BINUS International

Implementation and comparison of Floyd warshall and Johnson's algorithm

REPORT

Submitted by:

Arvin Yuwono, Christopher Owen, Peter Nelson Subrata, Philipus Adriel Tandra

Department of Computer Science, Bina Nusantara University

COMP6049001

Dr. Maria Seraphina Astriani, S.Kom., M.T.I.

January 11, 2023

TABLE OF CONTENTS

I. INTRODUCTION

Travelling is one of the most frequent activities that humans do in their daily lives. People go from one place to another to get things done in order to continue their lives. Because of the advancement of technology, all people can go anywhere by using the maps that are in the palm of their hands. Their high-technology smartphones can bring them anywhere they want, as long as there are signals for the phones to receive. Mobile map applications rely on quota signals and the location emitted to track the phone where it is and where it is going. Some examples of map applications are google maps, Waze, and other applications similar to them. These mobile maps have developers implementing a certain algorithm in order to give the best outcome to the users when using their apps. According to Postoast, Google Maps use Dijkstra's algorithm and A* algorithm to find the shortest route for users to go somewhere (Adak - et al., 2022). This is an example of computer science developers choosing the best algorithm for all users facing different problems, which is different routes in this specific case. Not only algorithms, but these digital maps also learn from the user's activities through the phone. According to Medium, Waze uses the historical data of the user and real-time updates to consider which routes are the best for users to use (Medium, 2020).

Based on the examples above, different map applications use different algorithms and implementations which can give different routes for users. Even though it is a subjective view on which map is the best, many algorithms were tried and implemented to give the best outcome. This is why this study aims to learn and implement the most effective algorithms to use for an all to all pairs implementation in a map application, this includes speed, memory, scalability, and ease of use. The tools that are used are the gmplot library, pandas, and the time module to analyze the algorithms. This study will compare Floyd Warshall and Johnson's algorithms using

an adjacency matrix as input, as well as implement a working solution to finding the shortest paths between many locations all at once. Briefly, the Floyd Warshall algorithm is a dynamic programming algorithm, while Johnson's algorithm is a hybrid algorithm that combines elements of both the Bellman-Ford algorithm and Dijkstra's algorithm. With the difference between these algorithms, the research result will determine which algorithm is faster, easier, memory-efficient, and scalable. Live preview: https://pewterzz.github.io/BINUS-Locator/html/index.html

II. LIBRARIES AND FRAMEWORKS USED

DJANGO

Note: You don't have to read through this part as it is mostly not that related to the actual topic for algorithm analysis. Proceed to just the gmplot library if you want to.

Django is a high-level Python web framework that enables the rapid development of secure and maintainable websites. It takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel.

Django follows the Model-View-Template (MVT) architectural pattern. In this pattern, the model represents the data, the view represents the user interface, and the template defines how the interface should be displayed.

One of the key features of Django is its automatic admin interface, which makes it easy to manage your data and content. Django also includes several security features out of the box, such as cross-site request forgery protection and SQL injection protection.

Some of the specific files and commands in Django include:

- **urls.py**: This file defines the URL patterns for the Django project. It maps URLs to specific views and functions in your code.
- **settings.py**: This file contains settings and configurations for the Django project. It controls things like the database connection, installed apps, and middleware.

- **manage.py**: This script is used to manage the Django project. It provides a number of commands for performing tasks such as starting the development server, creating new projects, and running tests.

- **python manage.py runserver**: This command starts the development server for the Django project. You can use this command to test your project locally before deploying it to a production server.

- **python manage.py livereload**: Using this command in conjunction with runserver will provide live updates when code has been changed. You can use this command to test your project locally before deploying it to a production server.

GM-PLOT

The gmplot library is a Python library that allows you to generate Google Maps plots. It provides several functions that allow you to customize the appearance of the map and add various types of data such as points, lines, and polygons.

Here is a list of the main functions that we used provided by the gmplot library, along with a brief description of each:

- **GoogleMapPlotter:** This function creates a GoogleMapPlotter object that represents a Google Map. You can specify the latitude and longitude of the center of the map, as well as the zoom level.

- **plot:** This function plots a polyline to the map drawing a line between two points. You can specify the latitude and longitude of each point, as well as the color and size of the marker.

- **scatter:** This function adds a set of points to the map, using a scatter plot style. It is what is used to draw the circles on the map that we can draw a line to. You can specify the latitude and longitude of each point, as well as the color and size of the marker.

- **draw:** This function generates the HTML code for the map, which can be saved to a file or displayed in a web browser.

In addition to these functions, the gmplot library also provides several options that can be used to customize the appearance of the map. For example, you can set the map type (roadmap, satellite, hybrid, or terrain), the zoom level, the center point, the width and height of the map, and the language of the map labels.

III. METHOD

We want to show a comparison between Floyd Warshall algorithm and johnson's algorithm by comparing their growth rate when using different inputs. At the end of the report, we want to prove what is the most effective way of finding the shortest path between all pairs of a graph at the same time.

IV. MEASUREMENT

We will be conducting some time and space complexity analysis as well as measuring the running times of each of the algorithms under different inputs. We will also attempt to find a reason as to which one might be better than the other.

V. THEORY OF THE ALGORITHM

Floyd Warshall

The Floyd Warshall algorithm is an algorithm for finding the shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices.

