Time & Cost Optimization of Life at Northeastern via Graph-Based Modeling of NEU Boston Campus

Matthew Gregorio, Arushi Gupta, Ikonkar Khalsa, Jonathan Ockert
CS 5800 Fall Final Project
Khoury College of Computer Sciences
Northeastern University
Boston, MA 02115

December 11, 2024

Abstract

This project explores the application of graph theory in modeling navigation and decision-making on a university campus, with a focus on optimizing travel times and costs. Two directed graphs are constructed: a Time Graph (G_t) that represents estimated travel times between various campus locations, and a Cost Graph (G_c) that models the monetary cost of traversing these paths. The project employs both the Bellman-Ford and Dijkstra's algorithms for pathfinding, each chosen based on their ability to handle specific edge cases. Bellman-Ford is applied to graphs with potential negative edge weights, while Dijkstra's algorithm is leveraged for efficient analysis in sparse graphs with positive edge weights. The project also addresses the implementation of these algorithms with respect to real-world data, such as campus building access and costs incurred at various campus locations. Ultimately, the work aims to optimize campus navigation, providing students with the shortest and most cost-effective paths for daily travel.

1 Motivation

The motivation for this project arises from the common challenges faced by full-time students at Northeastern in managing both time and cost while navigating the campus. With a focus on students living on a budget, finding the most efficient and affordable routes is crucial for improving daily life. Whether it's selecting the fastest path to class or the most cost-effective route to nearby amenities, optimized solutions can make campus navigation more efficient and budget-friendly. This project aims to address these needs by applying graph-based modeling to develop algorithms that help students identify the best routes based on specific factors like time, distance, and cost.

2 Goals

The primary goal of this project is to develop an algorithm capable of identifying the optimal routes on the Northeastern campus, based on specific needs such as minimizing time, distance, or cost. The project will address these objectives through two distinct optimization approaches.

For time and distance optimization, the project will primarily implement Dijkstra's algorithm, which is well-known for its efficiency in finding the shortest path between nodes in a graph with non-negative edge weights. This algorithm will serve as the foundation for comparing other potential shortest-path algorithms to evaluate their performance in terms of computational complexity and practical application. By exploring alternative algorithms, the project aims to assess which method provides the best balance of speed and accuracy for real-world campus navigation.

For cost optimization, the project will focus on paths that involve negative edge weights, which can represent scenarios such as discounts, incentives, or penalties (e.g., using certain routes that may require a fee or offer rewards). To handle these negative weights, the project will implement the Bellman-Ford algorithm, known for its ability to efficiently find the shortest path even in the presence of negative edge weights. Additionally, the team will explore the use of Johnson's algorithm, which is capable of solving all-pairs shortest path problems and is useful when dealing with a wide range of edge weights in a large graph. This dual approach will ensure that the algorithm can not only find the shortest paths but also handle cost factors that may impact route selection.

By combining these techniques, the project aims to offer a comprehensive solution that caters to both time-sensitive and cost-conscious students. Ultimately, the goal is to provide a practical, data-driven resource that helps students navigate the campus in the most efficient and affordable way possible. This will not only improve the daily campus experience but also demonstrate the real-world application of graph theory and algorithm design in optimizing everyday decisions.

3 Introduction

To effectively model and optimize campus routes at Northeastern University, the project followed a structured methodology that began with identifying key locations across the campus. First, Google Maps was used to pinpoint approximately 10 major locations, both on and near campus, that are frequently visited by students. Once these locations were identified, the next step involved verifying accessibility, ensuring that paths through specific buildings or shortcuts were viable based on access restrictions, such as card-controlled entry points.

The third step involved mapping campus amenities, stores, and potential shortcuts that could impact route optimization. By utilizing the Northeastern Boston campus map, a clearer picture of both the physical layout and possible shortcuts was formed. Finally, a thorough analysis was conducted using various graph-based algorithms, including Dijkstra's and Bellman-Ford, to calculate the best routes based on time, distance, and cost.

These steps provided the foundation for accurately modeling the campus and ensuring that the resulting algorithms offered practical, data-driven solutions for navigating the campus efficiently.

4 Research

First, we conducted research by using Google Maps to identify 10 major hubs on Northeastern's Boston Campus. This was used to determine how much time it would take to walk from one location to another.

