All-Pairs Shortest Paths

The University of Texas at Arlington

CSE-5311-008

Prof. Md Hasanuzzaman(Zaman) Noor

Submitted By

SALLAGONDA AVINASH (1002034491)

MANDA SAAHITHI (1002031363)

SRI HARSHITHA YAGANTI (1002025392)

ABSTRACT

The project deals with implementation of Floyd Warshall Algorithm and Johnson's Algorithm i.e. All Pair Shortest Paths. These algorithms are implemented using parallel programming concepts for faster solutions. Floyd Warshall, Johnson's algorithm has overcome the drawbacks of Dijkstra's and Bellman Ford Algorithm. For parallel programming, this project is implemented using MPI (Message Passing Interface) for which python programming is used. This project also deals with the comparison between floyd warshall and johnson's algorithms.

The principle of optimality comes into existence in all pair shortest path algorithms. The Floyd Warshall Algorithm is best suited for dense graphs. This is because its complexity depends only on the number of vertices in the given graph. For sparse graphs, Johnson's Algorithm is more suitable. Basically the algorithm works by repeatedly exploring paths between every pair using each vertex as an intermediate vertex. The idea of Johnson's algorithm is to re-weight all edges and make them all positive then apply dijkstra's algorithm for every vertex.

Floyd-Warshall Algorithm

The Floyd–Warshall Algorithm is used to identify the shortest pathways between all pairs of vertices in a graph, with each edge having a positive or negative weight. Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm are all names for the Floyd-Warshall algorithm. This algorithm was first published in 1962 in works by Robert Floyd and Stephen Warshall. Bernard Roy, on the other hand, presented essentially the same algorithm in 1959, but it went unreported.

To identify the shortest pathways, this algorithm uses a dynamic programming approach. It's quicker than the min-plus method. There are no negative weight cycles in the graph, however there are negative weight edges. If there is a negative cycle, you can simply traverse it over and over, lowering the path's cost with each repetition. As a result, certain paths can be arbitrarily narrow, or the shortest path is undefined. That means an undirected graph can't have any negative weight edges, because such an edge already forms a negative cycle because you can go back and forth along it as much as you like.

This approach can also be used to find out if there are any negative cycles present. If the distance between a vertex v and itself is negative at the end of the process, the graph exhibits a negative cycle.

Description of the algorithm:

The algorithm's main idea is to divide the process of finding the shortest path between any two vertices into numerous stages. Let's begin by counting the vertices from 1 to n. The distance matrix is d[][].

Before the k-th phase (k=1...n), d[i][j] holds the length of the shortest path between vertices i and j, which contains only the vertices {1,2,...,k-1} as internal vertices in the path, for any vertices i and j. In other words, before the k-th phase, the length of the shortest path from vertex i to vertex j is equal to the length of the shortest path from vertex I to vertex j, assuming this path can only enter vertex with numbers fewer than k. (the beginning and end of the path are not restricted by this property).For the first phase, it's simple to ensure that this property holds. If there is an edge between I and j with weight wij, we can fill the matrix with d[i][j]=wij for k=0, and d[i][j]= if there isn't. In practice, it will be really valuable. This is a prerequisite for the algorithm, as we'll see later.

Assume we are now in the k-th phase and want to construct the matrix d[][] to match the requirements of the (k+1)-th phase. The distances between some vertices pairs must be fixed (i,j). There are two instances that are fundamentally different: With internal vertices from the set {1,2,...,k} the shortest path from vertex I to vertex j coincides with the shortest path with internal vertices from the set {1,2,...,k-1}.