The Floyd Warshall algorithm is an example of dynamic programming and was published by Robert Floyd in 1962. It is also known as the Floyd-Warshall algorithm, the Floyd-Warshall-Roy–Floyd algorithm, or the Roy–Floyd algorithm.

Here is how the algorithm works:

1. We will Initialize a distance matrix dist where dist[i][j] is the distance between vertex i and vertex j. Set all the distances to infinity, except the distance from each vertex to itself, which is set to 0.

2. Loop through each vertex k as an intermediate vertex. For every pair of vertices (i, j), if the distance from i to k plus the distance from k to j is less than the current distance from i to j, update the distance from i to j to be the new lower distance.

3. Repeat step 2 until the distances stop changing (that is, the algorithm converges).

The Floyd Warshall algorithm has a time complexity of O(n^3), where n is the number of vertices in the graph. This makes it slower than some other algorithms for finding shortest paths, such as Dijkstra's algorithm, which has a time complexity of O(n^2). However, the Floyd Warshall algorithm has the advantage of being able to handle

graphs with negative edge weights, as long as they do not contain negative cycles (cycles whose edges sum to a negative value).

VI. MAIN CODE EXPLANATION

We will provide a somewhat detailed explanation of how the code from this repository works (more detailed explanations can be found in the comments in this repository so feel free to check it out and try it yourselves): https://github.com/PewterZz/BINUS-Locator/tree/updates

There are two main functions in our program that are used to show Johnson's and Floyd Warshall's algorithms respectively; it also contains a node class and many HTML outputs.

We will first be explaining the custom Python class called **Node**. The class has several attributes, including an ID, location, type, destination nodes, distances to those destination nodes, and a chain of nodes.

The class has several methods, including:

- **__init__()**: This is the constructor method, which is called when a new Node object is created. It sets the initial values for the attributes of the object.
- **isrestaurant()**: This method returns True if the type attribute of the Node object is 'restaurant', and False otherwise.
- **isrecreational()**: This method returns True if the type attribute of the Node object is 'recreational', and False otherwise.
- **getdest()**: This method returns the value of the dest attribute of the Node object.
- **getlat()**: This method returns the latitude component of the loc attribute of the Node object.

- **getlong()**: This method returns the longitude component of the loc attribute of the Node object.
- **getdist()**: This method returns the value of the dist attribute of the Node object.
- **getchain()**: This method returns the value of the chain attribute of the Node object. The chain represents where the node will eventually lead to. The final destination/s of the current node.

The first main program we will talk about is **mainFloyd.py**:

We are using the Python libraries os, gmplot, pandas, time, and floyd. We are also using a custom Node class defined elsewhere in the code in another file as well as the Floyd Warshall algorithm also in another file Node.py and floyd.py respectively.

The code performs the following actions:

1. It reads data from a CSV file named coords23.csv using the pandas library.
2. It creates a Node object for each row of data in the CSV file, and stores these objects in a list called coords.
3. It initializes an adjacency matrix using the initialize() function and the coords list. This adjacency matrix is used to represent the connections between the nodes.
4. It replaces any zeros in the adjacency matrix with the value float('inf'), which represents infinity.

5. It runs the Floyd-Warshall algorithm on the adjacency matrix using the floyd_warshall() function from the floyd.py file. This algorithm is used to find the shortest paths between all pairs of nodes in the graph.
6. It prints out the shortest path in reverse order.
7. It calculates the total time taken to run the Floyd-Warshall algorithm.
8. It uses the gmplot library to plot the locations of the nodes on a Google Map.

Some specific files and commands used in this code include:

- **coords23.csv**: This is the CSV file that contains the data for the nodes in the graph.
- **pandas.read_csv()**: This function is used to read the data from the CSV file into a Pandas DataFrame.
- **Node**: This is the custom class that represents a node in the graph. It has several attributes, including an ID, location, type, destination nodes, distances to those destination nodes, and a chain of nodes.
- **initialize()**: This function takes in a list of Node objects and the total number of nodes, and returns an adjacency matrix representing the connections between the nodes.
- **floyd.floyd_warshall()**: This function is imported from the floyd.py file and is used to find the shortest paths between all pairs of nodes in the graph. It takes an

adjacency matrix as an argument and returns two values: a matrix of distances and a matrix of predecessors.

- **gmplot.GoogleMapPlotter()**: This function is part of the gmplot library and is used to plot the locations of the nodes on a Google Map. It takes several arguments, including the latitude and longitude of the center of the map, the zoom level, and an API key.

The next code to talk about is the main.py file which is the implementation of the johnson algorithm. It also uses these imports: os, gmplot, pandas, time, and johnson.

The code performs the following actions:

1. It reads data from a CSV file named coords23.csv using the pandas library.
2. It creates a Node object for each row of data in the CSV file, and stores these objects in a list called coords.
3. It initializes an adjacency matrix using the initialize() function and the coords list. This adjacency matrix is used to represent the connections between the nodes.
4. It runs the Johnson algorithm on the adjacency matrix using the JohnsonAlgorithm() function from the johnson library. This algorithm is used to find the shortest paths between a specified source vertex and all other vertices in the graph.
5. It calculates the total time taken to run the Johnson algorithm.

6. It uses the gmplot library to plot the locations of the nodes on a Google Map.

7. It provides a menu for the user to choose between finding the shortest path to a restaurant, sports and recreational facilities, or other types of locations.

8. For each chosen location type, it plots the shortest path to each location of that type on Google Maps and marks it with a label.

