Traffic Navigation System

EECE2560

Will Jappe, John Walton and Ryder Robbins

Introduction

Scope

The scope of this project is to design and develop a real time traffic navigation system that can determine the <u>shortest route</u> for vehicles in a state network while considering the <u>time</u> of day and <u>traffic</u>.

Objectives

- Track a state network for possible routes
- Determine levels of traffic from the time of day
- Determine the shortest route from this information
- Graph the data and output via graphing libraries

Literature Review

Dijkstra’s Algorithm is commonly used to determine the shortest possible **route** provided a network of **nodes** (locations or intersections) and **edges** (connections or roads).

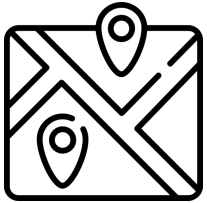
Dijkstra’s Algorithm is utilized in a number of applications such as Apple Maps, Google Maps, Ways, etc.

Dijkstra’s Algorithm can be modified to support particular scenarios, such as **traffic**.

Methodology

Route Determination

Calculating traffic, finding length of possible routes, and determining shortest route



Map Visualization

Plotting cities and connections while also visualizing traffic and routes

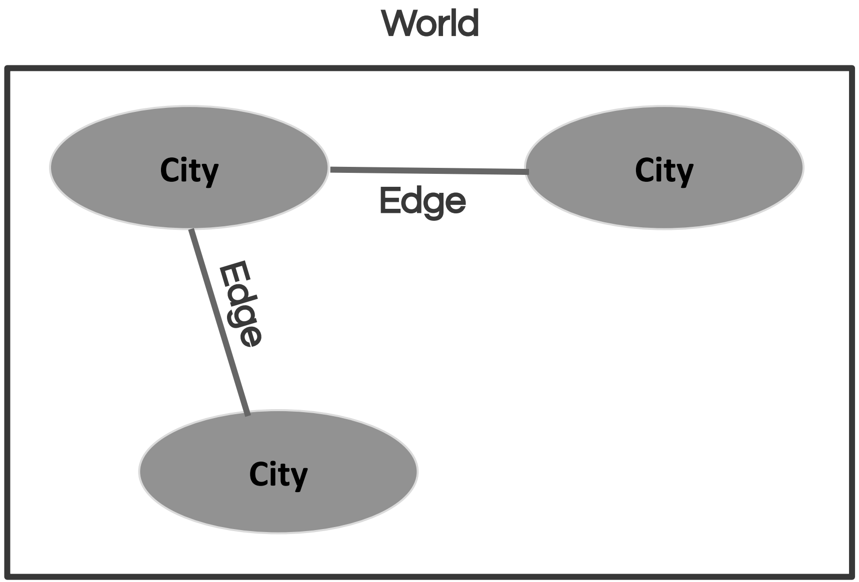
Methodology: Graph Structure Used

Three Data Structures make up the graph.

Struct Edge - Represents Connections Between Cities

Class City - Holds City Name and Vector of Edges

Class World - Holds Vector of Cities, and holds contains methods for Graph Visualization and Dijkstra's Algorithm



Methodology: World Class and City Management

The World class manages cities and edges, allowing for adding cities and edges to the graph.

Pseudocode:

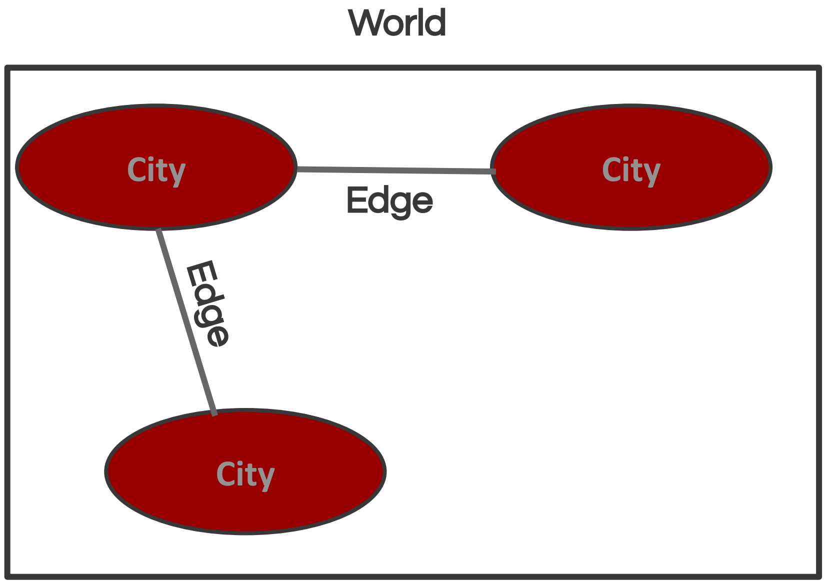
```
class World:
   cities: list of City  // List of cities
   cities_dictionary: map of string to integer  // Maps city names to their index
   city_count: integer  // Number of cities

   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)q1
```

Methodology: City Class

This class models a city with a list of connecting edges (roads).

