cities_dictionary: map of string to integer  // Maps city names to their index
    city_count: integer  // Number of cities

    constructor:
        city_count = 0

    method add_city(name):
        create new City instance with given name
        add City to cities list
        add mapping from name to index in cities_dictionary
        increment city_count

    method add_edge(start, end, time, variation):
        find start city and add edge to end city
        find end city and add edge to start city (undirected graph)
```

These structures allowed us to efficiently manage the network of towns and routes that would make up our world (Massachusetts).

4.2 Algorithms Used

Dijkstra's Algorithm Purpose: To calculate the shortest path from a source to a destination while considering traffic conditions.

Key Steps:

1. Initialize shortest times for all cities as infinity, except the source (set to 0).
2. Use a priority queue to explore nodes with the smallest current time.
3. Update times and paths for adjacent nodes if shorter paths are found.

Time Complexity (see subsection 4.4 for detailed analysis):

- Best case: $\Omega(E \log V)$
- Average case: $\Theta(E \log V)$
- Worst-case: $O(E \log V)$

Graph creation for nodes and edges has a complexity of $O(V+E)$ while each traffic calculation for edges is a constant-time operation $O(1)$. This makes the overall performance dependent on the graph structure and edge distribution.

Traffic Calculation Traffic variations were modeled using polynomial functions based on the time of day, simulating rush hour conditions.

Time Complexity: $O(n)$

Reason: The time complexity for the traffic adjustments is $O(n)$ where n represents the number of edges in the graph.

4.3 Visualization

A `.dot` file was generated for graph visualization, with shortest paths highlighted using red arrows.

Procedure:

1. The graph was represented in the DOT language for visualization.
2. GraphViz library was used to convert the DOT file into a visual graph.
3. Highlighted the optimal path in red for better user understanding.

Time Complexity: $O(n)$

Reason: The time complexity for the `.dot` file creation is $O(n)$ where n represents the number of edges in the graph.

4.4 Time Complexity Analysis for Functions

Pictured below is our time complexity for our Dijkstra's algorithm implementation, it matches the most optimal complexity for the algorithm's implementation, estimating when there'll be heavy traffic depending on the time of day.

1. **trafficCalc Function**
   Functionality: Computes a traffic multiplier based on the hour of the day using piecewise polynomials.
   Time Complexity: $\mathcal{O}(1)$
   Reason: The function uses a constant number of operations, regardless of the input size.

2. **edge::ChangeTraffic Function**
   Functionality: Updates the travel time of a road based on its base time, traffic multiplier, and variation.
   Time Complexity: $\mathcal{O}(1)$

Reason: This function performs a constant number of arithmetic operations.

3. **City::add_edge Function**
   Functionality: Adds a road (edge) to a city and updates its traffic time.
   Time Complexity: $\mathcal{O}(1)$
   Reason: The function creates a new edge and pushes it to the city's vector of edges. Both operations have constant time complexity.

4. **World::add_city Function**
   Functionality: Adds a new city to the world and updates data structures.
   Time Complexity: $\mathcal{O}(1)$
   Reason: Adding a city to a vector and a map has constant average time complexity.

5. **World::add_edge Function**
   Functionality: Adds a road between two cities by calling `City::add_edge` twice.
   Time Complexity: $\mathcal{O}(1)$
   Reason: Each `add_edge` call is $\mathcal{O}(1)$, and there are two of those calls.

6. **World::findMin Function**
   Functionality: Finds the city with the smallest travel time that has not been finalized.
   Time Complexity: $\mathcal{O}(V)$, where V is the number of cities.
   Reason: The function iterates over all cities to find the minimum value.

7. **World::dijk Function (Dijkstra's Algorithm)**
   Functionality: Computes the shortest path from a source city to a destination city.
   Time Complexity: $\mathcal{O}(V^2 + E)$, where V is the number of cities, and E is the number of edges (roads).
   Details:

   - The outer loop runs V times (once for each city).
   - Within the loop, `findMin` is called $\mathcal{O}(V)$.
   - The inner loop iterates over all edges of the current city ($\mathcal{O}(E/V)$ on average).

   In total, this results in $\mathcal{O}(V^2 + E)$.

8. **World::outputGraphToDotFile Function**
   Functionality: Generates a `.dot` file representing the graph and highlights the optimal path.
   Time Complexity: $\mathcal{O}(V + E)$
   Reason: Iterates over all cities (V) and their edges (E) to construct the graph representation.

9. **World::updateTraffic Function**
   Functionality: Updates traffic times for all roads based on the current hour.
   Time Complexity: $\mathcal{O}(E)$, where E is the total number of edges.
   Reason: Iterates through all edges in the graph and calls `ChangeTraffic` $\mathcal{O}(1)$ for each.

10. **Main Function**
    Functionality: Integrates all methods to build the world, update traffic, calculate routes, and output results.
    Key Calls:

    - City and Edge Initialization: Adding V cities: $\mathcal{O}(V)$. Adding E edges: $\mathcal{O}(E)$.
    - Traffic Update: $\mathcal{O}(E)$.
    - Dijkstra's Algorithm: $\mathcal{O}(V^2 + E)$.
    - Graph Output: $\mathcal{O}(V + E)$.

    Total Time Complexity: $\mathcal{O}(V^2 + E)$, dominated by Dijkstra's algorithm.

Overall Complexity: For a graph with V cities and E roads:

- Graph Construction: $\mathcal{O}(V + E)$.
- Traffic Update: $\mathcal{O}(E)$.
- Shortest Path Calculation: $\mathcal{O}(V^2 + E)$.
- Graph Output: $\mathcal{O}(V + E)$.
- **Overall:** $\mathcal{O}(V^2 + E)$.

5 Results

5.1 Findings

The program successfully avoided high traffic routes during rush hours if it saved time for the user. By determining the optimal route while considering traffic, our modified version of Dijkstra's algorithm could find alternate routes. We gathered data on the average commute time (as multipliers of commute times with no traffic conditions) with and without traffic analysis to determine the time saved for users as a multiplier.

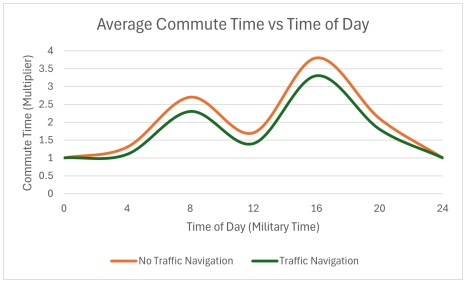

Figure 1: Average Commute Time vs Time of Day

The output graphs also accurately displayed the network of towns and routes for the user while also visually returning the determined optimal route. The execution time for Dijkstra's algorithm also remained highly efficient for larger datasets, but its limitations could not be tested due to the constraints of the graphing algorithm.

5.2 Test Cases

Before implementing our main code we tested the results of our functions and methods.

5.2.1 No Path

The first case that we tested was if there was no path from the source to the destination city. The results of this can be seen below, Dijkstra's finds City3 to have a commute time of INTMAX so it prints that there is not path from the source to the destination (see Figure 2).

5.2.2 Same Source and Destination

The second Case that we tested was if the input source and destination were the same city. In this case, Dijkstra's algorithm doesn't run and the program outputs that the source and destination are the same (see Figure 3).