Some specific files and commands specifically used in this code include:

-johnson.JohnsonAlgorithm(): This function is imported from the johnson.py file and is used to find the shortest paths between a specified source vertex and all other vertices in the graph. It takes an adjacency matrix, an empty distance and path array, and the number of vertices in the graph as arguments, and returns the shortest path array.

VII. ALGORITHM CODE EXPLANATION

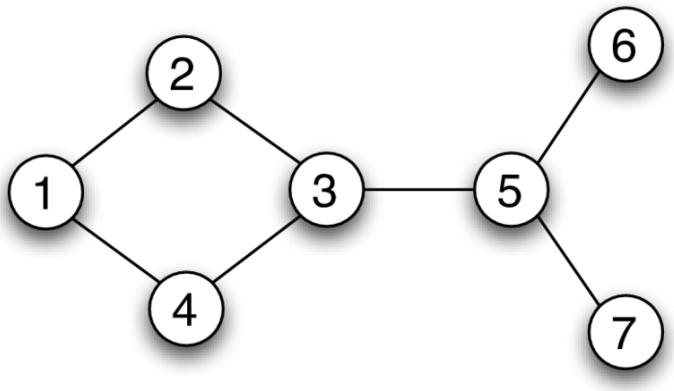
Before we start explaining the algorithms let us explain a little bit about graphs and adjacency lists and adjacency matrices.

A graph is a data structure that consists of a finite set of vertices (or nodes) and a set of edges connecting these vertices. The edges can be directed (meaning they have a specific starting vertex and ending vertex) or undirected (meaning they have no specific starting or ending vertex and can be traversed in either direction).

There are two main ways to represent a graph in a computer: using an adjacency list or an adjacency matrix.

An adjacency list is a way to represent a graph as an array of linked lists. Each element of the array represents a vertex in the graph and the linked list at that element represents the edges coming out of that vertex. For example, consider the following undirected graph:

Image 1 An example of an undirected graph



The adjacency list representation of this graph in python would be made using a dictionary like this,

{ 1 : [2,4], 2 : [1,3], 3 : [2,4,5], 4: [1,3], 5 : [3,6,7], 6 : [5], 7 : [5] }

Or it could just be made with the indexes of a matrix being used to identify the nodes,

[[2,4], [1,3], [2,4,5], [1,3], [3,6,7], [5], [5]]

As you can see though they give a good general idea of what the graph looks like, and this is one of the only ways that python can tell what the graph is like. For example, we can learn that at node 1, we can go to nodes 2 and 4, and in node 5, we can go to nodes 3,6 and 7 just like we see in the visual graph.

The other type of representation of a graph and the one that we will more commonly use within our algorithms is the adjacency matrix. An adjacency matrix serves the same purpose as an adjacency list, the only difference is the way it is presented. In an adjacency matrix using the same graph as the one shown above, it will look like this :

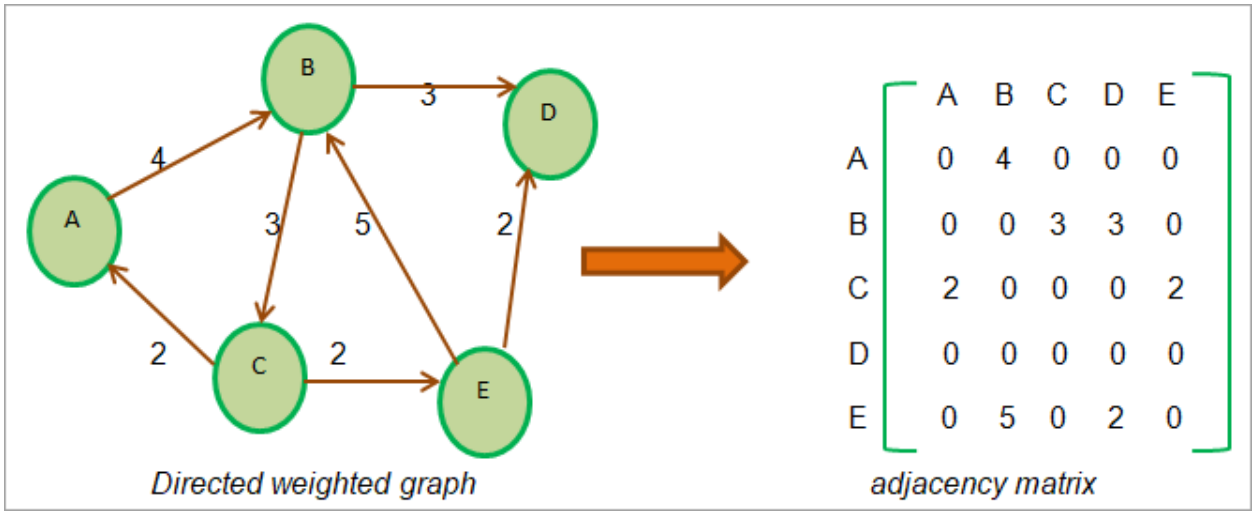
Node	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	[[0 ,	1 ,	0 ,	1 ,	0 ,	0 ,	0],
2	[1 ,	0 ,	1 ,	0 ,	0 ,	0 ,	0]
3	[0 ,	1 ,	0 ,	1 ,	1 ,	0 ,	0]
4	[1 ,	0 ,	1 ,	0 ,	0 ,	0 ,	0]
5	[0 ,	0 ,	1 ,	0 ,	0 ,	1 ,	1]
6	[0 ,	0 ,	0 ,	0 ,	1 ,	0 ,	0]
7	[0 ,	0 ,	0 ,	0 ,	1 ,	0 ,	0]]

From the adjacency matrix above we can resolve the same things that we concluded from the adjacency list. Node 5 is connected to nodes 3, 6 and 7. You can look at node 5 in the matrix either in row or column form as it does not matter in this case since it is an undirected matrix.

