Computer Science Extended Essay:

Optimizing Shortest Path Algorithms in Weighted Graphs using Priority Queue Implementation

---

Research Question:

How Does the Use of a Priority Queue and the Implementation of Bellman-Ford and Dijkstra's Algorithm Affect the Time Complexity of Solving the Shortest Path Problem in Weighted Graphs?

Candidate Code: jpk691

Essay Word Count: 3999

Table of Contents

1. Introduction

Finding the most optimal route, whether it be networks, transportation or information flow, lies at the heart of efficient resource utilization and problem-solving in various domains. As the scale and complexity of these systems continue to expand, algorithms that determine the shortest path between points become indispensable. The Shortest Path Problem (SPP) is a well-known optimization problem in graph theory that deals with determining the minimum weight path between two vertices in a weighted graph. The efficiency of solving the SPP varies greatly based on the algorithms and data structures employed. The problem is considered computationally challenging due to its high complexity, and different algorithms have been proposed to solve it efficiently.

This essay will focus on investigating the time complexity (refer to **Appendix C** for definition) of pathfinding algorithms at solving the SPP on weighted graphs. This essay will specifically explore *Dijkstra's Algorithm* and the *Bellman–Ford Algorithm*, which are two prominent shortest-path algorithms. Both algorithms will also be compared with the implementation of a priority queue, which is an abstract data structure which, in the context of this essay, is used to keep track of the vertices to be explored. Which gives rise to the research question: "*How does the use of a priority queue and the implementation of Bellman-Ford and Dijkstra's algorithm affect the time complexity of solving the shortest path problem in weighted graphs?*".

2. Background Information

2.1 Weighted Graphs and The Shortest Path Problem

A graph consists of points (vertices) connected by lines (edges), in weighted graphs, each edge has an associated weight. This weight can represent various things depending on the context, commonly distance or cost. There are two types of weighted graphs: directed (digraphs) and undirected graphs. Digraphs have edges that point from one vertex to another in a specific direction. Undirected weighted graphs have bidirectional edges that can be traversed in either direction between two vertices. Weighted digraphs are useful for modeling one-way relationships, such as transportation networks. Undirected weighted graphs are suited for modeling mutual relationships like social networks (example shown in **Figure 1**).

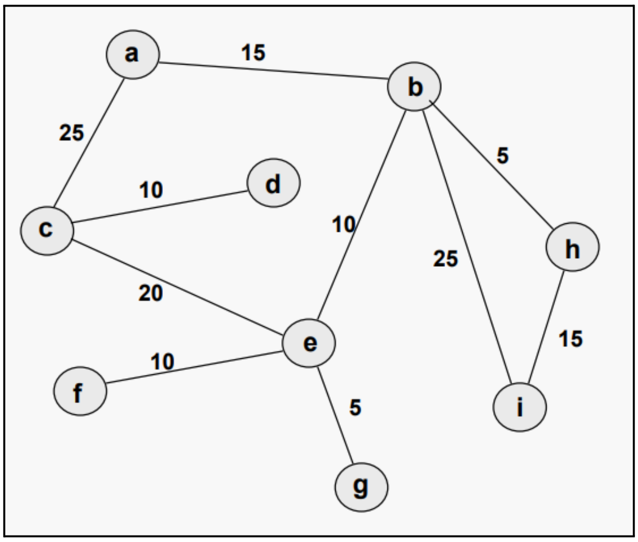


Figure 1: *Undirected Weighted Graph Consisting of 9 Vertices* (Virginia Tech: Department of Computer Science)

When the sum of the edge weights between two vertices is minimum (relative to all other possible routes), it is considered the shortest path. Given a weighted digraph $\mathrm{G} = (\mathrm{V}, \mathrm{E}, w)$, to find the shortest path from a source vertex s to and an end vertex e, a path P from the start to end vertex needs to be found that minimizes the function:

$$w(P) := \sum_{u \to v \in P} w(u \to v)$$

(Erickson 273)

There are two main variations of the SPP: single-source (SSSP) and all-pairs (APSP), see **Appendix C** for definitions. This essay will be specifically focusing on the $SSSP^+$ problem, where all weights are positive (i.e. $w : \mathbf{E} \to \mathbb{R}^+$), as the APSP is comparatively more computationally taxing and time-consuming.

2.2 Shortest-Path Algorithms

2.2.1 Dijkstra's Algorithm

The most popular shortest-path algorithm widely used in applications such as Google Maps. It works by iteratively selecting the vertex with minimum weight, updating distances to its neighbors by considering the weights of the connecting edges, and repeating this process until all vertices have been visited.

Dijkstra's Algorithm is an example of a greedy algorithm (refer to **Appendix C** for definition). One stipulation to using the algorithm is that the graph needs to have a nonnegative weight on every edge (Abiy et al.).

In the context of this essay, a "naive" algorithm (refer to **Appendix C** for definition) is one with no optimizations (i.e. no priority queue).

```
function NaiveDijkstra(graph, source):
    dist = {}      // dictionary to store the shortest distances
    visited = {}   // dictionary to keep track of visited nodes

    for each vertex in graph:
        dist[vertex] = infinity   // initialize all distances to infinity
        visited[vertex] = false   // mark all nodes as not visited

    dist[source] = 0    // distance from source to itself is 0

    while there are unvisited nodes:
        current = node with the minimum distance from dist dictionary

        visited[current] = true   // mark the current node as visited

        for each neighbor of current:
            if visited[neighbor] is false:
                // calculate the new distance from source to neighbor
                newDistance = dist[current] + edge_weight(current, neighbor)

                if newDistance < dist[neighbor]:
                    dist[neighbor] = newDistance   // update the distance

    return dist
```

Figure 2: *Pseudocode for Naive Dijkstra's Algorithm* (ChatGPT, 2023)

The following is a breakdown of the pseudocode (**Figure 2** above):

1. Declaration: The 'dist' dictionary stores the shortest distances between source and every other vertex in the graph, while 'visited' keeps track of visited vertices.
2. Initialization: The distance from the source vertex to every other vertex is initially set as ∞ (unknown), except the source vertex, which is set to 0.
3. The algorithm will run while there are unvisited vertices.
    a. Selecting Vertex With Minimum Distance: The algorithm selects the vertex 'current' with the minimum distance from the 'dist' dictionary among the unvisited vertices in each iteration, or $\text{current} = \arg\min(\text{dist}[\text{vertex}])$. The current vertex is marked as visited or *true*.
    b. For each neighbor vertex of the current vertex:
        i. Check if the neighbor vertex is unvisited: If visited[neighbor] is *false*, it means it has not been visited. 'newDistance' is a variable that will calculate and store the new distance from the source to the neighbor vertex.

ii. The calculated distance is compared with the current distance in the ‘dist[neighbor]’ dictionary. If smaller, a shorter path has been found from the source vertex to the neighbor. ‘dist[neighbor]’ is updated with the new, smaller distance. Denoted mathematically as $\mathrm{dist}[\mathrm{neighbor}] = \min(\mathrm{dist}[\mathrm{neighbor}], \mathrm{newDistance})$. ‘min’ is used here to choose the smaller of the two distances. By updating ‘dist[neighbor]’ with the new smaller distance, the edge between the current vertex and its neighbor vertex is “relaxed” (refer to **Appendix C** for definition). ‘dist’ is returned, which now contains all the shortest paths from the source vertex to every other vertex.

Consider the following graph:

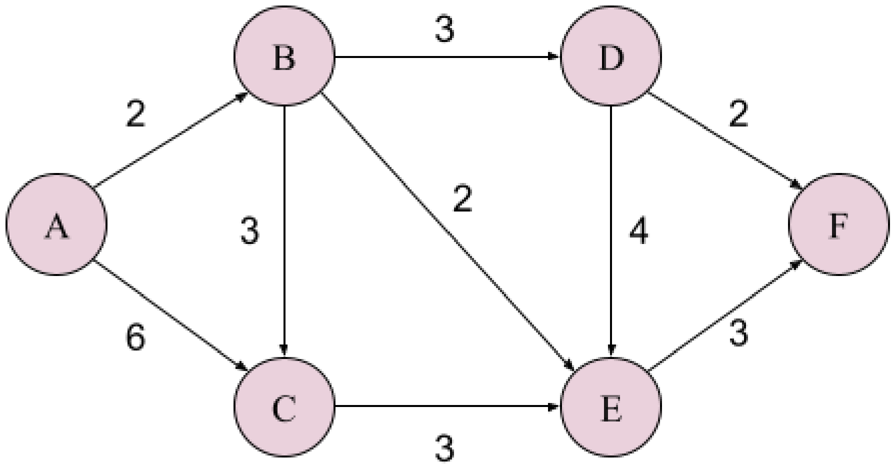


Figure 3: *Example Weighted Digraph with 6 Vertices and 9 Edges*

Below is an example use of the algorithm against **Figure 3**.

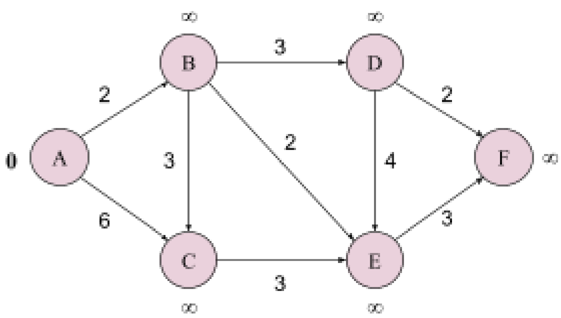
<table>
<tr><th>Graph</th><th>Explanation</th></tr>
<tr><td>


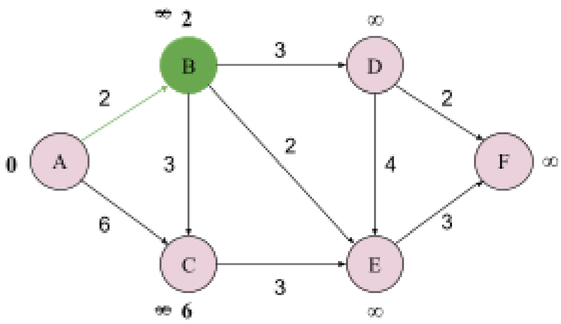
Figure 4: Distances Initially Unknown
</td><td>
The distance from every vertex to the start vertex is initialized to be ∞ except the starting vertex itself, which would be 0.
The only vertex visited so far is the starting vertex itself, A.
$\text{dist} = \{\text{A: } 0, \text{ B: } \infty, \text{ C: } \infty, \text{ D: } \infty, \text{ E: } \infty, \text{ F: } \infty\}$
$\text{visited} = \{\text{A: True, B: False, C: False, D: False, E: False, F: False}\}$
</td></tr>
<tr><td>


Figure 5: Exploring Neighbors of Vertex A
</td><td>
The neighbors of the current vertex (A) are explored, which are B and C. Now, the edge from A to B is relaxed. Currently, the distance known from A to B is ∞, and the new potential distance is 0 + 2 = 2. Since 2 < ∞, the shortest distance from A to B is updated as 2. This can also be represented mathematically as:
$\text{dist}[\text{B}] = \min(\text{dist}[\text{B}], \text{dist}[\text{A}] + \text{edge\_weight}(\text{A}, \text{B}))$
$= \min(\infty, 0 + 2)$
$= 2$
The edge from A to C is to be relaxed, the known distance from A to C is ∞, the new potential distance is 0 + 6 = 6. Since 6 < ∞, the shortest distance from A to C is updated as 6. This can be represented mathematically in a similar way as the above.
Now, the current vertex is set to the neighbor vertex with the smallest distance, which is B (since 2 < 6).
$\text{current} = \text{B}$
$\text{visited} = \{\text{A: True, B: True, C: False, D: False, E: False, F: False}\}$
$\text{dist} = \{\text{A: } 0, \text{ B: } 2, \text{ C: } 6, \text{ D: } \infty, \text{ E: } \infty, \text{ F: } \infty\}$
</td></tr>
</table>