Pseudocode:

```
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
```

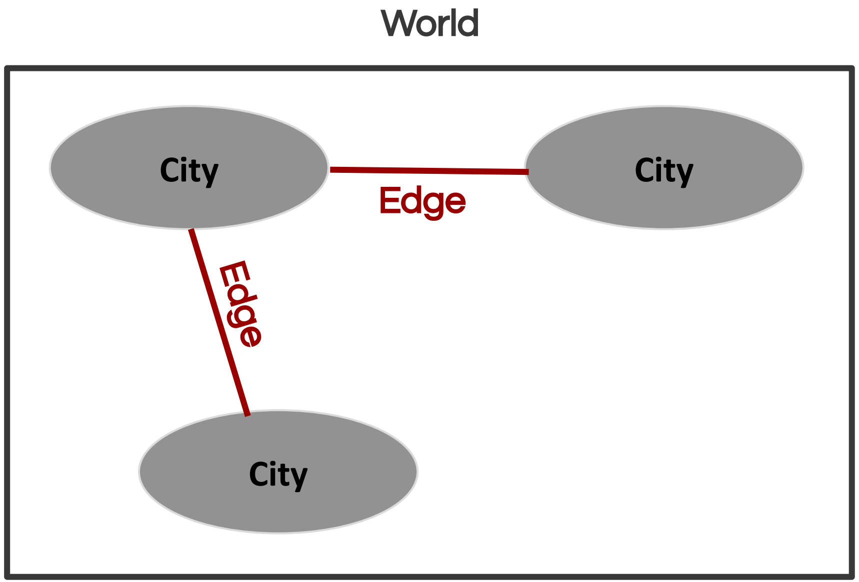


Methodology: Edge Structure and Traffic Change

This structure represents an edge (road) between two cities, with properties for base travel time and actual travel time modified by a traffic factor.

Pseudocode:

```
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)
```

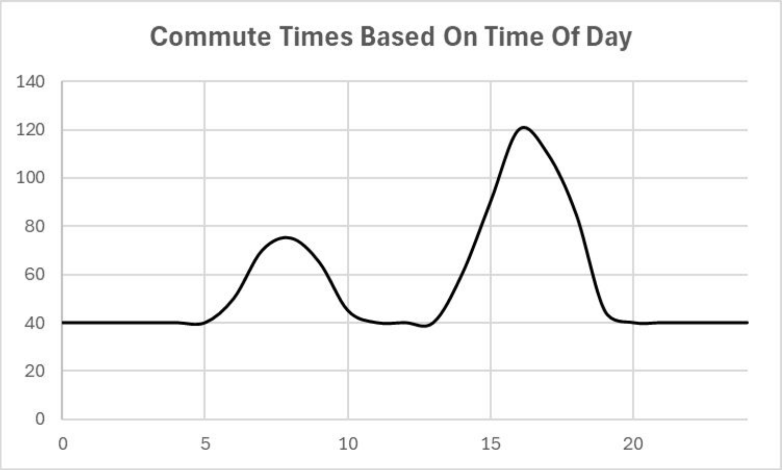


Methodology: Traffic Calculation Function

This function calculates the traffic variation based on the current hour using polynomial equations to simulate traffic patterns during peak hours.