The points in the array that have a value of 0 indicate that there is no connection from this node to the node at the index that you are looking at. For example at node 6, the values of the array are mostly 0 except for the one at index 5, this is because node 6 only connects to node 5 and nothing else.

This is only the undirected and unweighted version of the adjacency matrix though. In our case we use both a directed and weighted graph so we need a directed and weighted adjacency matrix. The next page will show you what that will look like.

Image 2 An example of a directed weighted graph converted to adjacency matrix



You can read the above adjacency matrix as such: at node C you can go to node A and node E both with a weight of 2. Note that you can only read most directed adjacency matrices from the row side and not from the column side. In a weighted adjacency matrix, a value of 0 represents no connection and any value other than 0 such as 0.1, 15, or 25 means that a connection is present to another node as well as the weight accompanying it. So once again at node E, you can

see in the adjacency matrix that it has a value at its second index for node B which is 5. This means that there is a connection from node E to node B and its value is 5.

Additional note: For adjacency matrices, to represent an empty node we can use 0 or INF (a really big number). We need to represent the empty connections as INF sometimes because in the case of the Floyd Warshall algorithm, it needs these INF values to be used as its conditional at the core of the algorithm. The conditional will check if the graph node currently is the shortest path by adding up two nodes specifically going in that direction. For example, if there is a connection between node 1 and node 2 it will be not INF but instead, a normal value such as 1 or 9, and the conditional will check whether or not this value is smaller than the current value in its modified graph which always is because its INF.

Now we can start talking about the two main algorithms that we have chosen to implement and compare:

Floyd Warshall :

The Floyd Warshall algorithm is an algorithm for finding all the shortest paths in a weighted graph with positive or negative edge weights, but no negative cycles. It is a bottom up dynamic programming algorithm that works by solving for the shortest path between all pairs of vertices in the graph, one pair at a time, and storing the solution in a matrix.

The Floyd-Warshall algorithm works by iteratively relaxing the shortest path estimates for each pair of vertices in the graph since the algorithm is dynamic and a lot of the relaxing is done by using memoization. It does this by considering the paths that pass through each intermediate vertex in the graph and choosing the shortest of these paths as the new shortest path

estimate. The algorithm continues this process until it reaches the final iteration, at which point it has the shortest path estimates for all pairs of vertices.

The time complexity of the Floyd Warshall algorithm is O(V^3), where V is the number of vertices in the graph. This is because the algorithm requires a nested loop to iterate over all pairs of vertices, and each iteration takes O(V) time to compute the shortest path. O(V) is the same as O(n) and O(V^3) is the same as O(n^3), V is just how it is normally used in graphs.

Here is an example of the Floyd Warshall algorithm in Python:

```
def floyd_warshall (graph) :
    # Initialize the distance matrix
    distance = [[float("inf") for _ in range(len(graph))] for _ in range (len(graph))]
    for i in range(len(graph)):
        for j in range(len(graph)):
            if i == j:
                distance[i][j] = 0
            elif graph[i][j] is not None:
                distance[i][j] = graph[i][j]

    # Iterate over all pairs of vertices
    for k in range(len(graph)):
        for i in range(len(graph)):
            for j in range(len(graph)):
                if distance[i][j] > distance[i][k] + distance [k][j]:
                    distance[i][j] = distance[i][k] + distance [k][j]
    return distance
```

It's a fairly simple piece of code that makes a matrix of all INF first and then uses multiple nested loops to access and change the values in this matrix. In the end, the algorithm is going to return this distance matrix which will store the shortest path from all the nodes to all the nodes.

Our algorithm that we use works similarly:

```
# Initialize distances to be the same as the input graph
# O(1)
distances = list(map(lambda i : list(map(lambda j : j, i)), graph))

# Initialize the number of nodes
# O(1)
num_nodes = len(distances)
# Initialize the predecessor matrix
# O(n^2)
predecessors = [[None for _ in range(num_nodes)] for _ in range(num_nodes)]
# Iterate through all pairs of nodes
# O(n^3)
for k in range(num_nodes):
        for i in range(num_nodes):
                for j in range(num_nodes):
                        # Check if going through node k gives a shorter path
                        # between node i and node j
                        # O(1)
                        if distances[i][k] + distances[k][j] < distances[i][j]:
                                distances[i][j] = distances[i][k] + distances[k][j]
                                #predecessors stores the previous value of the node, which
                                will be used later on outside
                                #to get the shortest path direction
                                # O(1)
                                predecessors[i][j] = k
# O(1)
return distances, predecessors
```

The time complexity of our Floyd Warshall algorithm is determined by the triple nested loop, which has a time complexity of O(V^3). All other operations have a time complexity of O(1) and can be ignored in the analysis. Therefore, the overall time complexity of the Floyd Warshall algorithm is O(V^3).

Johnson's:

Johnson's algorithm is an algorithm for finding the shortest paths between all pairs of vertices in a weighted, directed graph. It is a variant of the Bellman-Ford algorithm and has a

time complexity of O(V^2 * E), where V is the number of vertices and E is the number of edges in the graph.