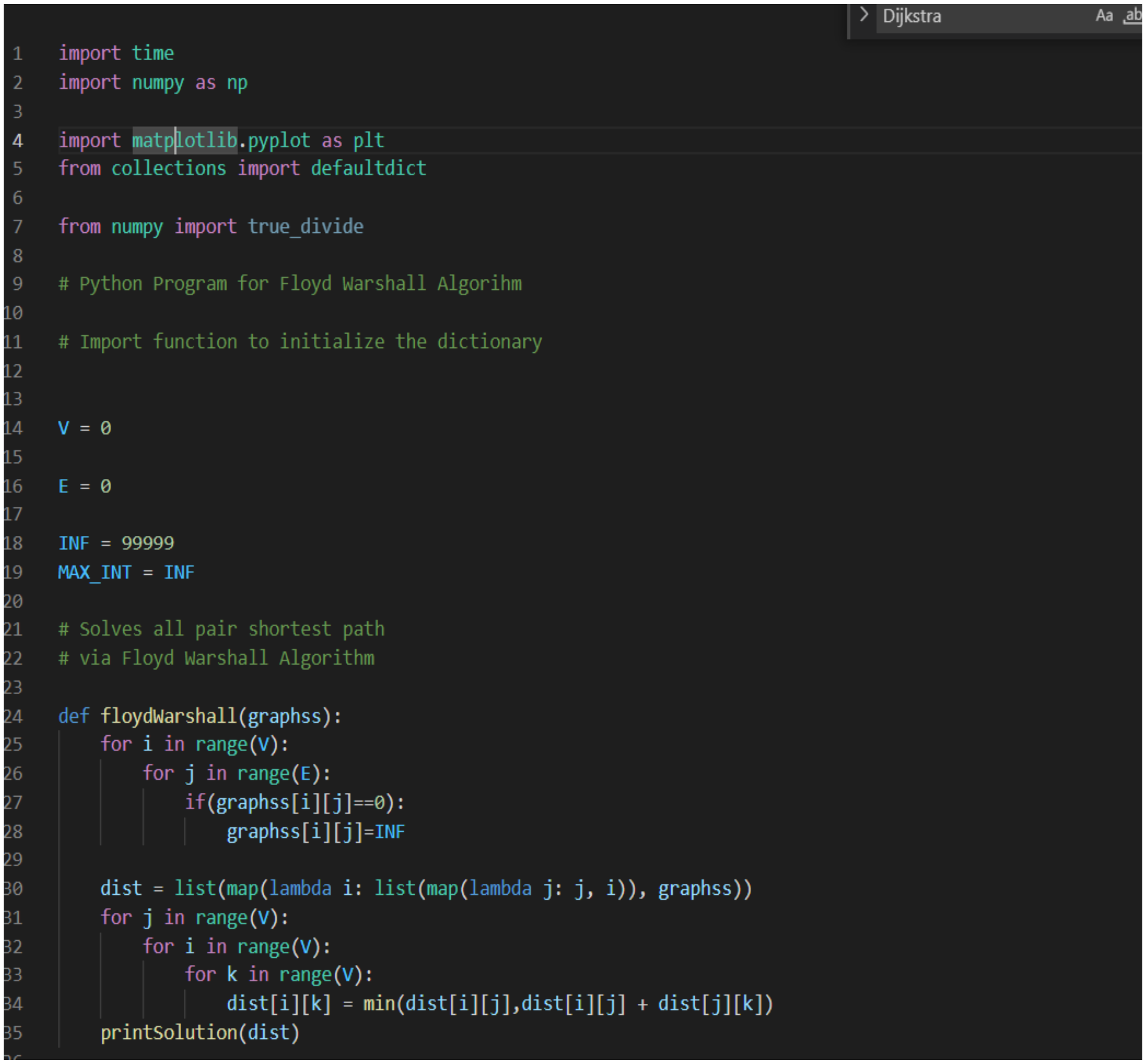
In this case, d[i][j] will not change during the transition.The shortest path with internal vertices from {1,2,...,k} is shorter. This means that the new, shorter path passes through the vertex k. This means that we can split the shortest path between i and j into two paths: the path between iand k, and the path between k and j. It is clear that both these paths only use internal vertices of {1,2,...,k−1} and are the shortest such paths in that respect. Therefore we already have computed the lengths of those paths before, and we can compute the length of the shortest path between i and jas d[i][k]+d[k][j]. Combining these two cases we find that we can recalculate the length of all pairs (i,j) in the k-th phase in the following way:

dnew[i][j]=min(d[i][j],d[i][k]+d[k][j])