<table>
<tr><th>Graph</th><th>Explanation</th></tr>
<tr><td>

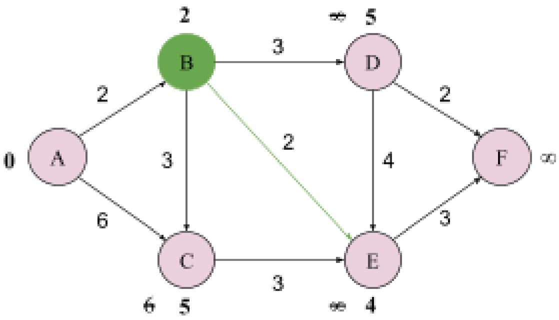


Figure 6: *Exploring Neighbors of Vertex B*

</td><td>

B is marked as visited in the ‘visited’ dictionary by setting it to *true*. The neighbors of B are to be explored, which are C, D and E. The process of edge relaxation is repeated. Now, the edge from B to C is relaxed. The current known shortest distance from A to C is 6, but the new potential distance through B is 2 + 3 = 5. Since 5 < 6, the shortest distance to C can be updated to be 5. Denoted mathematically as:

$$\mathrm{dist}[C] = \min(\mathrm{dist}[C], \mathrm{dist}[B] + \mathrm{edge\_weight}(B, C))$$
$$= \min(6, 2 + 3)$$
$$= 5$$

The edge from B to E is to be relaxed, the current distance is ∞ from A to E, the new potential distance through B is 2 + 2 = 4. Since 4 < ∞, the shortest distance from A to E is 4.

Finally, the edge from B to D is relaxed; the current distance is ∞, and the new potential distance through B is 2 + 3 = 5. Since 5 < ∞, the shortest distance from A to D is 5. Now, the current vertex is set to the neighbor vertex with the smallest distance, which is E (since 2 < 3).

$\mathrm{dist} = \{A: 0,\ B: 2,\ C: 5,\ D: 5,\ E: 4,\ F: \infty\}$
$\mathrm{visited} = \{A: \mathrm{True},\ B: \mathrm{True},\ C: \mathrm{True},\ D: \mathrm{True},\ E: \mathrm{True},\ F: \mathrm{False}\}$

</td></tr>
<tr><td>

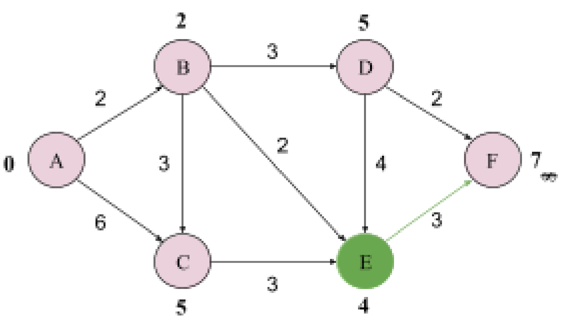


Figure 7: *Exploring Neighbors of Vertex E*

</td><td>

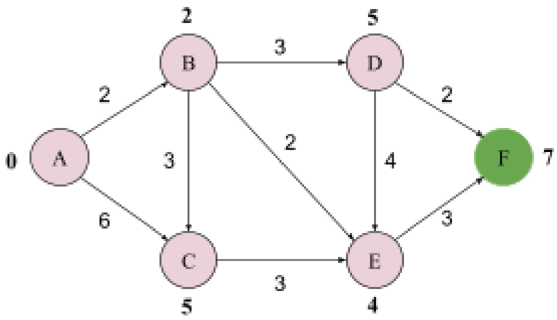
$\mathrm{current} = E$

Now explore the neighbors of E, which would be only F. The edge from E to F is to be relaxed. The current known distance from A to F is ∞ and the potential distance will be 2 + 2 + 3 = 7. Since 7 < ∞, the shortest distance from A to F is updated to be 7.

The current vertex is set to the neighbor vertex with the smallest distance, which is simply F.

$\mathrm{dist} = \{A: 0,\ B: 2,\ C: 5,\ D: 5,\ E: 4,\ F: \infty\}$

</td></tr>
</table>

Graph	Explanation
 **Figure 8:** *End of Loop, Shortest Paths Found*	current = F visited = {A: True, B: True, C: True, D: True, E: True, F: True} Now that all the vertices have been visited, the loop terminates and the 'dist' dictionary containing the shortest paths from A to every other vertex is returned. As shown below: dist = {A: 0, B: 2, C: 5, D: 5, E: 4, F: 7}

This essay will use Big-O notation (refer to **Appendix C** for definition) for measuring time complexity (as opposed to Big-Ω or Big-Θ), as it aims to evaluate the performances of both algorithms under maximum stress (worst-case).

The worst-case time complexity of Dijkstra's Algorithm without the use of a priority queue is $O(V^2)$, where V represents the number of vertices in the graph. In each iteration, the algorithm needs to find the vertex with minimum distance among the unvisited vertices. In a naive implementation, this requires iterating over all vertices, resulting in a time complexity of $O(V)$ for this operation. As this process is repeated for each vertex ($O(V)$), the overall time complexity becomes $O(V^2)$ (since $O(V) \times O(V)$).

2.2.2 Bellman-Ford Algorithm

Bellman-Ford's algorithm finds applications in networking, particularly in a distance-vector routing protocol. This protocol decides how to route packets of data on a network (Chumbley et al.).

Similar to Dijkstra's Algorithm, it uses the idea of relaxation but doesn't use [it] with [a] greedy technique (Morampudi). It works by keeping track of weight distance from the origin and previous node in the shortest path, looping over the edges/connections for n times (n being the number of nodes/vertices), and updating the fastest route to the destination (stevenard). In comparison to Dijkstra's algorithm, the Bellman-Ford algorithm admits or acknowledges the edges with negative weights (Magzhan and Mat).

```
function bellmanFord(graph, source):
    distance = {}  // dictionary to store the shortest distance from the source

    // Step 1: Initialization
    for each vertex v in graph:
        distance[v] = infinity
    distance[source] = 0

    // Step 2: Relax edges repeatedly
    for i from 1 to |V|-1:  // |V| is the number of vertices in the graph
        for each edge (u, v, w) in graph:
            if distance[u] + w < distance[v]:
                distance[v] = distance[u] + w

    // Step 3: Check for negative weight cycles
    for each edge (u, v, w) in graph:
        if distance[u] + w < distance[v]:
            return "Graph contains a negative weight cycle"

    return distance
```

Figure 9: *Pseudocode for Naive Bellman-Ford Algorithm* (ChatGPT, 2023)

The following is a breakdown of the pseudocode (**Figure 9** above):

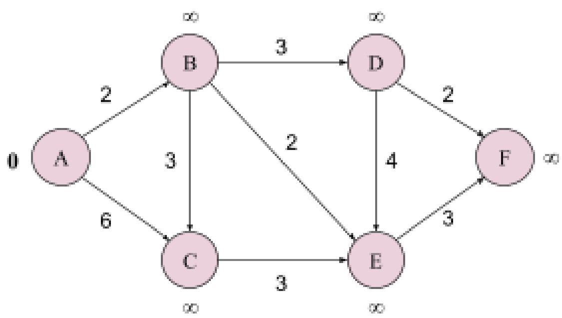
1. Declaration: A dictionary called 'distance' is created to store the shortest distance from the source vertex to every other vertex.
2. Initialization: Set the distance from the start vertex to every other vertex to be ∞, except the start vertex, which is set to 0.
3. Continuously Relax Edges: Iteratively update the distances until the optimal solution is found. Let $|V|$ be the total number of vertices in the graph. This process is repeated $|V| - 1$ times.
    a. For every edge (u, v, w) in the graph, where u and v are vertices and w is the weight of the edge, the distance to v is checked to see if it can be minimized by going through u.

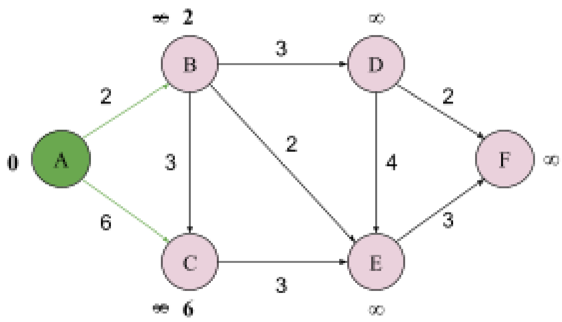
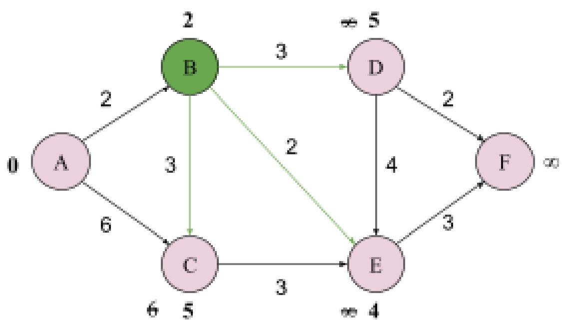
i. If this condition is satisfied, the distance of vertex v is updated to be the sum of the distance to vertex u and the weight of the edge (u, v). The relaxation can be denoted mathematically as:

$$\text{distance}[\mathrm{v}] = \min(\text{distance}[\mathrm{v}], \text{distance}[\mathrm{u}] + w)$$

4. Check for negative weight cycles: If the sum of the weights of the edges along the cycle is negative, it is defined as a negative weight cycle. For every edge $(\mathrm{u}, \mathrm{v}, w)$, the distance from the start vertex to vertex v $(\text{distance}[\mathrm{u}] + w)$ is checked to see if it is smaller than the current shortest distance to v $(\text{distance}[\mathrm{v}])$. If a negative weight cycle is present, the algorithm will fail to provide a solution.
5. If the condition above is not satisfied, the 'distance' dictionary is returned which now contains all the shortest paths.

The algorithm will not be demonstrated using a graph with negative weights as it is not relevant to this experiment. The following example demonstrates the algorithm's use on the same graph (**Figure 3**) as discussed in the previous section.

Graph	**Explanation**
 **Figure 10:** *Distances Initially Unknown*	The distance from every vertex to the start vertex is initialized to be ∞ except the starting vertex itself, which would be 0. The only vertex visited so far is the starting vertex itself, A. distance $= \{$A: 0, B: ∞, C: ∞, D: ∞, E: ∞, F: $\infty\}$

Graph	Explanation
 **Figure 11:** *First Iteration*	The main loop is entered, where edges in the graph are continuously relaxed for a total of 5 iterations in this case. For every edge (u, v, w) in the graph: $(A, B, 2)$: $\text{distance}[A] + 2$ $= 0 + 2$ $= 2$ Now if this distance is smaller than the current distance[B], which is infinity, the distance from A to B is updated as 2 $(\text{distance}[B] = 2)$. $(A, C, 6)$: $\text{distance}[A] + 6$ $= 0 + 6$ $= 6$ Similar to the above, $\text{distance}[C]$ is currently infinity, which is greater than 6. Thus $\text{distance}[C] = 6$. This is the first iteration of Step 3 (refer to pseudocode). The known distances are as shown below. distance = {A: 0, B: 2, C: 6, D: ∞, E: ∞, F: ∞}
 **Figure 12:** *Second Iteration*	$(B, C, 3)$: $\text{distance}[B] + 3$ $= 2 + 3$ $= 5$ Now, compare this with the current $\text{distance}[C]$, which is 6. Since $5 < 6$, $\text{distance}[C]$ is updated to be 5, which is the shorter path. $(B, E, 2)$: $\text{distance}[B] + 2$ $= 2 + 2$ $= 4$ Since $4 < \infty$, $\text{distance}[E]$ is updated to be 4. $(B, D, 3)$: $\text{distance}[B] + 3$ $= 2 + 3$ $= 5$

<table>
<tr><th>Graph</th><th>Explanation</th></tr>
<tr><td></td><td>Since $5 < \infty$, $\text{distance}[D]$ is updated to be 5.