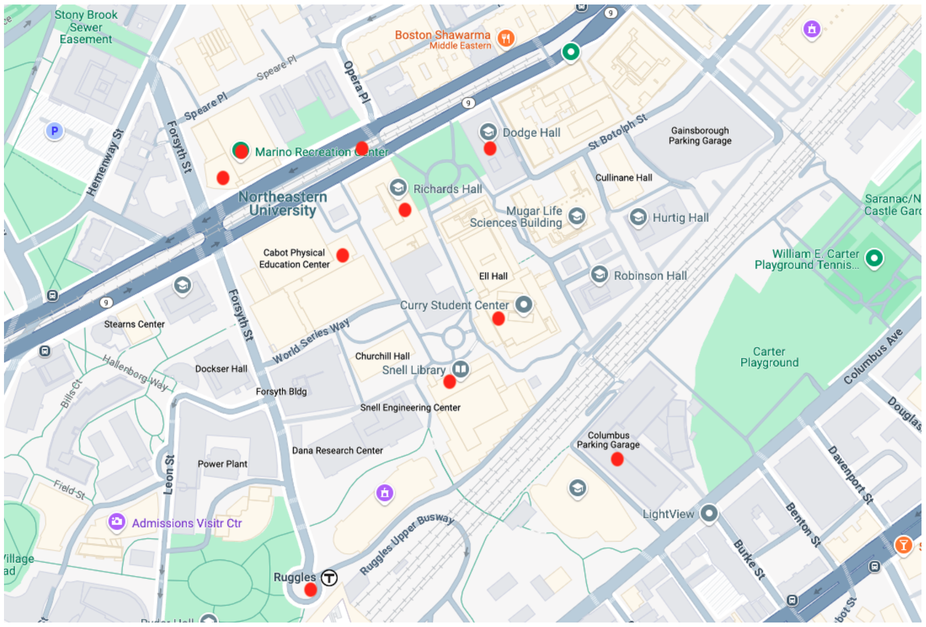


Figure 1: Map of Northeastern's Boston Campus Marked with Vertices

The Northeastern Boston Campus map was retrieved from Google Maps [1].

The following are the 10 major hubs where we will find edges we can travel through and an algorithm to go in between vertices:

1. Ruggles T stop
2. Northeastern T Stop

3. Columbus Parking Garage
4. Marino Recreation Center
5. Cabot Pysical Education Center
6. Curry Student Center
7. Tatte on Huntington
8. Richards Hall
9. Dodge Hall
10. Snell Library

Each location will be labeled as a vertices which will be essential in differentiating between journeys:

Vertices	Location
v_1	Ruggles T stop
v_2	Northeastern T Stop
v_3	Columbus Parking Garage
v_4	Marino Recreation Center
v_5	Cabot Physical Education Center
v_6	Curry Student Center
v_7	Tatte on Huntington Ave
v_8	Richards Hall
v_9	Dodge Hall
v_{10}	Snell Library