Johnson's algorithm works by first reweighting the edges of the input graph so that it becomes a graph with non-negative edge weights. It then runs the Bellman-Ford algorithm on the reweighted graph to compute the shortest paths between all pairs of vertices. The key difference between Johnson's algorithm and the Bellman-Ford algorithm is that Johnson's algorithm uses a heap data structure to implement the relaxation step (relaxing is when solutions to calculations are saved and reused instead of running the calculations again, Floyd Warshall also does this) of the Bellman-Ford algorithm, which allows it to achieve a better time complexity.

Here is an example of Johnson's algorithm in Python:

```
def johnson(graph):
    # initialize the reweighted graph
    reweighted_graph = [[0 for_in rang(len(graph))] for_in range(len(graph))]
    for i in range(len(graph)):
        for j in range(len(graph)):
            if graph[i][j] is not None:
                reweighted_graph[i][j] = graph[i][j] + h[i] - h[j]
    # Run the Bellman-Ford algorithm on the reweighted graph
    distances, predecessors = bellman_ford(reweighted_graph, 0)
    # Initialize the shortest path distances
    shortest_paths = [[float("inf") for_in range(len(graph))] for_in range(len(graph))]
    for i in range(len(graph)):
        for j in range (len(graph)):
            if graph[i][j] is not None:
                Shortest_paths[i][j] = distances[i] + graph[i][j] - h[i] + h[j]
    return shortest_paths
```

In this example, the input graph is a 2D matrix representing the weights of the edges in the graph. The matrix reweighted_graph is a reweighted version of the input graph with non-negative edge weights. The h array is an auxiliary array used to store the reweighting values for each vertex. The bellman_ford function is an implementation of the Bellman-Ford algorithm.

The shortest_paths array stores the shortest paths between all pairs of vertices in the original graph. The value shortest_paths[i][j] is the shortest path from vertex i to vertex j in the original graph.

Our code works similarly as well with some differences in terms of the path being stored in another array for the main part of the code to read and draw the distances.

VIII. FINDINGS AND DISCUSSION

General Comparisons

To discover the characteristics of the utilized algorithms, it is important that we know how long and how much time the algorithms require for the quantity of input. The time and space complexity of the two algorithms are listed below. Take note that the algorithms utilized for the comparison are optimized versions of their respective algorithms.

- Time Complexity of Optimized Floyd Warshall = O(V^3)
- Space Complexity of Optimized Floyd Warshall = O(V^2)
- Time Complexity of Optimized Johnson's = O(V * E + V log V)
- Space Complexity of Optimized Johnson's = O(V^2)

For the comparison of the Floyd-Warshall and Johnson algorithms, both algorithms used an adjacency matrix and list and were compared with a variety of vertices: 100, 250, and 500.

Table 1 Comparison of Optimized Floyd Warshall and Johnson's using Adjacency Matrix

Input Size (Vertices)	100	250	500
Floyd Time (Sec)	0.005	0.002	0.017
Floyd Space (byte)	184	312	568
Johnson Time (Sec)	0.0005	0.003	0.03

Johnson Space (byte)	184	312	568

For space consumption, there was zero difference between the two algorithms. However, it is interesting to see that while the Johnson algorithm proves to be more time efficient with input size of 100, it is much slower than the Floyd Warshal as the input size increases.

Table 2 Comparison of Optimized Floyd Warshall and Johnson's using Adjacency List

Input Size (Vertices)	100	250	500
Floyd Time (Sec)	0.005	0.002	0.027
Floyd Space (byte)	184	312	568
Johnson Time (Sec)	0.0005	0.001	0.011
Johnson Space (byte)	184	312	568

Similar to the results in the previous table, there was no difference in terms of space for both algorithms at all input sizes. A stark contrast of results from the adjacent matrix was that the Johnson algorithm proves to be more efficient than Floyd Warshal at all input sizes.

Implementation

To discover the characteristics of the utilized algorithms, it is important that we know how long and how much time the algorithms require for the quantity of input. The time and space complexity of the two algorithms are listed below. In the implementation, no modifications are performed on the algorithm.

- Time Complexity of Floyd Warshall in our Implementation = O(V^3)
- Space Complexity of Floyd Warshall in our Implementation = O(V^2)
- Time Complexity of Johnson's in our Implementation = O(V^2 * VE)
- Space Complexity of Johnson's in our Implementation = O(V^2)

Table 3 Comparison of Implemented Floyd Warshall and Johnson's using Adjacency Matrix

Input Size (Vertices)	10	25	55
Floyd Time (Sec)	0.005	0.002	0.017
Floyd Space (byte)	184	312	568
Johnson Time (Sec)	0.0005	0.003	0.03
Johnson Space (byte)	184	312	568

Table 4 Comparison of Implemented Floyd Warshall and Johnson's using Adjacency List

Input Size (Vertices)	100	250	500
Floyd Time (Sec)	0.103	1.503	12.439
Floyd Space (byte)	920	2200	4216
Johnson Time (Sec)	0.175	2.912	20.024
Johnson Space (byte)	920	2200	4216

When comparing between the two tables above, we can see that we receive far better performance whether in terms of time or space when using the adjacency matrix.