```
Test Case 1:
City1               0
City2               10
City3               2147483647
There is no path from source to destination.
```

Figure 2: No Path Output

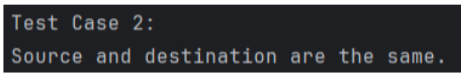

Figure 3: Same Source and Destination Output

Figure 4: Additional Nodes Output

5.2.3 Additional Nodes

The third case that we tested was testing the functionality of the Dijkstra's algorithm ensuring that the shortest path was taken even if more nodes were visited. The results can be seen below (see Figure 4).

5.2.4 Different Times

The fourth test case we used was for calculating the scaling values for traffic at different times. The results of this can be seen below with increased time during hours closer to peak (see Figure 5).

5.2.5 Rerouting

The final test case that we had tested to see if Dijkstra's algorithm would change routes based on traffic causing a different round to be faster. The algorithm did adjust to the traffic as seen below (see Figure 6).

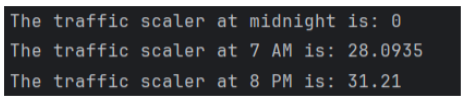

Figure 5: Different Times Output

Figure 6: Rerouting Output

6 Discussion

6.1 Implications

This project demonstrated the effectiveness of Dijkstra's algorithm in real-time navigation systems, even with added traffic constraints. Our results proved that traffic-dense routes would often be avoided at peak rush hours as expected, in favor of less traffic-dense routes that would require longer travel times under normal conditions. This outcome aligns with our expectations and demonstrates that our modified version of Dijkstra's algorithm works successfully in navigating traffic conditions.

Our output graph also demonstrates our success in being able to effectively communicate routes to our users, as clear visualization of cities and roads in addition to a visible determined route allow for an accessible and intuitive user interface. Overall, the interface came out as we hoped and the entirety of the frontend program met our hopes and expectations.

While the algorithm worked well for static traffic patterns, real-time data integration would further enhance accuracy.

6.2 Limitations

Overall, the traffic data that was used for this project was relevant and statistical, but a lack of real-time data restricted our ability to be able to provide real-time predictions of optimal routes and their navigation times. All traffic variation functions were based on approximations from data that was gathered from traffic studies in Massachusetts, and are not able to reflect real-time conditions.

In addition, our unidirectional Dijkstra's Algorithm implementation is optimal for our medium-sized graph, but does not scale well for additional nodes and routes. If the project were to be scaled to represent the entirety of Massachusetts' towns and roads, in addition to turns and ramps as nodes, then a priority queue could not be used and would require another data set.

7 Conclusion

Overall, this project was a success as we were able to successfully implement a traffic navigation system that could calculate and determine the shortest route for vehicles provided a network of cities and a data collection of traffic throughout the day. Learning how to implement the fundamentals of Dijkstra's Algo-

rithm in addition to the utilization of traffic data and navigation was a great learning experience. Furthermore, By integrating data structures and algorithms with real-world traffic scenarios, this system highlighted the practical applications of Dijkstra's algorithm in solving real-world problems. For future work, being able to implement live data for traffic would result in even greater accuracy for traffic navigation calculations.

8 References

GeeksforGeeks. (2024, August 6). Find shortest paths from source to all vertices using Dijkstra's algorithm. GeeksforGeeks. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
Klein, A. (2023, June 1). Boston traffic is up. here's how much and when Highways Clear. NBC Boston.

A Appendix A: Code

A.0.1 World, City and Edge Classes