Table 1: Label Vertices to its Location

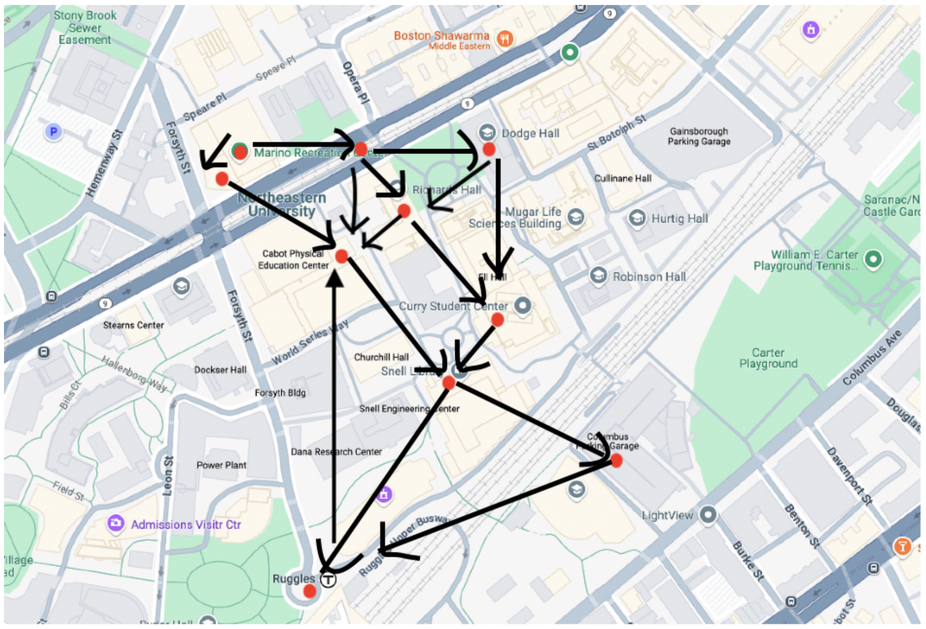


Figure 2: Directed Graph with Vertices

Potential Cost Breakdown:

Everyone in this group has been around the campus several times and are well aware of all the coffee shops, student stores, parking garages, and restaurants in campus. We also utilized the Google Maps [1] to check if there are more coffee shops, student stores, parking garages, etc where costs could be made.

1. **Vertex v_1:**
   - (Ruggles T Stop) $v_1 \rightarrow$ (Cabot Physical Education Center) v_5:
     - Bank of America ATM
     - Ruggles Pizza & Cafe
     - Dunkins
2. **Vertex v_2:**
   - (Northeastern T Stop) $v_2 \rightarrow$ (Cabot Physical Education Center) v_5: No Cost
   - (Northeastern T Stop) $v_2 \rightarrow$ (Richards Hall) v_8:
     - Northeastern University Arboretum
   - (Northeastern T Stop) $v_2 \rightarrow$ (Dodge Hall) v_9: No Cost
3. **Vertex v_3:**

- (Columbus Parking Garage) $v_3 \to$ (Ruggles T Stop) v_1:
  - Parking
  - Saxbys
  - Ruggles Pizza & Cafe
  - Dunkins
  - Fuel America Coffee Store

4. **Vertex v_4:**
   - (Marino Recreation Center) $v_4 \to$ (Northeastern T Stop) v_2:
     - Tatte Bakery & Cafe
     - Wollaston's Market
   - (Marino Recreation Center) $v_4 \to$ (Tatte on Huntington Ave) v_7:
     - Wollaston's Market
     - Tatte Bakery & Cafe
   - (Marino Recreation Center) $v_4 \to$ (Snell Library) v_{10}:
     - Wollaston's Market
     - Tatte Bakery & Cafe
5. **Vertex v_5:**
   - (Cabot Physical Education Center) $v_5 \to$ (Snell Library) v_{10}:
     - Campus Coffee & Tea
6. **Vertex v_6:**
   - (Curry Student Center) $v_6 \to$ (Snell Library) v_{10}:
     - The Market
     - Kigo Kitchen
     - Popeyes Louisiana Kitchen
     - Student Store
     - Starbucks
     - Campus Coffee & Tea
7. **Vertex v_7:**
   - (Tatte on Huntington Ave) $v_7 \to$ (Cabot Physical Education Center) v_5:
     - Tatte Bakery & Cafe
8. **Vertex v_8:**
   - (Richards Hall) $v_8 \to$ (Cabot Physical Education Center) v_5:
     - The Market

  - Kigo Kitchen
  - Popeyes Louisiana Kitchen
  - Student Store
  - Starbucks
  - Campus Coffee & Tea

- (Richards Hall) $v_8 \rightarrow$ (Curry Student Center) v_6:
  - The Market
  - Kigo Kitchen
  - Popeyes Louisiana Kitchen
  - Student Store
  - Starbucks
  - Campus Coffee & Tea

9. **Vertex v_9:**

   - (Dodge Hall) $v_9 \rightarrow$ (Curry Student Center) v_6:
     - The Market
     - Kigo Kitchen
     - Popeyes Louisiana Kitchen
     - Student Store
     - Starbucks
     - Campus Coffee & Tea
   - (Dodge Hall) $v_9 \rightarrow$ (Richards Hall) v_8: No Cost