This is the second iteration of Step 3. The new distances are shown as below:

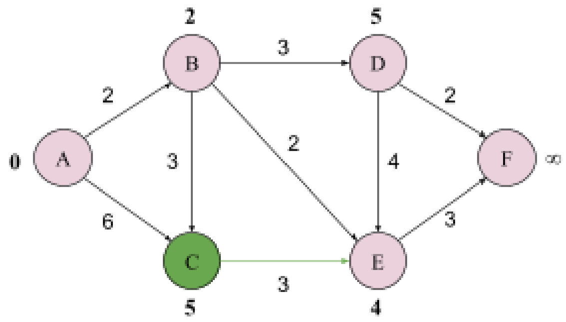
$\text{distance} = \{A: 0, B: 2, C: 5, D: 5, E: 4, F: \infty\}$</td></tr>
<tr><td>

Figure 13: Third Iteration</td><td>(C, E, 3): $\text{distance}[C] + 3$
$= 5 + 3$
$= 8$
Since 8 is greater than the current $\text{distance}[E]$, which is 4. The distance dictionary is not updated.

This is the third iteration of Step 3. The new distances are shown as below:

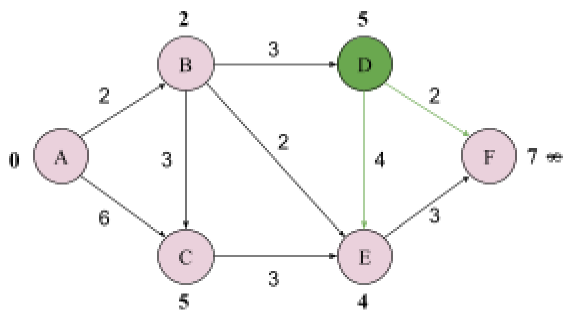
$\text{distance} = \{A: 0, B: 2, C: 5, D: 5, E: 4, F: \infty\}$</td></tr>
<tr><td>

Figure 14: Fourth Iteration</td><td>(D, E, 4): $\text{distance}[D] + 4$
$= 5 + 4$
$= 9$
Since 9 is greater than the current distance $\text{distance}[E]$, which is 4. The distance dictionary is not updated.

(D, F, 2): $\text{distance}[D] + 2$
$= 5 + 2$
$= 7$
Since 7 is less than the current value of $\text{distance}[F]$, which is ∞. $\text{distance}[F]$ is updated to be 7.

This is the fourth iteration of Step 3. The new distances are shown as below:

$\text{distance} = \{A: 0, B: 2, C: 5, D: 5, E: 4, F: 7\}$</td></tr>
</table>

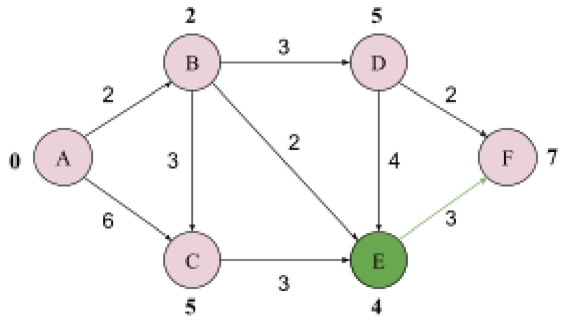
<table>
<tr><th>Graph</th><th>Explanation</th></tr>
<tr><td>

Figure 15: Fifth Iteration</td><td>$(\text{E, F, } 3)\text{: distance}[\text{E}] + 3$
$= 4 + 3$
$= 7$

No updates occur in the fifth and final iteration, all edges have been successfully relaxed. This is the fifth iteration of Step 3. The final distances are shown as below:

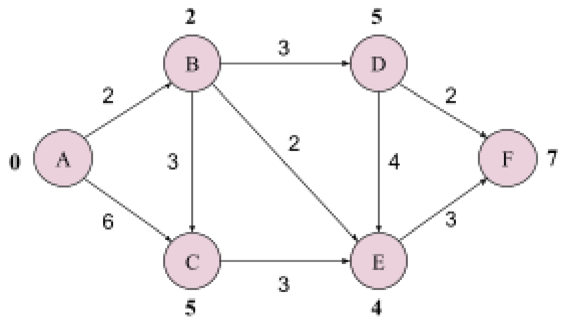
$\text{distance} = \{\text{A: } 0, \text{ B: } 2, \text{ C: } 5, \text{ D: } 5, \text{ E: } 4, \text{ F: } 7\}$</td></tr>
<tr><td>

Figure 16: End of Loop, Shortest Paths Found</td><td>Now, check for any negative weight cycles, for which there are none

Finally, the ‘distance’ dictionary is returned containing all the shortest paths:

$\text{distance} = \{\text{A: } 0, \text{ B: } 2, \text{ C: } 5, \text{ D: } 5, \text{ E: } 4, \text{ F: } 7\}$</td></tr>
</table>

The time complexity of the naive Bellman-Ford Algorithm is $\text{O}(\text{VE})$. Where V is the number of vertices and E is the number of edges. The algorithm repeatedly relaxes each edge for $\text{V} - 1$ iterations, in each iteration, it checks all edges E within the graph and updates the distance to each vertex in case a shorter path is found. Resulting in $(\text{V} - 1) \times \text{E}$ relaxation operations, the constant -1 can be ignored. Resulting in a total time complexity of $\text{O}(\text{VE})$.

2.3 Priority Queues

A priority queue is a variation of the traditional queue data structure where each element in the queue is associated with a priority value, elements are attended to in the queue based on this priority value. Conventionally, the element at the front of the queue has the highest priority value.

Priority queues are usually implemented using heaps (refer to **Appendix C** for definition). There are two different types of heaps, a max heap and min heap. In a min heap, the value of the parent node is less than or equal to the value of the child node, for all nodes, this property is known as the heap invariant. The max heap is simply the opposite. The value of the nodes in the context of this experiment being distances. **Figure 16** shows a min heap of height 3 satisfying the heap invariant, while **Figure 17** shows a violated heap invariant where the child of parent node 3 is less than its parent.

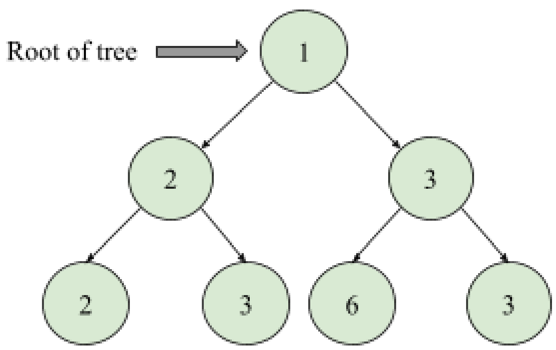


Figure 17: *Valid Min Heap*

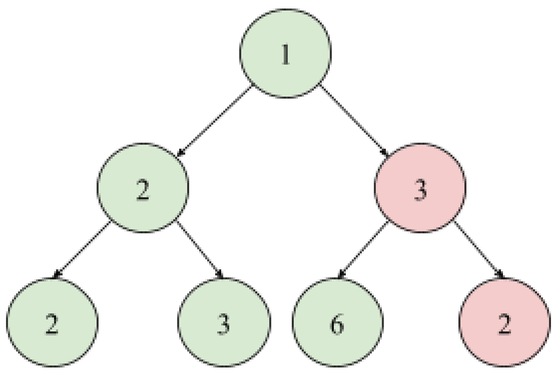


Figure 18: *Invalid Min Heap*

The two main operations of the priority queue that concern this experiment are insertion and extraction (removal). The worst-case time complexity of both operations is $O(\log_2 N)$ (Garg), where N is the number of elements in the heap. Since the min heap can be visualized as a complete binary tree, the tree's height can be expressed as a logarithm of N (e.g. **Figure 18** above has 7 nodes, therefore $\lceil \log_2 7 \rceil = 3$). Leading to a logarithmic time complexity for both insertion and extraction.

In this experiment, the priority queue being implemented into the algorithms will be using a min heap, as Python has a built-in data structure which is essential to prioritize the smallest distance at each localized step of the algorithm, in order to find an optimal path.

3. Hypothesis and Applied Theory

The experiment will find the relationship between execution time in nanoseconds (y) and the number of vertices in the digraph (x). By increasing the number of vertices, a clear correlation can be drawn between both variables and how the relationship differs when a priority queue is implemented within both algorithms.

The graph used in this experiment is such that $|\mathrm{E}| > |\mathrm{V}|$. Thus, recalling the time complexities for both algorithms, it is hypothesized that for naive implementations, Dijkstra's Algorithm will run with a lower execution time. The number of edges in the graph are such that $|\mathrm{V}| \times 1.5 \approx |\mathrm{E}|$, so the time complexity of the Bellman-Ford can be approximated to be a quadratic as $\mathrm{O}(\mathrm{V} \times \mathrm{V} \times 1.5) \approx \mathrm{O}(\mathrm{V}^2)$. Therefore, for both naive algorithms, there is predicted to be a quadratic relationship between x and y.

For Dijkstra's Algorithm, the priority queue is queried to extract the vertex with the smallest current distance, taking $\mathrm{O}(\log_2 \mathrm{V})$ time. Then, the algorithm performs edge relaxation on the neighboring vertices of the extracted vertex, potentially updating their current distances. This is repeated until all vertices have been processed or the priority queue is empty, improving the time complexity to $\mathrm{O}((\mathrm{V}+\mathrm{E}) \log_2 \mathrm{V})$, as the extraction of the vertex with the smallest distance becomes optimized through the logarithmic time complexity of the min heap extraction.

As mentioned in **Section 2.2.2**, the number of relaxation operations in the Bellman-Ford Algorithm is $\mathrm{O}(\mathrm{VE})$. With a priority queue, each relaxation operation involves inserting or extracting an element in the priority queue, which is time complexity $\mathrm{O}(\log_2 \mathrm{V})$, leading to the relaxation operations having a time complexity of $\mathrm{O}((\mathrm{VE}) \log_2 \mathrm{V})$. The priority queue benefits the selection of the vertex with the smallest distance in each iteration which can be done in $O(\log_2 V)$ time. Combining this time complexity with the relaxation operations results in $O((VE)\log_2 V \times \log_2 V)) = O((V+E)\log_2 V)$.

It appears that both min heap implemented algorithms have the same time complexity, however, the graph that will be used in the experiment is such that there are no negative weights, and Dijkstra's Algorithm tends to perform better in this environment. Furthermore, Dijkstra's Algorithm does not iterate through all the edges, while the Bellman-Ford Algorithm does multiple times.