```
    //Include Necessary Libraries For Our Project
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <climits> // For INT_MAX
#include <cstdlib> // For rand()
#include <ctime>   // For srand()
#include <math.h>
#include <fstream> // For .dot graph file generation
#include <iomanip>


using namespace std;


//Initialize global variable hour which is used to hold time of travel
double hour;


//Function to calculate traffic multiplier based off of time of commute
double trafficCalc() {
    if (hour >=5.5 && hour <=10.5) {
     return (0.277778*(pow(hour, 3))-10.2976*(pow(hour,2))+108.948*hour-325.238);
```

```
    }
    else if (hour >= 12.5 && hour <=20.5) {
        return 0.353535*(pow(hour,3))-23.5119*(pow(hour,2))+482.682*hour-3045.95;
    }
    else {
        return 0;
    }
}


//Struct that acts as a road, holds information about travel time and end of the road
struct edge {
    int end;
    int Basetime;
    int time;
    int variation;


    //Funtion to calculate travel time of road using traffic calculation and variation
    void ChangeTraffic() {
        time = Basetime * (1+trafficCalc()*variation*.002);
        //cout<<Basetime<<"    "<<time<<endl;
    }
};


//Class that holds city data. Has information on roads that connect to it and the city's nam
class City {
public:
    vector<edge> edges = {};
    string name;
    //Used to add an edge, or road, between cities
    void add_edge(int end, int time, int variation) {
        edge New;
        New.end = end;
        New.Basetime = time;
        New.variation = variation;
        New.ChangeTraffic();
        edges.push_back(New);
    }
};
//Class that holds vector of cities, and methods for routing and graph visulization
class World {
private:
    vector<City> cities;
    map<string, int> cities_dictionary;
```