10. **Vertex v_{10}:**

    - (Snell Library) $v_{10} \rightarrow$ (Ruggles T Stop) v_1:
      - Bank of America ATM
      - Ruggles Pizza & Cafe
      - Dunkins
    - (Snell Library) $v_{10} \rightarrow$ (Columbus Parking Garage) v_3:
      - Saxbys
      - Parking

5 Project Content

To model the navigation and decision-making across the vertices (locations), we constructed two distinct directed graphs:

1. **Time Graph (G_t)**: This graph represents the estimated travel times between vertices. The edge weights are calculated using realistic estimates based on travel paths, such as walking along streets or cutting through buildings. The weights are directional, meaning the time to travel from v_i to v_j may differ from the reverse route (v_j to v_i) due to path variations or obstacles. Another point we want to point out is that students have card access to several buildings which is why the time to walk around is determined based off that. Students can make several cuts throughout campus due to card access.

2. **Cost Graph (G_c)**: This graph represents the monetary cost associated with moving between vertices. The edge weights are mostly positive, indicating expenses like spending money at stores or vending machines along the way. However, some edges may have negative weights, representing opportunities to gain monetary value, such as finding loose change, receiving free samples, or utilizing coupons. Care is taken to avoid negative weight cycles to ensure compatibility with algorithms like Bellman-Ford or Johnson's algorithm.

Methodology for Edge Weights

- **Time Weights (G_t)**:
  - Based on real-world distances derived from Google Maps.
  - Adjusted for shortcuts, such as cutting through buildings or alleys, when feasible.
- **Cost Weights (G_c)**:
  - Positive weights for expenses encountered along the way.
  - Negative weights for monetary gains from specific locations (e.g., free samples, discounts).
  - A sparse graph structure minimizes unnecessary complexity and ensures a manageable number of paths.

Data Structures for Pathfinding

In practical coding, adjacency lists or priority queues are often preferred for optimal runtime in pathfinding algorithms, especially when dealing with large graphs. For this project, we decided to use an adjacency matrix to represent the graph because it's straightforward and easy to implement. However, this might not be the most efficient choice for larger, sparser graphs, where an adjacency list could be a better fit.

The following graphs provide an abstract representation of the problem space, suitable for applying graph algorithms to analyze optimal paths in terms of time and cost. The adjacency matrices for G_t and G_c are shown below. Each matrix element a_{ij} corresponds to the directed edge from vertex v_i to v_j, with the value representing the respective weight (time or cost).

Adjacency Matrix for Time (G_t) in Minutes:

$$G_t = \begin{array}{c|cccccccccc} & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} \\ \hline v_1 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ v_2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \\ v_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_5 & 4 & 3 & 0 & 0 & 0 & 0 & 4 & 3 & 0 & 0 \\ v_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 0 \\ v_7 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_8 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ v_9 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_{10} & 0 & 0 & 0 & 5 & 2 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Adjacency Matrix for Cost (G_c):

$$G_c = \begin{array}{c|cccccccccc} & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} \\ \hline v_1 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ v_2 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ v_4 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_5 & 3 & 0 & 0 & 0 & 0 & 0 & 1 & 6 & 0 & 0 \\ v_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 6 & 0 \\ v_7 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_8 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_{10} & 0 & 0 & 0 & 2 & 1 & 6 & 0 & 0 & 0 & 0 \end{array}$$

Description of Costs:

The cost in the adjacency matrix G_c is represented in dollar values, where each location or place corresponds to a unit cost of 1 dollar. We picked these places because according to our personal knowledge of where stores are, we also utilized Google Maps [1] to determine the costs. Here's the breakdown for each vertex and their connections:

1. **Vertex v_1 (Ruggles T Stop):**
   - To v_5 (Cabot Physical Education Center): The cost is 3 dollars. This covers locations such as the Bank of America ATM, Ruggles Pizza & Cafe, and Dunkins.
   - To v_3 (Columbus Parking Garage): The cost is 5 dollars. This includes costs for parking and spots like Saxbys, Ruggles Pizza & Cafe, Dunkins, and Fuel America Coffee Store.

2. **Vertex v_2 (Northeastern T Stop):**
   - To v_5 (Cabot Physical Education Center): No cost (0 dollars).
   - To v_8 (Richards Hall): Gain 1 dollar in the chance of finding a coin at the Northeastern University Arboretum.
   - To v_9 (Dodge Hall): No cost (0 dollars).