Thus, it is hypothesized that with the priority queue implementations, both algorithms will reduce in execution time, specifically, Dijkstra's Algorithm will run with a lower execution time compared to the Bellman-Ford Algorithm. For both priority queue implemented algorithms, there is predicted to be a logarithmic relationship.

4. Experimental Methodology

4.1 Weighted Graph Used

As a resident of Singapore, the author of this paper frequently relies on the public bus network for their transportation needs. Therefore, for this experiment, they decided to represent the Singapore Bus Network as a weighted digraph, which can be used to evaluate the execution times of the algorithms. The vertices being bus stops and the weights being the travel distances between the stops in kilometers.

The dataset used in the experiment's code originates from the Land Transport Authority; this data was mostly scraped from the website and compiled into a data repository on Github (Aun). Two files were saved locally on the author's system, stops.json and services.json.

```python
# Constants for Haversine formula
AVG_EARTH_RADIUS = 6371  # in kilometers

# Step 1: Parse services.json
with open('services.json') as services_file:
    services_data = json.load(services_file)

# Step 2: Parse stops.json
with open('stops.json') as stops_file:
    stops_data = json.load(stops_file)

# Step 3: Create a dictionary of stop IDs to coordinates
stop_coordinates = {}
for stop_id, stop_info in stops_data.items():
    longitude, latitude, _, _ = stop_info
    stop_coordinates[stop_id] = (float(latitude), float(longitude))
```

Figure 19: *Code Snippet Showing The Parsing of JSON Data and Dictionary Population*

Both files services.json and stops.json are read and deserialized into Python objects.

The services.json file contains data about bus services in Singapore, each identified by a distinct service ID, such as 3 or 4, and including information about its name and routes. The stops.json file contains information regarding the bus stops themselves, such as bus stop ID, latitude and longitude. A dictionary called stop_coordinates is created and is populated with bus stop IDs as the keys and their corresponding latitude and longitude coordinates as values.

```python
# Function to calculate Haversine distance
def haversine_distance(coord1, coord2):
    lat1, lon1 = coord1
    lat2, lon2 = coord2

    dlat = math.radians(lat2 - lat1)
    dlon = math.radians(lon2 - lon1)

    a = math.sin(dlat / 2) * math.sin(dlat / 2) + math.cos(
        math.radians(lat1)) * math.cos(math.radians(lat2)) * math.sin(
        dlon / 2) * math.sin(dlon / 2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    distance = AVG_EARTH_RADIUS * c
    return distance
```

Figure 20: *Code Snippet Showing The Haversine Distance Formula*

The weights are calculated using the Haversine distance formula (shown below), which accurately calculates the distances between two points on the surface of a sphere (approximating the shape of Earth as a sphere) given the latitude and longitude of the two points.

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\Phi_2 - \Phi_1}{2}\right) + \cos\left(\Phi_1\right)\cos\left(\Phi_2\right)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

where d = distance between bus stops, r = radius of earth, Φ_1 = latitude of first point, Φ_2 = latitude of second point, λ_1 = longitude of first point, λ_2 = longitude of second point

(Sydorenko)

The weights do not reflect the real-life distances between the bus stops but rather provide an approximation.

```python
# Step 4: Create an empty directed graph
graph = nx.DiGraph()

# Step 5: Add edges to the graph with weights
stop_ids = list(stop_coordinates.keys())[:len(stop_coordinates)//1]

for service_id, service_info in services_data.items():
    routes = service_info['routes']
    for route in routes:
        for i in range(len(route) - 1):
            start_stop_id = route[i]
            end_stop_id = route[i + 1]

            if start_stop_id in stop_ids and end_stop_id in stop_ids:
                start_coordinates = stop_coordinates[start_stop_id]
                end_coordinates = stop_coordinates[end_stop_id]
                distance = haversine_distance(start_coordinates, end_coordinates)
                graph.add_edge(start_stop_id, end_stop_id, weight=distance)
```

Figure 21: *Code Snippet Showing The Creation of The Digraph*

A digraph is created using the NetworkX library. The services_data dictionary is iterated over, and for each service, the routes are examined. Within each route, consecutive pairs of stops are considered. If both the start and end stops are present in the stop_ids list, their corresponding coordinates are retrieved from the stop_coordinates dictionary. The distance between the start and end coordinates are calculated with the aforementioned haversine distance function. Finally, an edge is added to the graph, connecting the start and end stop IDs, with the calculated distance as the weight. Resulting in **Figure 22** below:

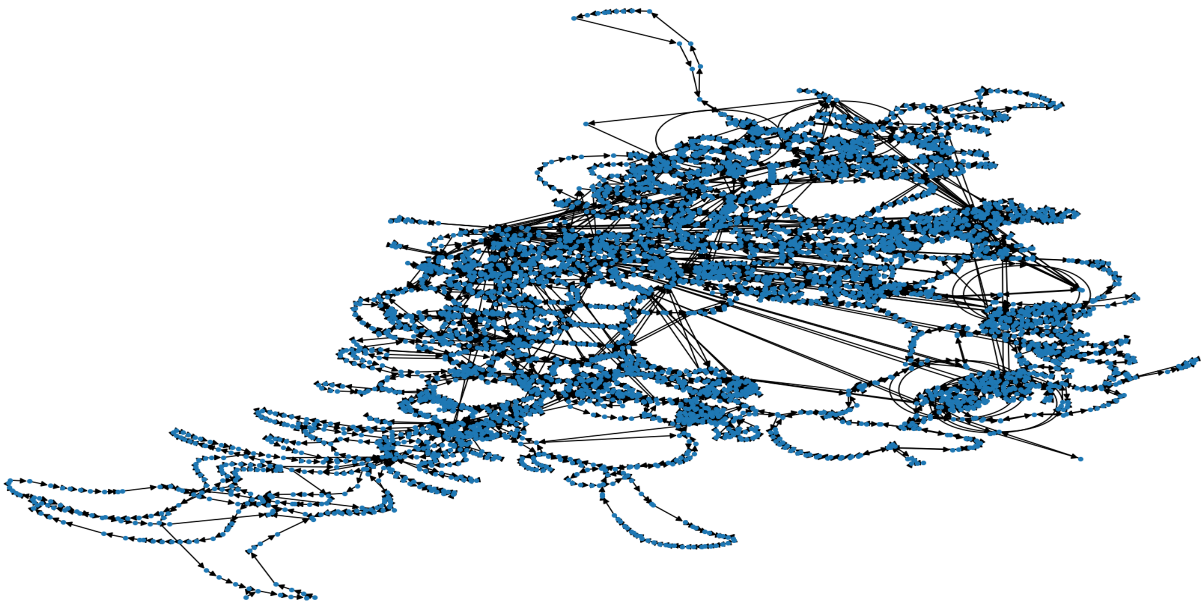


Figure 22: *Visualization of All The Bus Stops in Singapore Using the Kamada-Kawai Layout*

4.2 Independent Variables

The size of the weighted digraph will be changed, specifically, the number of vertices and edges in the graph. The idea is to iterate through increasing fractions of the stops.json file so that the number of vertices and edges in the graph can be increased, which is a convenient and logical way of changing the independent variables in this experiment. This will be done by changing this variable:

```
stop_ids = list(stop_coordinates.keys())[:len(stop_coordinates)//x]
```

By changing the x after the floor division operator, the fraction of the stops included in the graph can be controlled. For example, when $x = 10$, only one-tenth of the stops will be considered, resulting in a smaller graph size with a reduced number of vertices and edges. The value of x in this experiment will be varied from 10 to 1, with 1 being inclusive of all 5083 bus stops. The purpose is to showcase algorithm performance under escalating computational stress, revealing the correlation between execution time and input size. **Figure 23** shows the various number of vertices and edges that will be used to test the algorithms:

Vertices	Edges
508	734
564	816
633	911
726	1012
847	1186
1010	1363
1269	1681
1694	2241
2540	3454
5083	7420

Figure 23: *Range of Values for Independent Variable*

4.3 Dependent Variables

The only dependent variable is the execution time for the given algorithm to find the shortest path from the source vertex to all the other vertices (SSSP) in nanoseconds. This will be measured using the time.perf_counter_ns() function from the time module in Python, which provides a high-resolution timer for accurate timing measurements in nanoseconds.

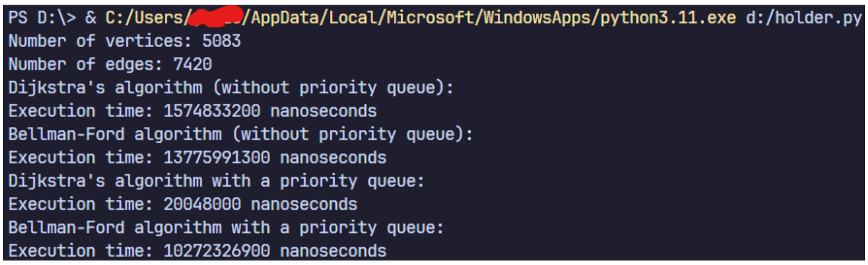
4.4 Controlled Variables

Variable	Description	Specification
IDE Used	The program will be run on the same IDE. To minimize variations in code execution and optimization, which can prevent systematic errors in execution times between the algorithms.	IDE: Visual Studio Code 1.79.0 (user setup) Python 3.11.4 - NetworkX 3.1 - json 2.0.9
Computer and OS	The program will be run on a Razer Book 13. This will minimize systematic and random error as there are no variations in hardware and OS.	OS: Microsoft Windows 11 Home Version 10.0.22621 Build 22621 Processor: Processor 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 Mhz, 4 Core(s), 8 Logical Processor(s) Memory: 16GB DDR4 SDRAM 4267MHz
Start Vertex	The first bus stop id found in the stops.json file will be used, so that the algorithms are subjected to the same initial conditions, reducing random error in results due to different starting points.	id: 10009
Input Dataset	The same data will be used to construct the weighted graph, reducing systematic error and ensuring a consistent basis for algorithmic comparison.	stops.json and services.json

Variable	Description	Specification
Graph Representation	The graph will be represented using the NetworkX library, the same graph structure and connectivity are used consistently throughout this experiment.	nx.DiGraph()
Graph	The same graph will be used to benchmark the algorithms. The distances between each vertex will be kept constant, reducing systematic error.	Weighted Digraph consisting of 5083 vertices and 7420 edges in total

4.5 Procedure

1. Install the following library: NetworkX
2. Open the file ee.py (see **Appendix A**) in Visual Studio Code version 1.79.0. Place stops.json and services.json in the same folder as the program.
3. Run the program by clicking the arrow button or pressing F5.
4. Perform 10 trials for each value of the independent variable (re-run the program). For each trial, record the execution times of the Dijkstra's algorithm (naive), Bellman-Ford algorithm (naive), Dijkstra's algorithm with a priority queue, and Bellman-Ford algorithm with a priority queue. Note down the results in a spreadsheet software.



```
PS D:\> & C:/Users/[redacted]/AppData/Local/Microsoft/WindowsApps/python3.11.exe d:/holder.py
Number of vertices: 5083
Number of edges: 7420
Dijkstra's algorithm (without priority queue):
Execution time: 1574833200 nanoseconds
Bellman-Ford algorithm (without priority queue):
Execution time: 13775991300 nanoseconds
Dijkstra's algorithm with a priority queue:
Execution time: 20048000 nanoseconds
Bellman-Ford algorithm with a priority queue:
Execution time: 10272326900 nanoseconds
```

Figure 24: *Results Outputted in the Terminal*

5. Calculate the average execution times for each algorithm (naive and priority queue) based on the recorded data.

5. The Experimental Results

5.1 Tabular Data Presentation

Figure 25 below shows the average execution time in nanoseconds for each algorithm to find the SSSP in the weighted digraph with increasing vertices and edges. See **Appendix B** for the raw data.