```
    int city_count;


public:
    vector<string> cityNames;
    World() {
        city_count = 0;
    }
    //Method to add a city to the world
    void add_city(string name) {
        City New;
        New.name = name;
        cities.push_back(New);
        cityNames.push_back(name);
        cities_dictionary[name] = city_count;
        city_count += 1;
    }
    //Method to add a road/edge between two cities
    void add_edge(string start, string end, int time, int variation) {
        cities[cities_dictionary[start]].add_edge(cities_dictionary[end], time, variation);
        cities[cities_dictionary[end]].add_edge(cities_dictionary[start], time, variation);
    }
    //Method to find the city with lowest time to get to that hasn't been finalized
    int findMin(const vector<int>& finalized, const vector<int>& times) {
        int minIndex = -1;
        int minTime = INT_MAX;
        for (int i = 0; i < city_count; i++) {
            if (!finalized[i] && times[i] < minTime) {
                minTime = times[i];
                minIndex = i;
            }
        }
        return minIndex;
    }
    //Method that runs dijkstras algorithm on the graph
    vector<int> dijk(string sourceS, string destinationS) {
        int source = cities_dictionary[sourceS];
        int destination = cities_dictionary[destinationS];


        vector<int> finalized(city_count, 0);
        vector<int> times(city_count, INT_MAX);
        vector<int> parent(city_count, -1);


        times[source] = 0;
```

```
    int current;
    for (int i = 0; i < city_count; i++) {
        current = findMin(finalized, times);
        if (current == -1) break; // All reachable nodes are finalized
        finalized[current] = 1;


        for (size_t x = 0; x < cities[current].edges.size(); x++) {
            if (!finalized[cities[current].edges[x].end] && times[cities[current].edges[
                times[cities[current].edges[x].end] = times[current] + cities[current].e
                parent[cities[current].edges[x].end] = current;
            }
        }
    }


    for (int i = 0; i < city_count; i++) {
        cout << left << setw(20) << cityNames[i] << setw(10) << times[i] << endl;
    }
    vector<int> path;
    for (int position = destination; position != -1; position = parent[position] ) {
        path.push_back(position);
    }
    reverse(path.begin(), path.end());
    return path;
}


//Method creates DotFile that can be used with GraphViz to create a visual representatio
void outputGraphToDotFile(const vector<int>& path) {
    ofstream dotFile("graph.dot");
    dotFile << "graph DijkstraGraph {\n";


    // Define the edges in the graph
    for (size_t i = 0; i < cities.size(); ++i) {
        for (const auto& edge : cities[i].edges) {
            if(edge.end > i) {
                dotFile << "  " << cityNames[i] << " -- " << cityNames[edge.end] << " [l
            }
            }
    }
```

```
        // Highlight the optimal path
        dotFile << "\n  // Highlight the optimal path\n";
        for (size_t i = 1; i < path.size(); ++i) {
            dotFile << "  " << cityNames[path[i - 1]] << " -- " << cityNames[path[i]] << " [
        }


        dotFile << "}\n";
    }


    //Used to update the time of each road based on the time of commute
    void updateTraffic() {


        for (size_t i = 0; i < cities.size(); ++i) {
            for (size_t x = 0; x < cities[i].edges.size(); ++x) {
                //cout<<"City"<<i<<endl;
                cities[i].edges[x].ChangeTraffic();
            }
        }
    }


};
```

A.1 Main Function with Test Cases

```
int main() {
hour = 18;
srand(static_cast<unsigned int>(time(0)));
//Create a world
World USA;
//Add all cities to world
USA.add_city("Boston");
USA.add_city("Salem");
USA.add_city("Burlington");
USA.add_city("Newton");
USA.add_city("Norwood");
USA.add_city("Quincy");
USA.add_city("Lawrence");
USA.add_city("Lowell");
USA.add_city("Marlborough");
USA.add_city("Mansfield");
USA.add_city("Plymouth");
```