3. **Vertex v_3 (Columbus Parking Garage):**
   - To v_1 (Ruggles T Stop): The cost is 5 dollars. This reflects parking fees and places like Saxbys, Ruggles Pizza & Cafe, Dunkins, and Fuel America Coffee Store.

4. **Vertex v_4 (Marino Recreation Center):**
   - To v_2 (Northeastern T Stop): The cost is 6 dollars, covering locations like Tatte Bakery & Cafe and Wollaston's Market.
   - To v_7 (Tatte on Huntington Ave): The cost is 2 dollars, which includes Tatte Bakery & Cafe and Wollaston's Market.
   - To v_{10} (Snell Library): The cost is 5 dollars, which includes Wollaston's Market and Tatte Bakery & Cafe.

5. **Vertex v_5 (Cabot Physical Education Center):**
   - To v_{10} (Snell Library): The cost is 2 dollars, which includes Campus Coffee & Tea.

6. **Vertex v_6 (Curry Student Center):**
   - To v_{10} (Snell Library): The cost is 6 dollars, which includes a variety of dining options like The Market, Kigo Kitchen, Popeyes Louisiana Kitchen, Student Store, Starbucks, and Campus Coffee & Tea.

7. **Vertex v_7 (Tatte on Huntington Ave):**
   - To v_5 (Cabot Physical Education Center): The cost is 2 dollars, which covers Tatte Bakery & Cafe.

8. **Vertex v_8 (Richards Hall):**
   - To v_5 (Cabot Physical Education Center): The cost is 6 dollars, covering places like The Market, Kigo Kitchen, Popeyes Louisiana Kitchen, Student Store, Starbucks, and Campus Coffee & Tea.
   - To v_6 (Curry Student Center): The cost is 6 dollars, similar to the previous route with various dining options.

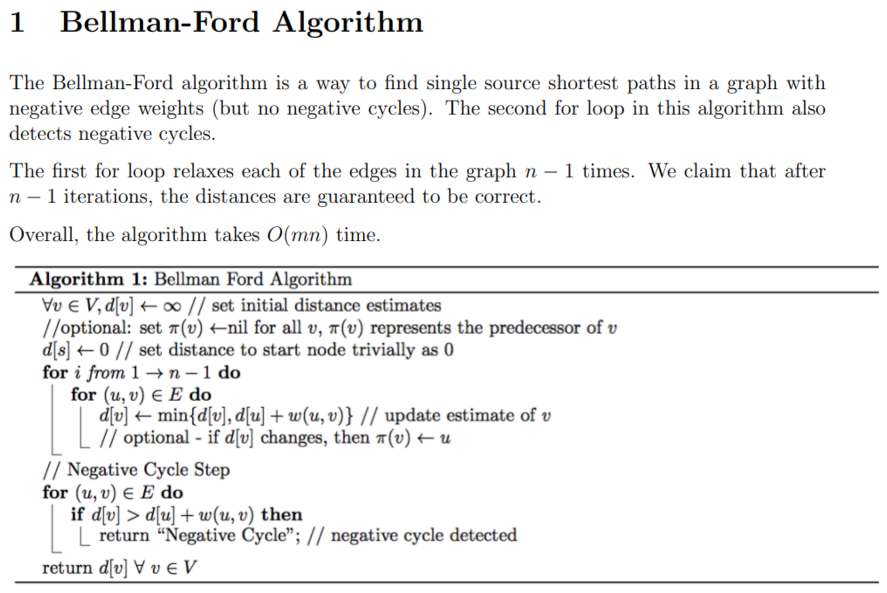
9. **Vertex v_9 (Dodge Hall):**
   - To v_6 (Curry Student Center): The cost is 6 dollars, which includes The Market, Kigo Kitchen, Popeyes Louisiana Kitchen, Student Store, Starbucks, and Campus Coffee & Tea.
   - To v_8 (Richards Hall): No cost (0 dollars).

10. **Vertex v_{10} (Snell Library):**
    - To v_1 (Ruggles T Stop): The cost is 3 dollars, which includes the Bank of America ATM, Ruggles Pizza & Cafe, and Dunkins.
    - To v_3 (Columbus Parking Garage): The cost is 2 dollars, covering Saxbys and parking.