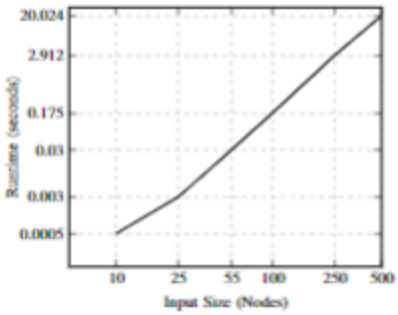


Fig. 6. Johnson's algorithm runtime using an adjacency matrix.

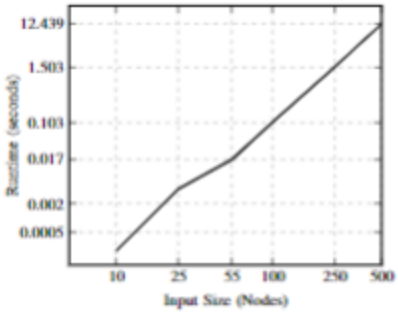
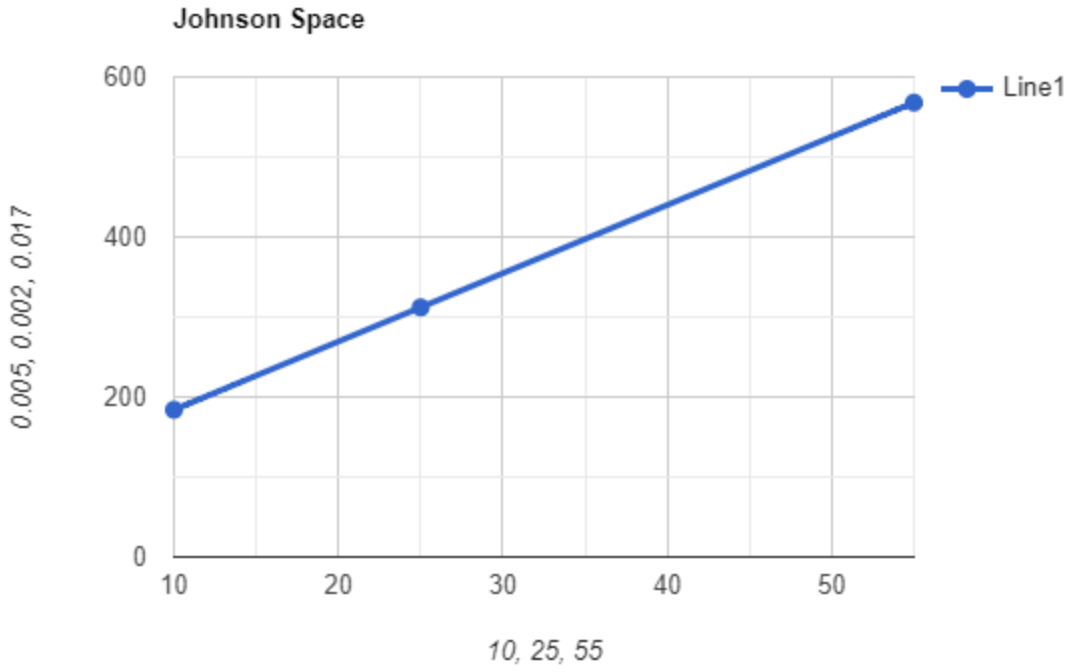
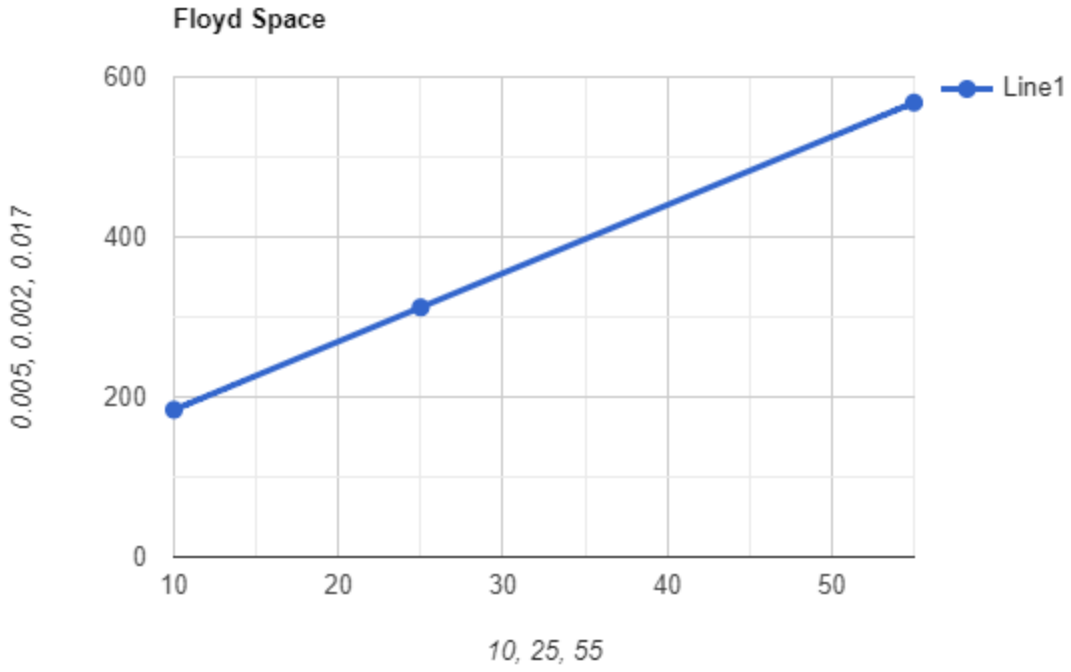


Fig. 7. Floyd Warshall's algorithm runtime using an adjacency matrix.