```
USA.add_city("Littleton");
USA.add_city("Leominster");
USA.add_city("Worcester");
USA.add_city("Providence");
USA.add_city("Orange");
USA.add_city("Greenfield");
USA.add_city("Springfield");
//Add roads/edges between cities
USA.add_edge("Boston","Salem",26, 5);
USA.add_edge("Boston","Burlington",24, 5);
USA.add_edge("Salem","Burlington",30, 4);
USA.add_edge("Boston","Newton",17, 5);
USA.add_edge("Burlington","Newton",26, 4);
USA.add_edge("Boston","Norwood",30, 5);
USA.add_edge("Newton","Norwood",23, 4);
USA.add_edge("Boston","Quincy",15, 5);
USA.add_edge("Norwood","Quincy",20, 4);
USA.add_edge("Salem","Lawrence",40, 3);
USA.add_edge("Burlington","Lawrence",24, 4);
USA.add_edge("Lawrence","Lowell",19, 3);
USA.add_edge("Burlington","Lowell",17, 4);
USA.add_edge("Newton","Marlborough",30,4);
USA.add_edge("Lowell","Marlborough",30,3);
USA.add_edge("Norwood","Mansfield",18,4);
USA.add_edge("Marlborough","Mansfield",36,3);
USA.add_edge("Quincy","Plymouth",32,3);
USA.add_edge("Mansfield","Plymouth",42,3);
USA.add_edge("Lowell","Littleton",17,3);
USA.add_edge("Marlborough","Littleton",20,3);
USA.add_edge("Littleton","Leominster",19,3);
USA.add_edge("Leominster","Worcester",26,2);
USA.add_edge("Marlborough","Worcester",22,3);
USA.add_edge("Worcester","Providence",45,3);
USA.add_edge("Mansfield","Providence",27,4);
USA.add_edge("Plymouth","Providence",57,3);
USA.add_edge("Leominster","Orange",37,2);
USA.add_edge("Orange","Greenfield",26,2);
USA.add_edge("Orange","Springfield",59,1);
USA.add_edge("Greenfield","Springfield",38,3);
USA.add_edge("Worcester","Springfield",54,3);




//Test Case 1: Same source and destination
/*vector<int> record = USA.dijk("Worcester","Worcester");
```

```
if (record.size() != 1) {
    for (size_t i = 0; i < record.size(); i++) {
        cout << USA.cityNames[record[i]];
        if (i < record.size() - 1) cout << " -> ";
    }
}
else {
    cout<<"The source and destination cities are the same so a travel time can't be calc
}
*/


//Test Case 2: With Zero traffic we would expect time from Boston to Salem to be 26
/*
hour = 0;
vector<int> record = USA.dijk("Boston","Salem");
if (record.size() != 1) {
for (size_t i = 0; i < record.size(); i++) {
cout << USA.cityNames[record[i]];
if (i < record.size() - 1) cout << " -> ";
}
}
else {
cout<<"The source and destination cities are the same so a travel time can't be calculat
}
*/







//List all available cities in the world and prompts user for inputs
string source, destination;
    cout<<"This program allows you to calculate the time, and best route to take betwee
    cout<<"Below is a list of cities that are included in this program:"<<endl;
    cout<<"Boston      Salem      Burlington    Newton     Norwood      Quincy"<<endl;
    cout<<"Lawrence    Lowell     Marlborough   Mansfield  Plymouth     Littleton"<<endl
    cout<<"Leominster  Worcester  Providence    Orange     Greenfield   Springfield"<<en


    cout<<"Please enter the city that you are currently at (Exactly As Seen Above): ";
    cin>>source;
    cout<<"Please enter the city that is your destination (Exactly As Seen Above): ";
```

```
        cin>>destination;
        cout<<"Please Enter an Integer (0-24) Representing Time of Travel: ";
        cin>>hour;
        USA.updateTraffic();
    vector<int> record = USA.dijk(source,destination);


    //Output optimal path
    if (record.size() != 1) {
        for (size_t i = 0; i < record.size(); i++) {
            cout << USA.cityNames[record[i]];
            if (i < record.size() - 1) cout << " -> ";
        }
    }
    else {
        cout<<"The source and destination cities are the same so a travel time can't be calc
    }


    //Convert graph into a dot file that can be turned into a visualization
    USA.outputGraphToDotFile(record);







    return 0;