In summary, the cost between any two vertices is given in dollar amounts, where each place (such as a cafe, ATM, or parking) is equivalent to 1 dollar in cost. The cost matrix directly reflects the costs associated with traveling between each vertex and the locations connected to it.

6 Bellman-Ford Algorithm

In order to use Bellman-Ford to find the shortest path from v_1 to v_{10} of both matrices, we need to first understand the way the algorithm traverses from source node to other nodes. Through each iteration, there is a relaxation process that happens where the path is evaluated to determine the shortest weighted path between two nodes. However, when there is a negative weight, the edge weights get adjusted following the pseudocode:



1 Bellman-Ford Algorithm

The Bellman-Ford algorithm is a way to find single source shortest paths in a graph with negative edge weights (but no negative cycles). The second for loop in this algorithm also detects negative cycles.

The first for loop relaxes each of the edges in the graph $n - 1$ times. We claim that after $n - 1$ iterations, the distances are guaranteed to be correct.

Overall, the algorithm takes $O(mn)$ time.

Algorithm 1: Bellman Ford Algorithm

```
$\forall v \in V, d[v] \leftarrow \infty$ // set initial distance estimates
//optional: set $\pi(v) \leftarrow$nil for all $v$, $\pi(v)$ represents the predecessor of $v$
$d[s] \leftarrow 0$ // set distance to start node trivially as 0
for $i$ from $1 \rightarrow n - 1$ do
    for $(u, v) \in E$ do
        $d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$ // update estimate of $v$
        // optional - if $d[v]$ changes, then $\pi(v) \leftarrow u$
// Negative Cycle Step
for $(u, v) \in E$ do
    if $d[v] > d[u] + w(u, v)$ then
        return "Negative Cycle"; // negative cycle detected
return $d[v]$ $\forall$ $v \in V$
```

Figure 3: Picture from a lecture published on Stanford's website [2].

Now let us look at what this pseudocode is showing us. We always set the initial distance to ∞ for every node. Then we set the source node distance to 0. The distances will get updated through each iteration. Now we go through the relaxation period. We repeat the relaxation step $v - 1$ times because that is the maximum number of edges that could be between any two vertices. Each relaxation phase updates the shortest weight path in each iteration. We check if the path from any vertex to another vertex can be updated further. If an update can be made, then set the distance to the lesser value. This is the part of the code that represents the relaxation part. The last part of the code is looking for negative cycles in the graph. From our two matrices, we know that the cost graph has negative weighted edges. This part of the code will identify where the negative edge is. We go through $v - 1$ iterations again, we check for a negative weight cycle, then we output "Negative Cycle" and we stop the algorithm. In our projects context, we did not make this a possibility because we can't get infinite money from picking up coins in the arboretum. The graphs were specifically designed to be practical for the everyday Northeastern student. At the end of the algorithm, we return the shortest established distances. The runtime for Bellman-Ford $O(v \cdot e)$. Because our graphs are sparse graphs, we can assume $e = v$, so the runtime is $O(v^2)$.

The reason we use Bellman-Ford to find shortest path is because it's an algorithm that can evaluate negative edge weights. In our project, there is a cost graph in where we can either pay or gain money. When we encounter such places as the arboretum in where some students like to take money out. Of course, the students involved in this project will never do such a thing, but let's pretend that we have for the sake of this graph. On page 10 of this project, you will see a very clear example in where a Bellman-Ford algorithm will be helpful because if a student was at the Northeastern T Stop v_2 and needed to go to Richards Hall v_8 then there is a negative edge weight of 1. Later on, you will see why a Bellman-Ford is not preferred in cases on only positive edge weights because it has a slower runtime than other algorithms like Dijkstra's.

7 Dijkstra's Algorithm

Dijkstra's algorithm famously finds the shortest path in a graph G from a source vertex s to all other vertices v in G, as long as G does not have negative edge weights. Starting from s, Dijkstra's initializes the distances from s to each v as distance from s to s being 0 and all other distances being ∞. The algorithm then initializes the set of visited vertices and the priority queue. The algorithm selects all vertices other than s into the priority queue without regard to order, as all distances are currently set to ∞. The algorithm then proceeds to repeatedly select the v out of Q with the shortest tentative aggregate distance from s. As we process more and more vertices, there may be instances where the relaxation step uncovers a new shortest path from s to the newly processed u (i.e., a potential value existed but u had not yet been visited so the shortest path to u had not yet been set, as each vertex is visited only once in Dijkstra's). The algorithm concludes once the priority queue is empty, leaving us with the shortest path from the source vertex s to each other vertex u.