Vertices	Edges	Average Execution Time (nanoseconds)			
		Dijkstra's Algorithm (Naive)	Bellman-Ford Algorithm (Naive)	Dijkstra's Algorithm (Priority Queue)	Bellman-Ford Algorithm (Priority Queue)
508	734	18277760	145268750	1823730	92678050
564	816	20924270	167572360	1902800	102746450
633	911	26766110	226592440	2109500	131542560
726	1012	33335310	270494600	2194940	159646660
847	1186	48589910	365931930	2879720	247789270
1010	1363	66669930	507861230	3299370	344395450
1269	1681	105192020	734278890	3651060	466298250
1694	2241	174840830	1373807150	6027200	1015853050
2540	3454	388870850	3093997800	9051150	2287816100
5083	7420	1598144270	13528484770	17772930	9412515240

Figure 25: *Average Execution Time for Each Algorithm*

5.2 Graphical Representation of Data

Execution time was chosen to be graphed against the number of vertices instead of edges as it better reflects the complexity of the graph and enables analysis of algorithm efficiency and scalability with increasing graph size.

A log scale for the x axis was due to the wide and unevenly distributed range of values. The reason for this uneven distribution was justified in **Section 4.2**. Using a standard scale would result in a compressed representation, restricting effective observation of trends and patterns.

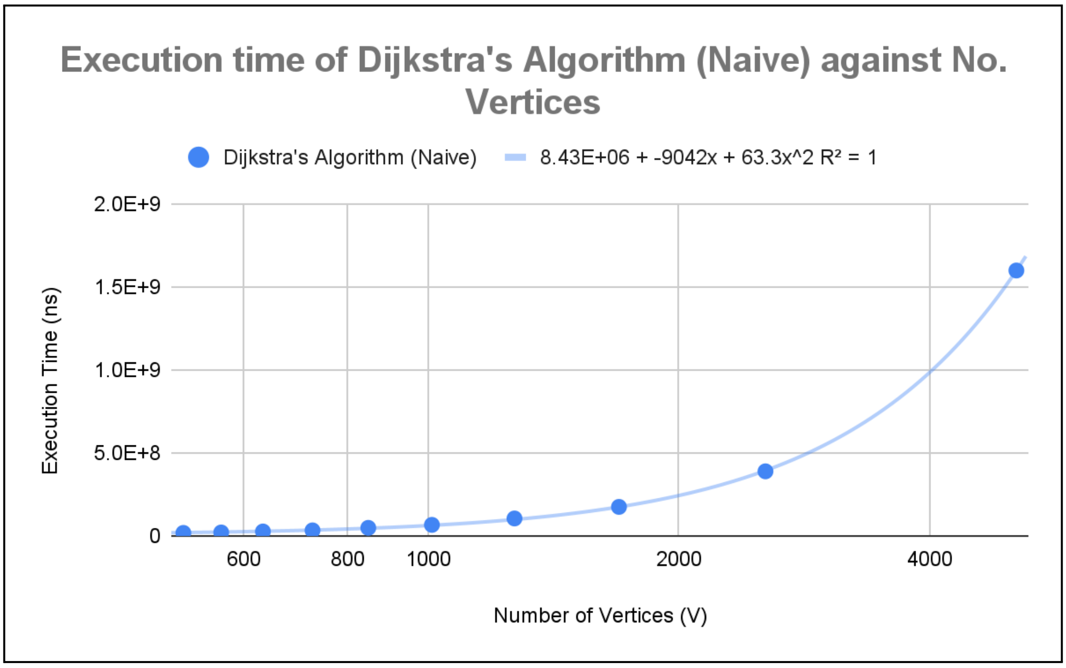


Figure 26: *Execution Time of Dijkstra's Algorithm (Naive) Against the Number of Vertices*

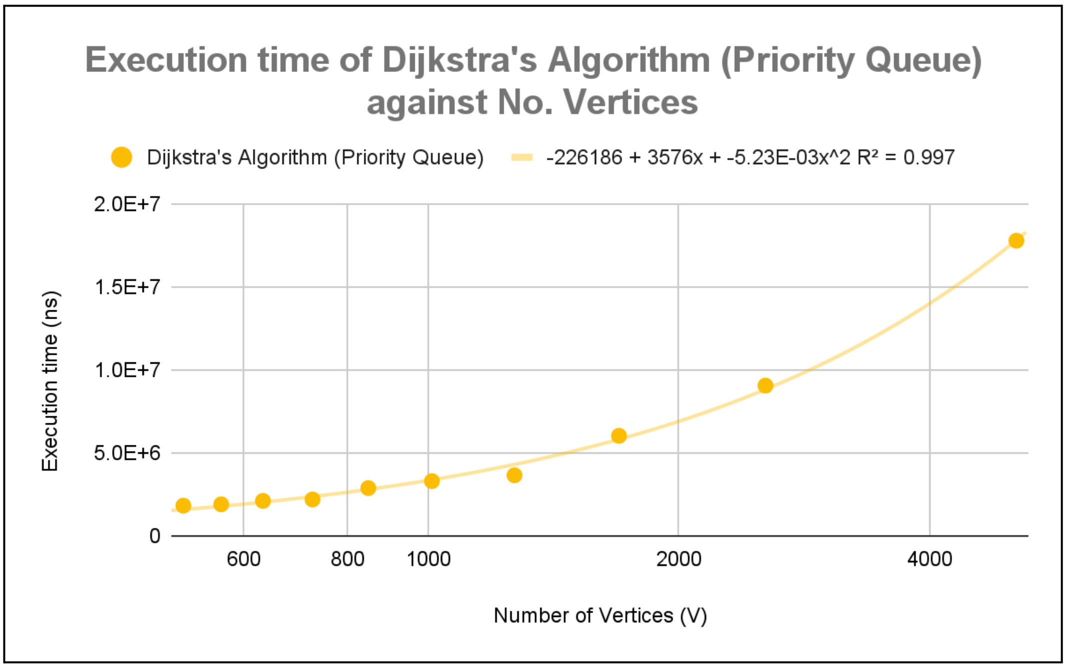


Figure 27: *Execution Time of Dijkstra's Algorithm (Priority Queue) Against the Number of Vertices*

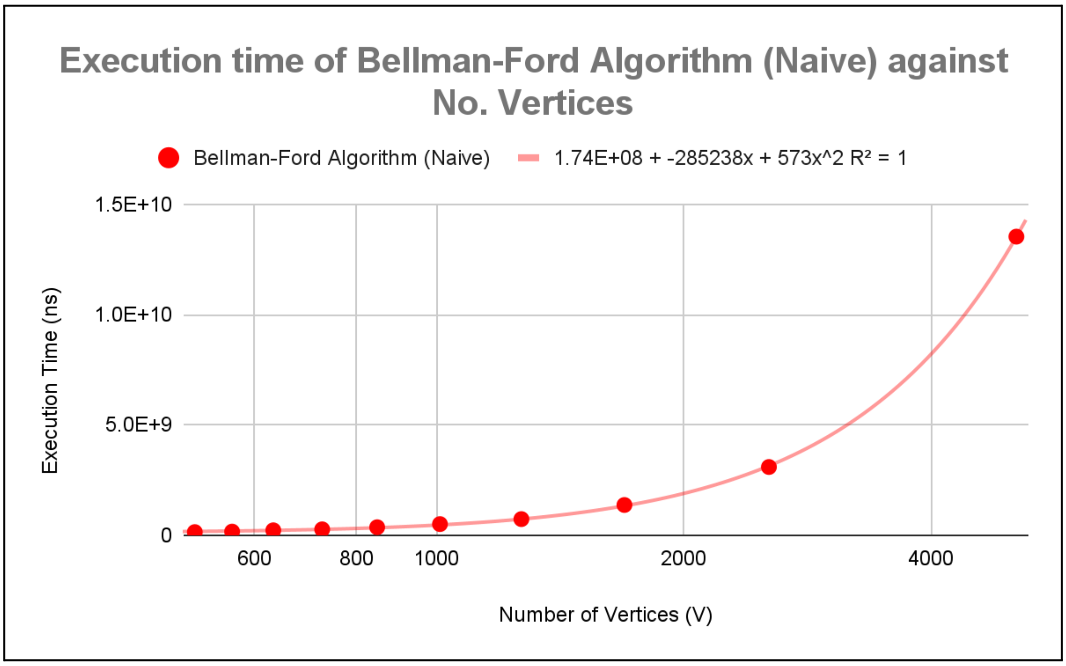


Figure 28: *Execution Time of the Bellman-Ford Algorithm (Naive) Against the Number of Vertices*

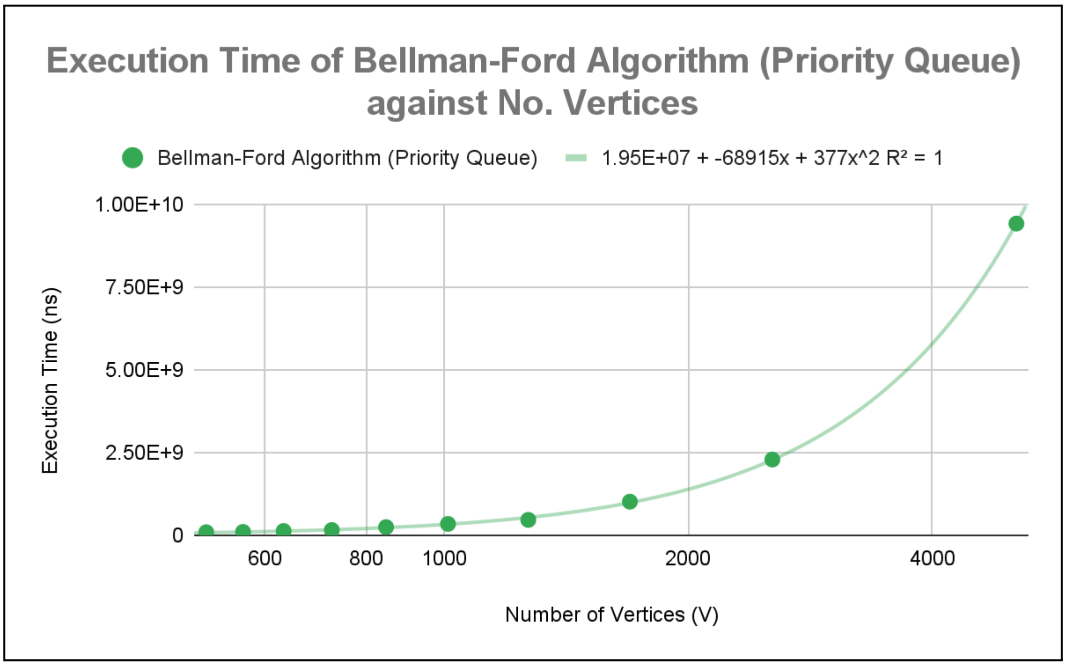


Figure 29: *Execution Time of the Bellman-Ford Algorithm (Priority Queue) Against the Number of Vertices*

5.3 Data Analysis

5.3.1 Analyzing Dijsktra's Algorithm

The hypothesis regarding the quadratic relationship between x and y has proven to be correct for the naive implementation. The R^2 value in **Figure 26** of exactly 1 indicates that the data points are a perfect fit to the quadratic curve. The equation is given in **Figure 26** can be re-written in the general form $ax^2 + bx + c$ as $63.3x^2 - 9042x + 8.43 \times 10^6$. The positive a coefficient suggests that as the number of vertices in the graph increases, the execution time of the algorithm will increase quadratically. The negative b value suggests that as the number of vertices increases, the execution time of the algorithm will decrease linearly (by a relatively small amount).