}
```

B Appendix B: Additional Figures

output > graph.dot

```
graph DijkstraGraph {
  Boston -- Salem [label="26"];
  Boston -- Burlington [label="24"];
  Boston -- Newton [label="17"];
  Boston -- Norwood [label="30"];
  Boston -- Quincy [label="15"];
  Salem -- Burlington [label="30"];
  Salem -- Lawrence [label="40"];
  Burlington -- Newton [label="26"];
  Burlington -- Lawrence [label="24"];
  Burlington -- Lowell [label="17"];
  Newton -- Norwood [label="23"];
  Newton -- Marlborough [label="30"];
  Norwood -- Quincy [label="20"];
  Norwood -- Mansfield [label="18"];
  Quincy -- Plymouth [label="32"];
  Lawrence -- Lowell [label="19"];
  Lowell -- Marlborough [label="30"];
  Lowell -- Littleton [label="17"];
  Marlborough -- Mansfield [label="36"];
  Marlborough -- Littleton [label="20"];
  Marlborough -- Worcester [label="22"];
  Mansfield -- Plymouth [label="42"];
  Mansfield -- Providence [label="27"];
  Plymouth -- Providence [label="57"];
  Littleton -- Leominster [label="19"];
  Leominster -- Worcester [label="26"];
  Leominster -- Orange [label="37"];
  Worcester -- Providence [label="45"];
  Worcester -- Springfield [label="54"];
  Orange -- Greenfield [label="26"];
  Orange -- Springfield [label="59"];
  Greenfield -- Springfield [label="38"];

  // Highlight the optimal path
  Boston -- Burlington [color=red, penwidth=3.0];
  Burlington -- Lowell [color=red, penwidth=3.0];
  Lowell -- Littleton [color=red, penwidth=3.0];
  Littleton -- Leominster [color=red, penwidth=3.0];
  Leominster -- Orange [color=red, penwidth=3.0];
  Orange -- Greenfield [color=red, penwidth=3.0];
}
```

Figure 7: List of Cities

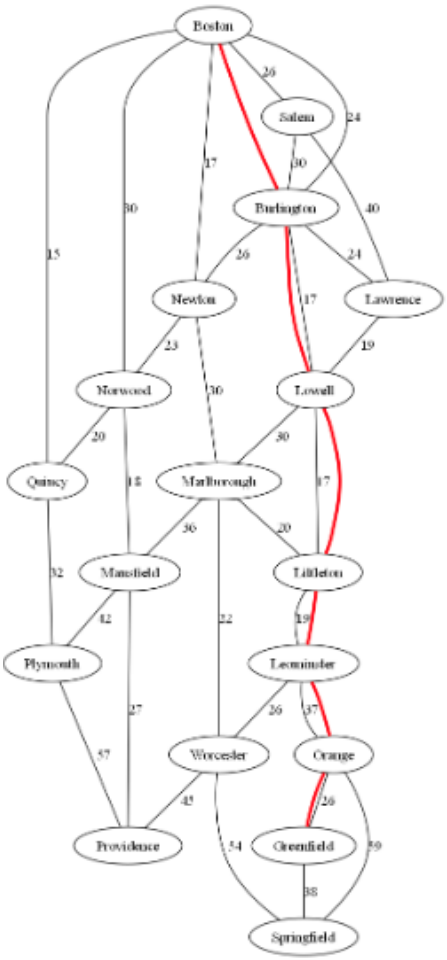

Figure 8: Map Graph Visual

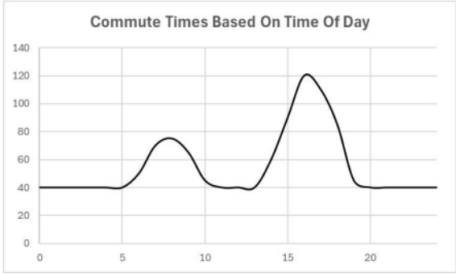

Figure 9: Commute Times Based on Day

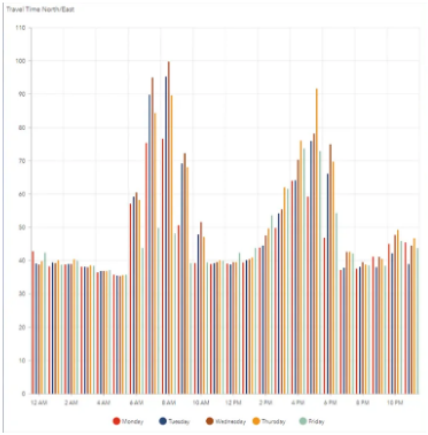

Figure 10: Traffic Data of Massachusetts