Pseudocode for Dijkstra's Algorithm:

```
DIJKSTRA($G$, $w$, $s$)
Require: $G$ is the graph $G_t$
1: INITIALIZE-SINGLE-SOURCE($G$, $s$)
2: $S = 0$
3: $Q = 0$
4: for each vertex $u \in$ G.V do
5:     INSERT($Q$, $u$)
```

```
 6: end for
 7: while $Q$ != 0 do
 8:     u = EXTRACT-MIN(Q)
 9:     S = S ∪ u
10:     for each vertex v ∈ G.Adj[u] do
11:         RELAX($u$, $v$, $u$)
12:         if Call of RELAX decreased $v.d$ then
13:             DECREASE-KEY($Q$, $v$, $v.d$)
14:         end if
15:     end for
16: end while
```

This is referenced from CLRS [3, p. 620]

Dijkstra's algorithm is particularly well suited for shortest path analysis on sparse graphs, like the Northeastern University Boston campus, and its ability to find the shortest path from any given starting vertex makes it particularly well-suited for campus navigation, where users need to determine the shortest path from start points such as their port of entry to campus (Ruggles, Columbus garage, etc.) or a particular class room to other locations on campus.

The reason Dijkstra's is particularly well suited for sparse graph analysis is run-time. The base time complexity for Dijkstra's is $O(V \log V)$, running for exactly one source vertex s we must visit every vertex (V) and also extract each vertex from the priority queue ($\log V$). While the true run-time is $O(V \log V + E \log V)$, the $E \log V$ represents the run time for the edge relaxation, where DECREASE-KEY runs in $O(\log V)$ time and E is the number of times DECREASE-KEY is run. In a sparsely connected graph, there simply are not many edges to run relative to the maximum number of potential edges were the graph fully connected (i.e. $E \cong V^2$). E is thus roughly linear with respect to V and we can think of the run-time as roughly 2 * $V \log V$ where, for $big - O$ purposes, the 2 is irrelevant. As discussed below, it would be prudent to consider a Dijkstra's time load of $O(V^2 \log V)$, as Johnson's Algorithm will be calling Dijkstra's for each of our V vertices (hence $V * V\ logV$) after Bellman-Ford re-weights the graph(s) for any negative edges.

Run specifically within the context of our use case and in isolation from the other algorithms discussed, a student entering the campus for the day at Ruggles could receive the shortest path available to each of the other locations on campus include Marino for a workout, Tatte for a bite to eat, or Dodge for class. They could then run it again after their chosen activity to figure out the shortest path from that activity to the rest of the graph, possibly to determine if it makes the most sense to go from the gym to Snell for some studying or the Curry Student Center for food. As discussed in the next section, applying Johnson's Algorithm to our use case will provide us with all pairs of shortest paths between any two locations on campus, thus removing the need for a student to repeatedly run

Dijkstra's every time they want to go somewhere.

8 Johnson's Algorithm

Overview of Johnson's Algorithm:
Johnson's algorithm finds all pairs of shortest paths between any two vertices. It operates by running Bellman-Ford outlined above to re-weight any negative path weights. The source node for Bellman-Ford is now deleted from the graph so the following can occur. Dijkstra's, the second part of Johnson's algorithm, can be employed now with re-weighted, and valid for Dijkstra's algorithm due to their positivity, path weights. After Dijkstra's all paths lengths must be calculated according to the original weights so they must go through the inverse operation. Below are the two equations one for re-weighting and one for finding original distances respectively.