The hypothesis regarding the logarithmic relationship for the priority queue implemented algorithm is partially correct. **Figure 27** shows the trend line to have equation $(-5.23 \times 10^{-3})x^2 + 3576x - 226186$. The negligible a value for this equation suggests that the relationship between x and y is more linear. As mentioned previously, the time complexity for the priority queue implemented Dijkstra's Algorithm is $\mathrm{O}((\mathrm{V}+\mathrm{E})\log_2 \mathrm{V})$, using the previous approximation $|\mathrm{V}| \times 1.5 \approx |\mathrm{E}|$, the time complexity can be written as $\mathrm{O}(2.5\mathrm{V}\log_2 \mathrm{V})$. Graphing this out against $y = x$, it is observed that the time complexity is superlinear (**Figure 30** below). A superlinear relationship is a non-linear function that appears to grow linearly.

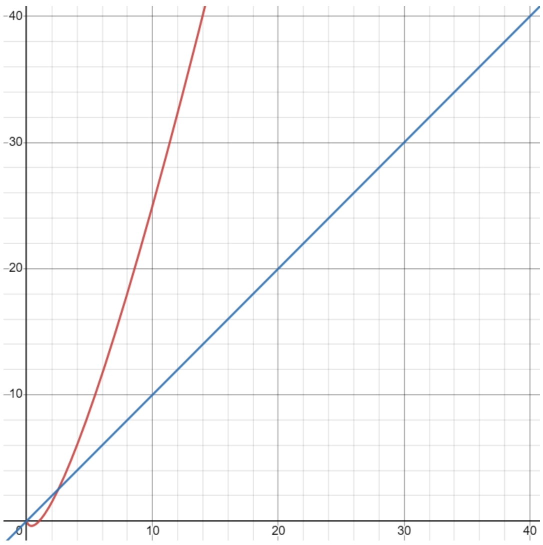


Figure 30: *Graph of* $\mathrm{O}(2.5\mathrm{V}\log_2\mathrm{V})$ *in Red and* $y = x$ *in Blue* (Desmos)

The relationship between x and y is indeed logarithmic, but appears to be linear. This can be due to the relatively small input size for the algorithm; the graph used to benchmark the algorithm contained not as many vertices and edges needed to observe a clear logarithmic relationship. Furthermore, the graph used was sparse, which was in favor of the algorithm since there were fewer edges to explore. However, for the size of the dataset used, the relationship can be better modeled using a linear function.

Looking back at **Figure 25**, taking the average execution times for both algorithms for the entire graph (5083 vertices) and applying the percentage decrease formula:

$$\frac{\text{initial - final}}{\text{initial}} \times 100$$

$$\frac{1598144270 - 17772930}{1598144270} \times 100$$

Which is approximately **98.9%**. It is evident that the min heap priority queue implementation has greatly optimized Dijkstra's Algorithm, resulting in significantly improved execution times.

5.3.2 Analyzing Bellman-Ford Algorithm

The hypothesis regarding the quadratic relationship has proven to be correct for the naive implementation. The R^2 value in **Figure 28** indicates that the data points are a perfect fit to the quadratic curve. It behaves similarly to Dijkstra's Algorithm (naive), however, with a longer execution time overall.

The hypothesis regarding the logarithmic relationship for the priority queue implementation was incorrect, as the $R^2 = 1$ value shown in **Figure 29** suggests that the data points perfectly fit the shape of a quadratic curve. As a consequence of the sparse graph, the number of priority queue operations is performed a relatively smaller number of times compared to relaxation operations and iterations. As fewer edges need to be considered during each iteration, the $O(\log_2 V)$ component of the time complexity becomes negligible, which results in a more quadratic relationship.

The a value of a quadratic represents how wide/narrow the parabola is, it can be interpreted as the rate at which the execution time increases, a higher a value exemplifying a steeper increase in execution time against the number of vertices. Using the percentage decrease formula with the a values as the input, it is observed that the Bellman-Ford Algorithm decreased in execution time by about **34.2%** after the priority queue was implemented. This is due to several reasons. With the implementation of a priority queue, the algorithm can terminate early once all shortest paths have been found, it guarantees that once a vertex's shortest distance is finalized, it will not be updated further. Furthermore, relaxation operations are optimized as the vertex with the smallest distance is always selected first, which reduces the number of comparisons required, thus improving execution time.

Applying the second derivative test on the trendlines in **Figures 28** and **29**, the rate of execution time change with input size increase is determined, aiding in identifying the algorithm with lower execution times as input size expands.

Trendline in **Figure 28**:	Trendline in **Figure 29**:
$y = 573x^2 - 285238x + 1.74 \times 10^8$	$y = 377x^2 - 68915x + 1.95 \times 10^7$
$\frac{dy}{dx} = 1146x - 285238$	$\frac{dy}{dx} = 754x - 68915$
$\frac{d^2y}{dx^2} = 1146$	$\frac{d^2y}{dx^2} = 754$

As $1146 > 754$, it is concluded that the Bellman-Ford algorithm with a priority queue is more efficient and scalable as input size increases.

6. Limitations

There were various experiment-related limitations that could potentially have impeded the achievement of better results.

The graph used in this experiment is Singapore's Bus Network. In **Figure 22**, it can be seen that each vertex only points towards one other vertex, indicating the graph was sparse. Empirically, most real-world graphs are sparse by nature. The number of edges is within a constant multiple of the number of vertices (Cook). The main issue is the algorithms may not encounter enough complexity to demonstrate the improvement of priority queue implementation. There were less relaxation and priority queue operations overall, which can be why all the algorithms mostly exhibited quadratic time complexity behaviors.

As Singapore is a small country, the graph contained only 5083 vertices and 7420 edges. This became an issue when collecting results for Dijkstra’s Algorithm with a priority queue, as the relationship was observed to be initially linear when graphed without using a log scale (**Figure 31**). The use of a graph with more vertices (and preferably more edges) is speculated to eliminate this issue.

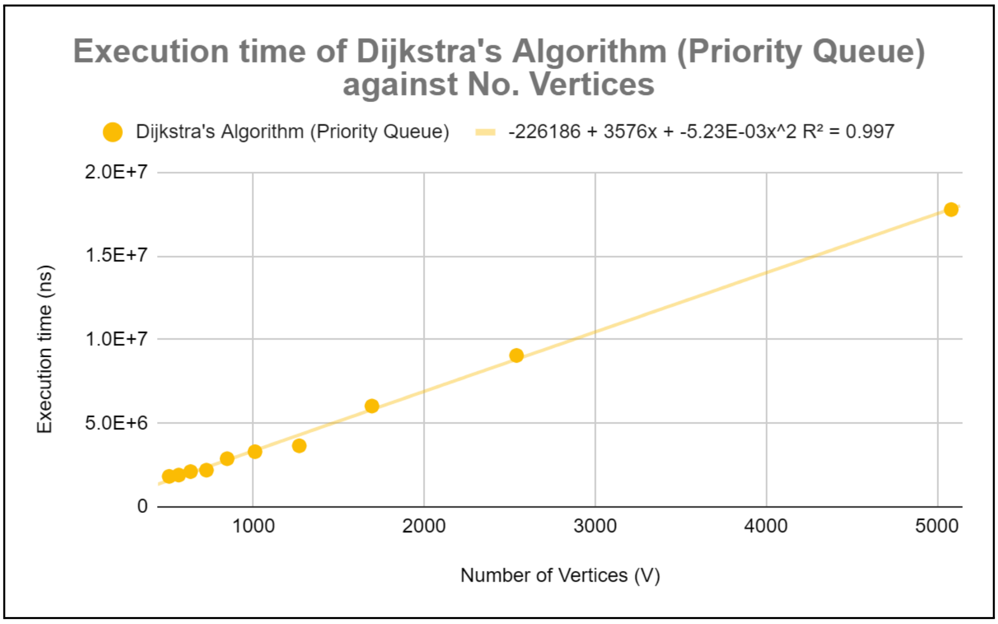


Figure 31: *Figure 27 without a Log Scale*

Hardware limitations can influence execution time, potentially causing fluctuations in results. The CPU of the laptop used was also being utilized by other processes such as Windows service host processes, which adds random error to the algorithm execution time. This is an inherent limitation to the device that was used in this experiment, as it is not meant to be a dedicated computing system for intricate algorithmic calculations.

7. Conclusion

The results clearly show that with the implementation of a min heap priority queue, both the Bellman-Ford and Dijkstra's Algorithm reduce in time complexity and execution time.

By implementing a min heap priority queue into Dijkstra's Algorithm, the selection and extraction of the minimum distance vertex during each iteration improves, reducing the number of relaxations and comparisons. This makes it more scalable as the number of vertices increases in a graph.

The Bellman-Ford Algorithm improves for the same reasons, however, it is not as pronounced due to the nature of the algorithm being dependent on the number of iterations and relaxation operations which overshadows the time complexity in sparse graph environments.

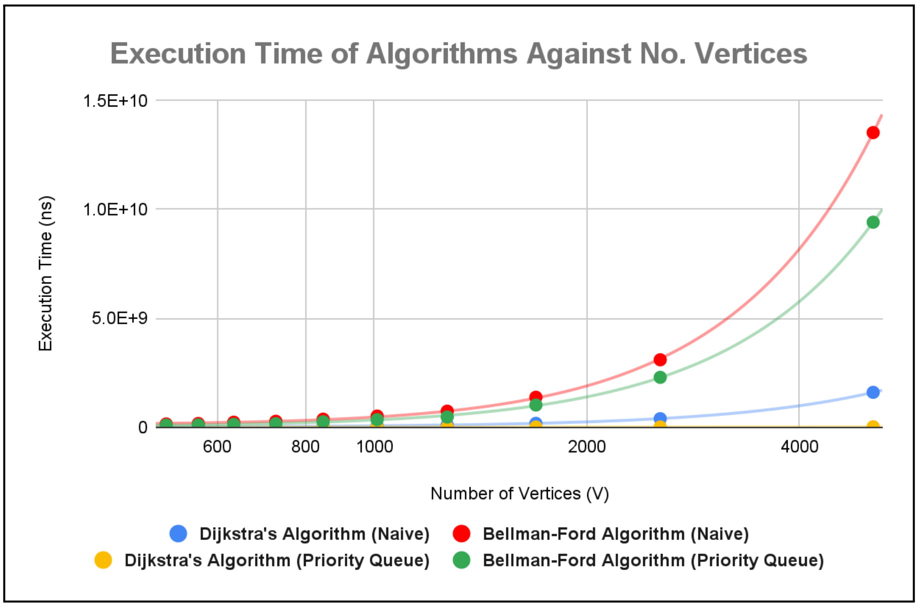


Figure 31: *Combined Graph Showing the Execution Times of All Algorithms Against the Number of Vertices*

Looking at **Figure 31** above, it can be concluded that Dijsktra's Algorithm is better at solving the SSSP problem in weighted graphs compared to the Bellman-Ford Algorithm. This is supported by the steepness of both the red and green lines (Bellman-Ford) as compared to the blue and yellow lines (Dijkstra), the blue and yellow lines are significantly more shallow, suggesting that Dijkstra's Algorithm is more efficient in terms of execution time as the number of vertices increases.