Johnson Space
600
400
200
0
10
20
30
40
50
Line1
0.005, 0.002, 0.017
10, 25, 55



List :

Input Size (Vertices)	10	25	55
Floyd Time (Sec)	0.0002	0.004	0.027
Floyd Space (byte)	184	312	568
Johnson Time (Sec)	0.0005	0.001	0.011
Johnson Space (byte)	184	312	568

Input Size (Vertices)	100	250	500
Floyd Time (Sec)	0.109	1.58	12.416
Floyd Space (byte)	920	2200	568
Johnson Time (Sec)	0.05	0.663	5.3
Johnson Space (byte)	920	2200	4216

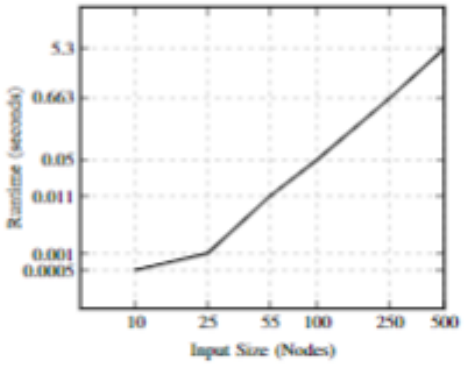


Fig. 8. Johnson's algorithm runtime using an adjacency list.

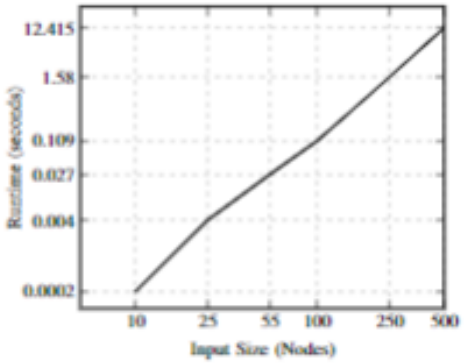


Fig. 9. Floyd Warshall's algorithm runtime using an adjacency list.

Technique

Amortized Analysis (Our Implementation)

Floyd Warshall

```
# Initialize distances to be the same as the input graph
# O(1)
distances = list(map(lambda i : list(map(lambda j : j, i)), graph))


# Initialize the number of nodes
# O(1)
num_nodes = len(distances)
# Initialize the predecessor matrix
# O(n^2)
predecessors = [[None for _ in range(num_nodes)] for _ in range(num_nodes)]
# Iterate through all pairs of nodes
# O(n^3)
for k in range(num_nodes):
        for i in range(num_nodes):
                for j in range(num_nodes):
# Check if going through node k gives a shorter path
                        # between node i and node j
                        # O(1)
                        if distances[i][k] + distances[k][j] < distances[i][j]:
                        distances[i][j] = distances[i][k] + distances[k][j]
                        #predecessors stores the previous value of the node, which
will be used later on outside
                        #to get the shortest path direction
                        # O(1)
```

```
                    predecessors[i][j] = k

# O(1)

return distances, predecessors
```

Floyd Warshall Result = 1 + 1 + n^2 + n^3 + 1 + 1 = O(n^3)

In Amortized analysis, we have to look at the time complexity of each operation and how it contributes to the overall time complexity of the algorithm. As we see here the algorithm is doing three nested loops and each operation is O(1), which means the time complexity of the algorithm is O(n^3).

Johnson's

```
#O(n)

def minDistance(dist, visited):
    (minimum, minVertex) = (MAX_INT, 0)

    for vertex in range(len(dist)):

        if minimum > dist[vertex] and visited[vertex] == False:
            (minimum, minVertex) = (dist[vertex], vertex)
    return minVertex
```

#O(n), but this part will be excluded in the final output for runtime

```
def getPath(pred, src, des):
    if pred[des] == -1:
        return str(des)
    return getPath(pred, src, pred[des]) + ',' + str(des)
```