$$w'(u,v) = w(u,v) + h(u) - h(v)$$

$$w(u,v) = w'(u,v) - h(u) + h(v)$$

Pseudocode for Johnson's Algorithm:

```
Johnson(G)
    1.
    create G‘ where G‘.V = G.V + {s},
        G‘.E = G.E + ((s, u) for u in G.V), and
        weight(s, u) = 0 for u in G.V
    2.
    if Bellman-Ford(s) == False
        return "The input graph has a negative weight cycle"
    else:
        for vertex v in G‘.V:
            h(v) = distance(s, v) computed by Bellman-Ford
        for edge (u, v) in G‘.E:
            weight‘(u, v) = weight(u, v) + h(u) - h(v)
    3.
        D = new matrix of distances initialized to infinity
        for vertex u in G.V:
            run Dijkstra(G, weight‘, u) to compute distance‘(u, v) for all v in G.V
            for each vertex v in G.V:
                D_(u, v) = distance‘(u, v) + h(v) - h(u)
        return D
```

This is referenced from [4].

Though both algorithms are outlined in previous sections it seems prudent to put a formulation of Johnson's algorithm as it utilizes both Bellman-Ford and Dijkstra's in a novel way to produce a more complex solution than either alone.

Runtime of Johnson's Algorithm: The runtime complexity of Johnson's is as follows:

$$O(V^2 \log V + VE)$$

This can be shown to be correct if given Bellman-Ford and Dijkstra's asymptotic behavior are given.

Bellman-Ford's:

$$O(VE)$$

Dijkstra's:

$$O(V \log V + E)$$

Bellman-Ford is run once to re-weight the negative edges so that Dijkstra's can be utilized. This provides the $+VE$ term in Johnson's runtime complexity.

Dijkstra's is run from a starting node to and calculates shortest path weights to them. The runtime complexity is $O(V(V \log V))$ since this algorithm is run for every vertex and the edges are proportional to V in a sparse graph and the $+E$ term does not impact asymptotic behavior.

For sparse graphs such as the one on Northeastern campus between the proposed nodes this runtime complexity is essentially

$$O(V^2 \log V)$$

as the number of edges is only proportional to V. The first term dominates the asymptotic behavior.

Application of Johnson's Algorithm:
One often finds themselves entering campus from the same port of entry every time they come to campus. Wouldn't it be useful to always know the quickest way to get to all other attractive destinations from this beginning position? This is where Johnson's algorithm is so useful as it finds shortest paths to every other vertex from a source node. On any given day one may need to reach a different destination and would like to do so in an optimal manner depending on the day's preference of starting location and time or cost constraint.

Let us assume we are an incoming student into Northeastern University. We have elected to use public transit and will enter the campus from Ruggles Station on the Orange line of the MBTA every day when we come to campus. On every day we enter campus we have different classes scheduled at different times. Sometimes we wish to go directly to the day's class and other day's we keep ourselves busy with other activities such as visiting the gym for a workout first. With Johnson's algorithm we can have a comprehensive list of shortest weight paths and the paths outlined themselves, given to us by the predecessor algorithms, to reach all other destinations on campus for the particular day's objective. This is quite a practical application of the above algorithm for the new student. And it can be applied differently in future semesters if maybe the incoming student obtains a parking pass and can now reference our result from the algorithm from a different port of entry. It is flexible and practical for a new student to master the campus early in their tenure here at Northeastern University. It can be used on either graph for time or cost depending on the short term need or long term need of the user.

References

[1] Google Maps, *Northeastern university*, Accessed: December 6, 2024, 2024. [Online]. Available: `https://www.google.com/maps/place/Northeastern+University/@42.339904,-71.0924695,16z/data=!3m1!4b1!4m6!3m5!1s0x89e37a1999cf5ce1:0xc97b00e66522b98c!8m2!3d42.339904!4d-71.0898892!16zL20vMDIyNXY5%5C?entry=ttu%5C&g%5C_ep=EgoyMDI0MTIwMy4wIKXMDSoASAFQAw%5C%3D%5C%3D`.

[2] J. Su, *Cs 161 lecture 14 – amortized analysis*, `https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture14.pdf`, Lecture notes, 9 pages, 2023–2024.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms, fourth edition*, `https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf`, Chapter 22: Single-Source Shortest Paths, p. 620, 2022.

[4] A. Chumbley, K. Moore, and J. Khim, *Johnson's algorithm*, Retrieved December 6, 2024, 2016. [Online]. Available: `https://brilliant.org/wiki/johnsons-algorithm/`.