This paper hopes to prove useful to computer scientists collaborating with transportation planners and urban developers, providing them with a deeper understanding of how algorithmic optimizations can lead to more efficient transportation networks in urban environments. This research may serve as a foundational resource for future studies aiming to improve various graph-based algorithms in diverse applications, from network routing to logistics and beyond.

8. Works Cited

Abiy, Thaddeus, et al. “Dijkstra’s Shortest Path Algorithm | Brilliant Math & Science Wiki.” *Brilliant.org*, 2016, brilliant.org/wiki/dijkstras-short-path-finder/. Accessed 25 July 2023.

Ashwani K. “Complete Tutorial on Big O (Big Oh) Notation - DevOpsSchool.com.” *DevOpsSchool.com*, 15 June 2021, www.devopsschool.com/blog/complete-tutorial-on-big-o-big-oh-notation/. Accessed 25 June 2023.

Aun, Chee. “SG Bus Data.” *GitHub*, 26 May 2023, github.com/cheeaun/sgbusdata. Accessed 29 June 2023.

ChatGPT. “Prompt: Pseudocode Naive Bellman Ford using 'distance' dictionary for SSSP. Using graph and source as the parameters. Please do not put any comments.” openai.com, 2023. ChatGPT May. 24 Version. https://chat.openai.com/share/7a25254e-ed33-4cdd-bc6d-211c218619cc. Accessed 22 Jun. 2023.

ChatGPT. “Prompt: Pseudocode Naive Dijkstra using dictionary for SSSP with no comments using a "visited" variable” openai.com, 2023. ChatGPT May. 24 Version. https://chat.openai.com/share/6cf54f75-c948-402e-aca9-2413aec013c2. Accessed 20 Jun. 2023.

ChatGPT. “Prompt: The research question for my IB Computer Science Extneded Essay is "How does the use of a priority queue and the implementation of Bellman-Ford and Dijkstra's algorithm affect the time complexity of solving the shortest path problem in weighted graphs?". For the weighted directed graph, I intend to use Singapore's Bus Network.” openai.com, 2023. ChatGPT Aug. 3 Version. https://chat.openai.com/share/f17c220d-4473-42bb-ae0c-eb426002e334. Accessed 15 Aug.

2023.

Chumbley, Alex, et al. “Bellman-Ford Algorithm | Brilliant Math & Science Wiki.” *Brilliant.org*, brilliant.org/wiki/bellman-ford-algorithm/#:~:text=%5Chspace%7B12mm%7D-. Accessed 2 July 2023.

Cook, John. “Real World Graphs.” *Www.johndcook.com*, 21 July 2013, www.johndcook.com/blog/2013/07/21/three-properties-of-real-world-graphs/. Accessed 20 July 2023.

Desmos. “Desmos | Graphing Calculator.” *Desmos*, www.desmos.com/calculator/azlovhgobb. Accessed 8 Aug. 2023.

Erickson, Jeff. *Algorithms*. Erscheinungsort Nicht Ermittelbar] Jeff Erickson June 13, 2019, p. 273, jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf. Accessed 16 Sept. 2023.

Garg, Prateek. “Heaps and Priority Queues - Prateek Garg.” *HackerEarth*, 2019, www.hackerearth.com/practice/notes/heaps-and-priority-queues/. Accessed 13 July 2023.

Jaludi, Mariam. “Data Structures: Heaps.” *The Startup*, 6 Oct. 2019, medium.com/@mariam.jaludi/data-structures-heaps-b039868a521b. Accessed 5 July 2023.

Magzhan, Kairanbay, and Hajar Mat. “A Review and Evaluations of Shortest Path Algorithms.” *International Journal of Scientific & Technology Research*, vol. 2, no. 6, Jan. 2013, www.ijstr.org/final-print/june2013/A-Review-And-Evaluations-Of-Shortest-Path-Algorithms.pdf. Accessed 11 July 2023.

Morampudi, Shanmukh Chowdary. "Bellman-Ford Algorithm." *Medium*, 25 Mar. 2021, shanmukhchowdary147.medium.com/bellman-ford-algorithm-21ac42a8cf3d. Accessed 28 June 2023.

Nvidia. "Shortest Path Problem." *NVIDIA Developer*, 29 July 2016, developer.nvidia.com/discover/shortest-path-problem. Accessed 20 June 2023.

Perplexity AI. "Prompt: What is a naive implementation of an algorithm?" Perplexity AI, n.d., https://www.perplexity.ai/search/e5d545d8-3b3e-4dba-82fd-f576752f4643?s=c. Accessed 19 Jun. 2023

SaturnCloud. "Relaxation of an Edge in Dijkstra's Algorithm: A Guide for Data Scientists | Saturn Cloud Blog." *Saturncloud.io*, 18 July 2023, saturncloud.io/blog/relaxation-of-an-edge-in-dijkstras-algorithm-a-guide-for-data-scientists/#:~:text=We%27ve%20learned%20that%20relaxation. Accessed 15 Aug. 2023.

stevenard, jb. "Bellman–Ford." *Medium*, 30 Sept. 2022, levelup.gitconnected.com/bellman-ford-6bd907c6c4c0. Accessed 28 June 2023.

Sydorenko, Daniil. "GitHub - DaniilSydorenko/Haversine-Geolocation: Get Distances between Two Points or Get Closest Position to Current Point. Based on the Haversine Formula." *GitHub*, 2020, github.com/DaniilSydorenko/haversine-geolocation. Accessed 16 Sept. 2023.

Team, Great Learning. "What Is Time Complexity and Why Is It Essential?" *GreatLearning Blog: Free Resources What Matters to Shape Your Career!*, 14 July 2022, www.mygreatlearning.com/blog/why-is-time-complexity-essential/#:~:text=Time%20complexity%20is%20defined%20as. Accessed 15 Aug. 2023.

Virginia Tech: Department of Computer Science. "Undirected Weighted Graph Containing 9 Vertices." *Virginia Tech*, 3 Dec. 2009, courses.cs.vt.edu/~cs3114/Fall10/Notes/T22.WeightedGraphs.pdf. Accessed 3 June 2023.

YOON MI KIM. “Dijkstra Algorithm: Key to Finding the Shortest Path, Google Map to Waze.” *Medium*, Medium, 13 June 2019, medium.com/@yk392/dijkstra-algorithm-key-to-finding-the-shortest-path-google-map-to-waze-56ff3d9f92f0. Accessed 24 June 2023.

9. Appendices

Appendix A: Python Code

The code was initially generated using the Large Language Model (LLM) "ChatGPT", however, underwent further modifications by the author (ChatGPT, 2023). Below is a table containing the significant prompts and their respective explanation as to why they were entered.

Prompt	Explanation
The research question for my IB Computer Science Extneded (sic) Essay is "How does the use of a priority queue and the implementation of Bellman-Ford and Dijkstra's algorithm affect the time complexity of solving the shortest path problem in weighted graphs?". For the weighted directed graph, I intend to use Singapore's Bus Network.	This was the first prompt fed into the LLM. The purpose of this prompt was to introduce ChatGPT, the core research question of the essay as well as provide context for the rest of the conversation.

There are two files you will need, "services.json" and "stops.json". Below are sample data from each file.
 	
services.json
{
    "2": {
        "name": "Changi Village Ter ⇄ Kampong Bahru Ter",
        "routes": [
            [
…
 	
stops.json
{
    "10009": [
        103.81722,
        1.2821,
        "Bt Merah Int",
        "Bt Merah Ctrl"
    ],
…
	The purpose of this prompt was to introduce the data files "services.json" and "stops.json", which was the information used to construct the weighted digraph.
	
	Sample data was provided from both files which enabled the LLM to understand how to deserialize the files into Python objects.

Prompt	Explanation
I want you to do this in python, and use the networkx library to create the digraph, and calculate the weights using the haversine distance	The purpose of this prompt was to give the specification that the program should be created in Python. It also specifies that the digraph is to be created with NetworkX, as opposed to igraph or PyGraphviz. It also specifies for the weights of the digraph to be created by applying the haversine formula on the latitude and longitude values.
Let me explain what I need fully: 1. Parsetops.json and services.json in python objects using the json library 2. Create a dictionary of stop IDs to coordinates 3. create a function to calculate haversine distance, with the paramters being coord1 and coord2 4. create an empty digraph using networkx 5. add edges to the graph with weights 6. Create the following algorithms: Dijkstra's Algorithm Naive, Dijkstra's Algorithm with a min heap priority queue, Bellman-Ford Algorithm Naive, Bellman-Ford Algorithm with a min heap priority queue. All of these algorithms will solve the SSSP problem. Implement the min heap priority queue using "from queue import PriorityQueue" 7. Measure the execution time of each algorithm in nanoseconds using time.perf_counter_ns() from the time library	This lengthy prompt helped generate most of the code, I gave instructions in the order of how I wanted them to be implemented within the program. In steps 1-5, I explain to the LLM how the digraph should be constructed using the stops.json and services.json file. These steps involve parsing the data, mapping stop IDs to coordinates, calculating distances, creating the digraph, and adding weighted edges to represent the transportation network. In steps 6 and 7, I ask the LLM to write the implementations for the naive and priority queue-implemented algorithms, as well as the specification to measure the execution time using the time library in Python.

# Measure the execution time of each algorithm in nanoseconds 	
source_stop = "10009" # Example source stop ID
target_stop = "10041" # Example target stop ID
	The purpose of this was to correct the LLM, as it initially thought that the problem to be solved was the Single-Pair Shortest Path (SPSP), which finds the shortest path only between a single pair of vertices.

Prompt	Explanation
since i am intending the alogirhtms (sic) to solve the SSSP problem, a target stop wont be needed.	However, the problem being focused on in this essay was the SSSP, therefore, ChatGPT was corrected accordingly.