Pseudocode:

```
function trafficCalc(hour):
    if hour is between 5.5 and 10.5:
        return (polynomial function of hour for morning traffic)
    else if hour is between 12.5 and 20.5:
        return (polynomial function of hour for afternoon traffic)
    else:
        return 0
```

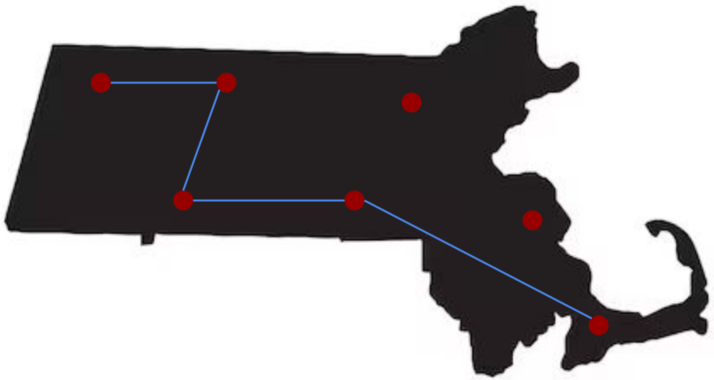


Methodology: Dijkstra's Algorithm for Shortest Path

This method implements Dijkstra's algorithm to find the shortest path between the source and destination.

Pseudocode:

```
method dijk(source, destination):
  finalized: list of integer initialized to 0  // Keeps track of finalized nodes
  times: list of integer initialized to infinity  // Shortest times to each node
  parent: list of integer initialized to -1  // Tracks parent nodes for path reconstruction

  set times[source] = 0  // Source node distance is 0

  for i from 0 to city_count:
    current = findMin(finalized, times)  // Find the node with minimum time
    if current is -1:
      break  // All reachable nodes are finalized

    set finalized[current] = 1  // Mark current node as finalized

    for each edge in cities[current].edges:
      if edge end node is not finalized and new path time is less than existing time:
        update time to the edge end node
        update parent of the edge end node

  construct path from destination to source using parent array
  return path
```

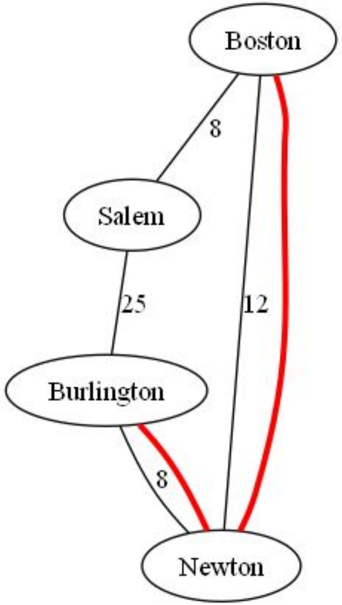


Methodology: Graph Output to DOT File

This method generates a .dot file for visualizing the graph and highlights the optimal path directional arrows in red.

Pseudocode:

```
method outputGraphToDotFile(path):
    create new .dot file
    write "graph DijkstraGraph {"  // Begin graph definition

    for each city in cities:
        for each edge in city.edges:
            if edge is not a duplicate (consider direction):
                write city to edge connection with travel time label

    for each pair of nodes in path:
        write directional arrow between nodes in path with red color and increased penwidth

    write "}"  // End graph definition
```



Methodology: Main Program

Time Complexity:
The primary algorithm used is Dijkstra’s algorithm, which has a time complexity of O(Elog(V)) where E is the number of edges and V is the number of vertices (cities).

- Best case: Ω(E*logV)
- Average case: Θ(E*logV)
- Worst-case: O(E*logV)
- 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.

Analysis and Results - Time Table

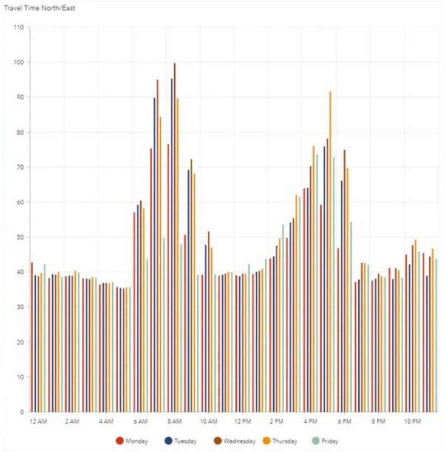
6 Hours / Week	2 Hours / Week	2 Hours / Week
Implementing algorithms	Refining code and testing results	Documenting progress and presentation work

Total: 50 Hours

Analysis and Results - Findings

When testing our program, results showed that the busiest routes (with higher traffic variation factors) would be **avoided** at busier times as expected.