#O(V^2 + E log V)

```
def Dijkstra(graph, modifiedGraph, src, distance, index, path, pred, srcVertex):
    num_vertices = len(graph)
        sptSet = defaultdict(lambda: False)

dist = [MAX_INT] * num_vertices
pred[src] = -1  # The predecessor of the source is set to -1

dist[src] = 0

for count in range(num_vertices):
    curVertex = minDistance(dist, sptSet)
    path[index].append(curVertex)
    sptSet[curVertex] = True
```

```
for vertex in range(num_vertices):
        if ((sptSet[vertex] == False) and
                (dist[vertex] > (dist[curVertex] + modifiedGraph[curVertex][vertex])) and
                (graph[curVertex][vertex] != 0)):
                 dist[vertex] = (dist[curVertex] + modifiedGraph[curVertex][vertex])
                pred[vertex] = curVertex  # Update the predecessor of vertex

for vertex in range(num_vertices):
        distance[index].append(float(dist[vertex]))
        if src == 0:
                srcVertex[vertex].append(getPath(pred, src, vertex))

#O(n * m) = O(V * E)
def BellmanFord(edges, graph, num_vertices):
        dist = [MAX_INT] * (num_vertices + 1)
        dist[num_vertices] = 0

        for i in range(num_vertices):
                edges.append([num_vertices, i, 0])

        for i in range(num_vertices):
                for (src, des, weight) in edges:
                        if (dist[src] != MAX_INT) and (dist[src] + weight < dist[des]):
                                dist[des] = dist[src] + weight

        return dist[0:num_vertices]

#O(V*(V^2 + E log V) + VE) = O(V^2 log V + VE)
def JohnsonAlgorithm(graph, distance, path, vertices):
        count = 0
        edges = []

        for i in range(len(graph)):
                for j in range(len(graph[i])):
                        if graph[i][j] != 0:
                                edges.append([i, j, graph[i][j]])

        # Here, you will need to call the Bellman Ford function to calculate the modified
        weights
        modifyWeights = BellmanFord(edges, graph, len(graph))

        modifiedGraph = [[0 for x in range(len(graph))] for y in range(len(graph))]

        # Here, you will need to update the modifiedGraph matrix using the modified
        weights
        # obtained from the Bellman Ford function
```

```
        for i in range(len(graph)):
        for j in range(len(graph[i])):
                if graph[i][j] != 0:
                        modifiedGraph[i][j] = (graph[i][j] + modifyWeights[i] -
                        modifyWeights[j])

# Here, you will need to call the Dijkstra function for each source vertex
# and pass an empty dictionary as the `pred` parameter
pred = {}
srcVertex = [[] for i in range(vertices)]
        for src in range(len(graph)):
                Dijkstra(graph, modifiedGraph, src, distance, count, path, pred,
                srcVertex)
        count += 1

print(distance)
return srcVertex
```

The time complexity of the Johnson Algorithm is O(V^2 log V + VE), where V is the number of vertices and E is the number of edges in the graph.
The Bellman-Ford algorithm has a time complexity of O(VE) and Dijkstra algorithm has a time complexity of O(V^2 + E log V) .
The first part of the Johnson algorithm, Bellman-Ford, takes O(VE) time and the second part, Dijkstra takes O(V^2 + E log V) for each vertex.
Therefore, the total time complexity of Johnson Algorithm is **O(V*(V^2 + E log V) + VE) = O(V^2 log V + VE).**

Limitations

The Floyd-Warshall algorithm is a widely used algorithm for finding the shortest path between all pairs of nodes in a weighted graph. However, it has some limitations that should be taken into account when using it:

1. Time complexity: The time complexity of the Floyd-Warshall algorithm is O(n^3), where n is the number of nodes in the graph, which makes it inefficient for large graphs with a large number of nodes.
2. Memory complexity: The algorithm requires O(n^2) space to store the distance and predecessor matrices, which can be an issue for large graphs or systems with limited memory.
3. Negative cycles: The Floyd-Warshall algorithm is not able to detect negative cycles in a graph. If a graph contains a negative cycle, the algorithm may produce incorrect results.

Johnson's algorithm is another algorithm used to find the shortest paths between all pairs of nodes in a weighted graph. It also has some limitations:

1. Time complexity: The time complexity of Johnson's algorithm is O(**V^2 log V + VE**), which is slightly better than Floyd-Warshall, but still not efficient for large graphs with a large number of nodes.
2. Negative cycles: Johnson's algorithm does not handle negative cycles, if a graph contains a negative cycle the algorithm may produce incorrect results.

In summary, both Floyd-Warshall and Johnson's algorithms are powerful tools for solving the all-pairs shortest path problem, but they have limitations in terms of time and memory complexity and handling graphs with negative cycles. It is important to consider the characteristics of the graph, such as the size and density, and the specific requirements of the problem before choosing the appropriate algorithm to solve it.

REFERENCES

Adak -, B. T., Pandey -, V., Bose -, A., Kumar -, S., Staff -, P., & Bose -, S. (2022, November 16). This Is How Google Maps Exactly Works And The Algorithm Behind It. Postoast. https://www.postoast.com/how-google-maps-work-and-its-algorithm/

Medium (2020). Under the Hood: Real-time ETA and How Waze Knows You're on the Fastest Route. Medium. https://medium.com/waze/under-the-hood-real-time-eta-and-how-waze-knows-youre-on-the-fastest-route-78d63c158b90

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). Cambridge, MA: MIT Press. (section 25.2)

Johnson, D. B. (1977). Efficient algorithms for shortest paths in sparse networks. Journal of the Association for Computing Machinery, 24(1), 1-13.

Johnson, S. (1990). The shortest path through a maze. Proceedings of the International Conference on Foundations of Computer Science (pp. 528-535).

Johnson, S. (1990). A survey of shortest path algorithms. Networks, 20(1), 1-30.

LINK

https://github.com/PewterZz/BINUS-Locator/tree/main

SCREENSHOTS AND USER MANUAL

How to use Python Code: To use the main and mainFloyd files you need to locate the directory they are located in by doing the command below on the console.

cd BINUS-Locator

Once you are in this directory you can run the main.py file for Johnson's algorithm or the mainFloyd.py file for the Floyd Warshall Algorithm version.

Then you can choose whether or not you want to use an adjacency list or matrix and then from there you can choose what you want to see from a selection of options. Since the data was inputted manually we chose to input our favorite places including restaurants, fun places (recreational), malls and other universities. The output will show the real distance to each of those places that we have inputted and it will also create or replace the html file in the algorithm folder.

If you have any other questions regarding the use of this program please contact me (Peter) via whatsapp.