Below is the Python code used in the experiment to benchmark the algorithms and collect the raw data for the execution times in nanoseconds.

```python
# filename: ee.py
import json
import networkx as nx
import math
from queue import PriorityQueue
import time

# Constants for Haversine formula
AVG_EARTH_RADIUS = 6371  # in kilometers

# Step 1: Parse services.json
with open('services.json') as services_file:
    services_data = json.load(services_file)

# Step 2: Parse stops.json
with open('stops.json') as stops_file:
    stops_data = json.load(stops_file)

# Step 3: Create a dictionary of stop IDs to coordinates
stop_coordinates = {}
for stop_id, stop_info in stops_data.items():
    longitude, latitude, _, _ = stop_info
    stop_coordinates[stop_id] = (float(latitude), float(longitude))

# Function to calculate Haversine distance
def haversine_distance(coord1, coord2):
    lat1, lon1 = coord1
    lat2, lon2 = coord2

    dlat = math.radians(lat2 - lat1)
```

```
    dlon = math.radians(lon2 - lon1)

    a = math.sin(dlat / 2) * math.sin(dlat / 2) + math.cos(
        math.radians(lat1)) * math.cos(math.radians(lat2)) * math.sin(
        dlon / 2) * math.sin(dlon / 2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    distance = AVG_EARTH_RADIUS * c
    return distance

# Step 4: Create an empty directed graph
graph = nx.DiGraph()

# Step 5: Add edges to the graph with weights
stop_ids = list(stop_coordinates.keys())[:len(stop_coordinates)//1]

for service_id, service_info in services_data.items():
    routes = service_info['routes']
    for route in routes:
        for i in range(len(route) - 1):
            start_stop_id = route[i]
            end_stop_id = route[i + 1]

            if start_stop_id in stop_ids and end_stop_id in stop_ids:
                start_coordinates = stop_coordinates[start_stop_id]
                end_coordinates = stop_coordinates[end_stop_id]
                distance = haversine_distance(start_coordinates,
end_coordinates)
                graph.add_edge(start_stop_id, end_stop_id,
weight=distance)

# Dijkstra's algorithm (without priority queue)
def dijkstra(graph, start_node):
    distances = {node: float('inf') for node in graph.nodes}
    distances[start_node] = 0

    visited = set()

    while len(visited) < len(graph.nodes):
        current_node = min((node for node in graph.nodes if node not in
visited), key=distances.get)
```

```python
        visited.add(current_node)

        for neighbor, edge_data in graph[current_node].items():
            weight = edge_data['weight']
            distance = distances[current_node] + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance

    return distances

# Bellman-Ford algorithm (without priority queue)
def bellman_ford(graph, start_node):
    distances = {node: float('inf') for node in graph.nodes}
    distances[start_node] = 0

    for _ in range(len(graph.nodes) - 1):
        for u, v, edge_data in graph.edges(data=True):
            weight = edge_data['weight']
            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight

    return distances

# Dijkstra's algorithm with a priority queue
def dijkstra_priority_queue(graph, start_node):
    distances = {node: float('inf') for node in graph.nodes}
    distances[start_node] = 0

    pq = PriorityQueue()
    pq.put((0, start_node))

    while not pq.empty():
        current_distance, current_node = pq.get()

        if current_distance > distances[current_node]:
            continue

        for neighbor, edge_data in graph[current_node].items():
            weight = edge_data['weight']
            distance = current_distance + weight
```

```python

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                pq.put((distance, neighbor))

    return distances

# Bellman-Ford algorithm with a priority queue
def bellman_ford_priority_queue(graph, start_node):
    distances = {node: float('inf') for node in graph.nodes}
    distances[start_node] = 0

    pq = PriorityQueue()
    pq.put((0, start_node))

    while not pq.empty():
        current_distance, current_node = pq.get()

        if current_distance > distances[current_node]:
            continue

        for u, v, edge_data in graph.edges(data=True):
            if u != current_node:
                continue

            weight = edge_data['weight']
            distance = current_distance + weight

            if distance < distances[v]:
                distances[v] = distance
                pq.put((distance, v))

    return distances

print("Number of vertices:", graph.number_of_nodes())
print("Number of edges:", graph.number_of_edges())

# Measure execution time of Dijkstra's algorithm (without priority queue)
start_time = time.perf_counter_ns()
shortest_paths_dijkstra = dijkstra(graph, '10009')
end_time = time.perf_counter_ns()
```

```
execution_time_dijkstra = end_time - start_time  # Time in nanoseconds

# Measure execution time of Bellman-Ford algorithm (without priority queue)
start_time = time.perf_counter_ns()
shortest_paths_bellman_ford = bellman_ford(graph, '10009')
end_time = time.perf_counter_ns()
execution_time_bellman_ford = end_time - start_time

# Measure execution time of Dijkstra's algorithm with priority queue
start_time = time.perf_counter_ns()
shortest_paths_dijkstra_pq = dijkstra_priority_queue(graph, '10009')
end_time = time.perf_counter_ns()
execution_time_dijkstra_pq = end_time - start_time

# Measure execution time of Bellman-Ford algorithm with priority queue
start_time = time.perf_counter_ns()
shortest_paths_bellman_ford_pq = bellman_ford_priority_queue(graph, '10009')
end_time = time.perf_counter_ns()
execution_time_bellman_ford_pq = end_time - start_time

# Print the results
print("Dijkstra's algorithm (without priority queue):")
print("Execution time:", execution_time_dijkstra, "nanoseconds")

print("Bellman-Ford algorithm (without priority queue):")
print("Execution time:", execution_time_bellman_ford, "nanoseconds")

print("Dijkstra's algorithm with a priority queue:")
print("Execution time:", execution_time_dijkstra_pq, "nanoseconds")

print("Bellman-Ford algorithm with a priority queue:")
print("Execution time:", execution_time_bellman_ford_pq, "nanoseconds")
```

Appendix B: Raw Data — Execution Times For Each Algorithm

Below are four tables containing the raw data for the execution times for all both algorithms naively and priority queue implemented.

Dijkstra's Algorithm (Naive)												
Vertices	Edges	Execution Time (nanoseconds)										
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
508	734	11981200	21934400	18719100	18874700	20814800	21334600	19235900	18074100	14620800	17188000	**18277760**
564	816	22998800	23953800	16544800	17178300	17298700	20640300	22623900	16552200	25723100	25728800	**20924270**
633	911	29409900	21111900	28999300	21792300	30336900	29728900	29359800	25534200	29384200	22003700	**26766110**
726	1012	35152900	39225900	37455000	39047500	28401400	32461400	28173600	33840000	30231900	29363500	**33335310**
847	1186	52392900	46105900	53701300	34253700	45261300	48200400	52807600	50324000	52461800	50390200	**48589910**
1010	1363	66005400	66154700	64456600	74033600	64706200	72119000	74553200	53278300	63849000	67543300	**66669930**
1269	1681	95114300	103253100	83637100	114374500	110008000	120537000	109009200	105947500	108108000	101931500	**105192020**
1694	2241	175072500	181875000	170464400	206013000	138782000	192145500	188640400	173174800	139807800	182432900	**174840830**
2540	3454	377145300	365253100	483338100	418696100	383313600	383556500	348133600	319151700	434086500	376034000	**388870850**
5083	7420	1884426500	1658591600	1321206300	1388226400	1547314400	1538802100	1656388100	1574833200	1712586800	1699067300	**1598144270**

Bellman-Ford Algorithm (Naive)												
Vertices	Edges	Execution Time (nanoseconds)										
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
508	734	138645100	144163600	159186900	140928100	136946800	158509300	134072000	144617900	145289700	150328100	**145268750**

Bellman-Ford Algorithm (Naive)												
Vertices	Edges	Execution Time (nanoseconds)										
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
564	816	204098200	203175500	167630900	123132900	147859500	200917900	171318100	116900800	179134100	161555700	**167572360**
633	911	258275300	217986500	235156500	239027400	235583800	215396400	230625000	211644700	191907700	230321100	**226592440**
726	1012	318274600	289944200	295647900	314248900	227696100	255524200	175536900	301477900	295709100	230886200	**270494600**
847	1186	396026400	387980000	306934900	377343200	389627900	357523100	318928100	361835200	356576900	406543600	**365931930**
1010	1363	508380200	549141700	471250500	553523900	496799700	520952800	537441800	512110500	569653100	359358100	**507861230**
1269	1681	811700400	694252700	640477700	745964900	802252600	806862000	623195600	652351200	714230200	851501600	**734278890**
1694	2241	1628369200	1337397300	1370179800	1173142100	1260270300	1441802600	1483032600	1351741800	1309590500	1382545300	**1373807150**
2540	3454	3374066700	3368888000	3098249400	2799032800	2512895700	3315074500	3142171700	3233649900	3320771300	2775178000	**3093997800**
5083	7420	14316089000	13657371800	14277746400	14967629500	12846789100	11772173400	13503970400	13775991300	13590726300	12576360500	**13528484770**

Dijkstra's Algorithm with Priority Queue												
Vertices	Edges	Execution Time (nanoseconds)										
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
508	734	1616000	1850700	1514500	1852300	2562500	1629000	1911800	1728600	1849000	1722900	**1823730**
564	816	2181800	2178300	1649100	1929400	2415600	2120200	1734700	1507700	1665800	1645400	**1902800**
633	911	2485600	2140200	2187800	1988500	2269300	1820100	2218700	1664800	2225200	2094800	**2109500**
726	1012	2210600	2258500	2333200	2435600	1865200	2706800	1697000	2435800	2342000	1664700	**2194940**

Dijkstra's Algorithm with Priority Queue												
Vertices	Edges	Execution Time (nanoseconds)										
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
847	1186	2835000	2982200	2416200	3614000	3014100	2994600	2865700	2146000	3449200	2480200	**2879720**
1010	1363	3438600	3431200	2537200	3647500	4102400	2670200	3663100	3354200	3390800	2758500	**3299370**
1269	1681	3219500	3305300	3385900	4357900	4314800	3573500	3373500	3403100	3248000	4329100	**3651060**
1694	2241	6791700	6055800	6044600	5901200	5938000	5798500	5712800	5795700	5841600	6392100	**6027200**
2540	3454	7303900	10653200	9176900	7071600	9016100	11185800	7299700	10976000	8902600	8925700	**9051150**
5083	7420	20586400	20269600	18011600	14750500	19908800	14880600	14594100	20048000	19451300	15228400	**17772930**

Bellman-Ford Algorithm with Priority Queue												
Vertices	Edges	Execution Time (nanoseconds)										
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
508	734	89220700	99353200	65564300	86732600	109478200	98060300	98086700	91262900	96033500	92988100	**92678050**
564	816	121085600	128722100	115316400	85109300	133008300	95012400	78248000	83066600	105363400	82532400	**102746450**
633	911	150521800	148400500	152023300	103652400	154740200	123229300	156813600	110479600	109957600	105607300	**131542560**
726	1012	130372600	178411600	183438200	179431300	139177600	147948500	162878800	180355000	189677800	104775200	**159646660**
847	1186	274182800	270720700	262788300	260363300	257145000	214754900	268824600	164592900	211039500	293480700	**247789270**
1010	1363	377515000	378410000	384107200	328014100	384777200	307763200	229923900	355940000	333227000	364276900	**344395450**
1269	1681	408273800	413298300	519683400	564734100	553576900	484097200	374041300	572099200	382550700	390627600	**466298250**
1694	2241	113576	113024	103404	930110	102354	105627	961803	105721	902341	927181	**1015853050**

Bellman-Ford Algorithm with Priority Queue												
Vertices	Edges	Execution Time (nanoseconds)										
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
		2200	1700	7300	200	9500	8400	400	4900	400	500	
2540	3454	2236027300	2366220200	1901060900	1893840400	2242223900	1983644200	2671764600	2075174100	3224573400	2283632000	**2287816100**
5083	7420	10483894700	9170248900	8581951000	9311216500	9922697100	8918218400	8755219000	10272326900	10390275600	8319104300	**9412515240**

Appendix C: Definitions and Key Terms

Below are some technical terms used in this essay that can be used to support the reader's comprehension:

Time Complexity: Time complexity is defined as the amount of time taken by an algorithm to run, as a function of the length of the input (Team).

Single-Source Shortest Path (SSSP): [The] Single-source shortest path (or SSSP) problem requires finding the shortest path from a source vertex to all other vertices in a weighted graph (Nvidia).

All-Pairs Shortest Path (APSP): [The] All-pairs shortest path (or APSP) problem requires finding the shortest path between all pairs of vertices in a graph (Nvidia).

Greedy Algorithm: A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem (YOON MI KIM).

Naive Algorithm: A naive implementation [of an algorithm] is a programming technique that prioritizes imperfect shortcuts for the sake of speed, simplicity, or lack of knowledge (Perplexity AI. "Prompt: What is a naive implementation of an algorithm?").

Relax/Relaxing/Relaxation: Relaxation is the process of updating the distance of a [vertex] if a shorter path is found (SaturnCloud).

Big-O Notation: Big O notation is a mathematical notation describing a function's limiting behavior when the argument goes towards a certain value or infinity (Ashwani K).

Heap: A Heap is a complete binary tree-based data structure (Jaludi).