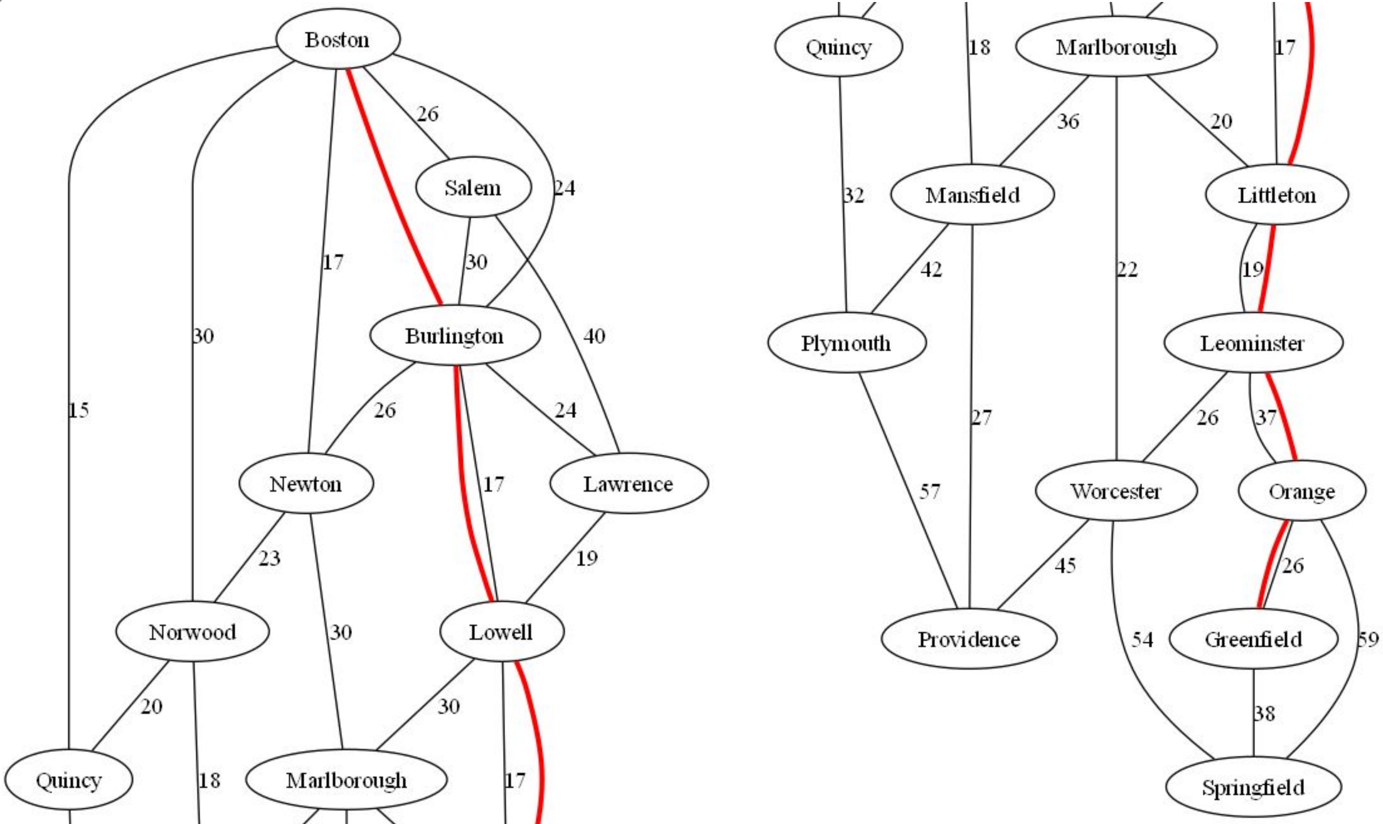
It is important to note that our results are **unique** to the base travel times and traffic data that we utilized for this specific application.



Traffic Data for I-90 East

- dot -Tpng graph.dot -o graph.png

Analysis and Results - Findings



Discussion

Implications of findings - Over the course of working on this project we were exposed to algorithms that are used for routing. While we were able to make a system that worked, the project highlighted the complexity of navigation apps that people use daily.

Project limitations - While this project does integrate a feature of traffic management, and tries to account for traffic variations it is not able to use real time data.

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 Algorithm in addition to the utilization of traffic data and navigation was a great learning experience.

For future work, being able to implement live data for traffic would result in even greater accuracy for traffic navigation calculations.

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. https://www.nbcboston.com/news/local/boston-traffic-is-up-heres-how-much-and-when-highways-clear/3058575/