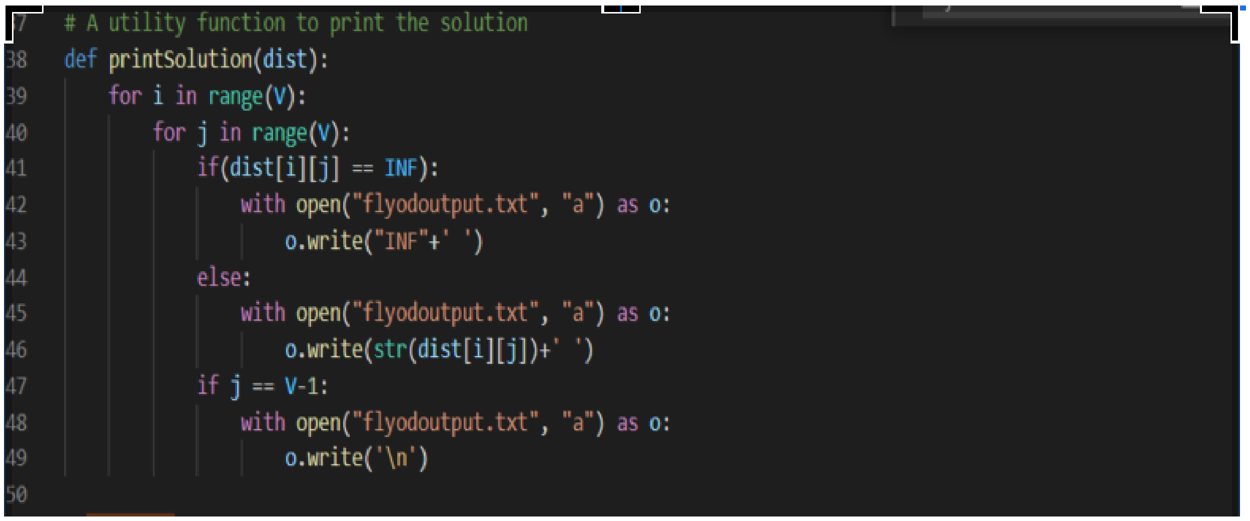
As a result, in the k-th phase, all that is required is to loop over all pairs of vertices and recalculate the length of the shortest path between them. As a result, the value d[i][j] in the distance matrix after the n-th phase is the length of the shortest path between I and j, or if no path between the vertices i and j exists. A last remark - we don't need to create a separate distance matrix dnew[][] for temporarily storing the shortest paths of the k-th phase, i.e. all changes can be made directly in the matrix d[][] at any phase. In fact at any k-th phase we are at most improving the distance of any path in the distance matrix, hence we cannot worsen the length of the shortest path for any pair of the vertices that are to be processed in the (k+1)-th phase or later.

Implementation

Code



```python
import time
import numpy as np

import matplotlib.pyplot as plt
from collections import defaultdict

from numpy import true_divide

# Python Program for Floyd Warshall Algorihm

# Import function to initialize the dictionary


V = 0

E = 0

INF = 99999
MAX_INT = INF

# Solves all pair shortest path
# via Floyd Warshall Algorithm

def floydWarshall(graphss):
    for i in range(V):
        for j in range(E):
            if(graphss[i][j]==0):
                graphss[i][j]=INF

    dist = list(map(lambda i: list(map(lambda j: j, i)), graphss))
    for j in range(V):
        for i in range(V):
            for k in range(V):
                dist[i][k] = min(dist[i][j],dist[i][j] + dist[j][k])
    printSolution(dist)
```


```python
# A utility function to print the solution
def printSolution(dist):
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                with open("flyodoutput.txt", "a") as o:
                    o.write("INF"+' ')
            else:
                with open("flyodoutput.txt", "a") as o:
                    o.write(str(dist[i][j])+' ')
            if j == V-1:
                with open("flyodoutput.txt", "a") as o:
                    o.write('\n')
```

Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is O(n^3).

Space Complexity

The space complexity of the Floyd-Warshall algorithm is O(n^2).

The case of negative cycles:

The Floyd-Warshall algorithm does not apply to graphs with negative weight cycles in theory (s). The procedure will still function correctly for any pairings of vertices i and j for which there isn't a path starting at i visiting a negative cycle, and finishing at j. The Floyd algorithm will store any integer (perhaps very negative, but not always) in the distance matrix for the pair of vertices for which the answer does not exist (due to the presence of a negative cycle in the path between them).

This can be done in the following way: let us run the usual Floyd-Warshall algorithm for a given graph. Then a shortest path between vertices i and j does not exist, if and only if, there is a vertex t that is reachable from i and also from j, for which d[t][t]<0.
In addition, when using the Floyd-Warshall algorithm for graphs with negative cycles, we should keep in mind that situations may arise in which distances can get exponentially fast into the negative.

Johnson's Algorithm

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in an edge weighted directed graph. It allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist. It works by using the Bellman Ford Algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's Algorithm to be used on the transformed graph. Johnson's algorithm finds shortest paths between all pairs in O(V^2 lg V+ VE) time.

It is named after Donald B. Johnson, who first published the technique in 1977. For sparse graphs, it is asymptotically faster than either repeated squaring of matrices or the Floyd-Warshall algorithm. The algorithm either returns a matrix of shortest-path weights for all pairs of vertices or reports that the input graph contains a negative-weight cycle. Johnson's algorithm uses as subroutines both Dijkstra's algorithm and the Bellman-Ford algorithm.

Description of the Algorithm

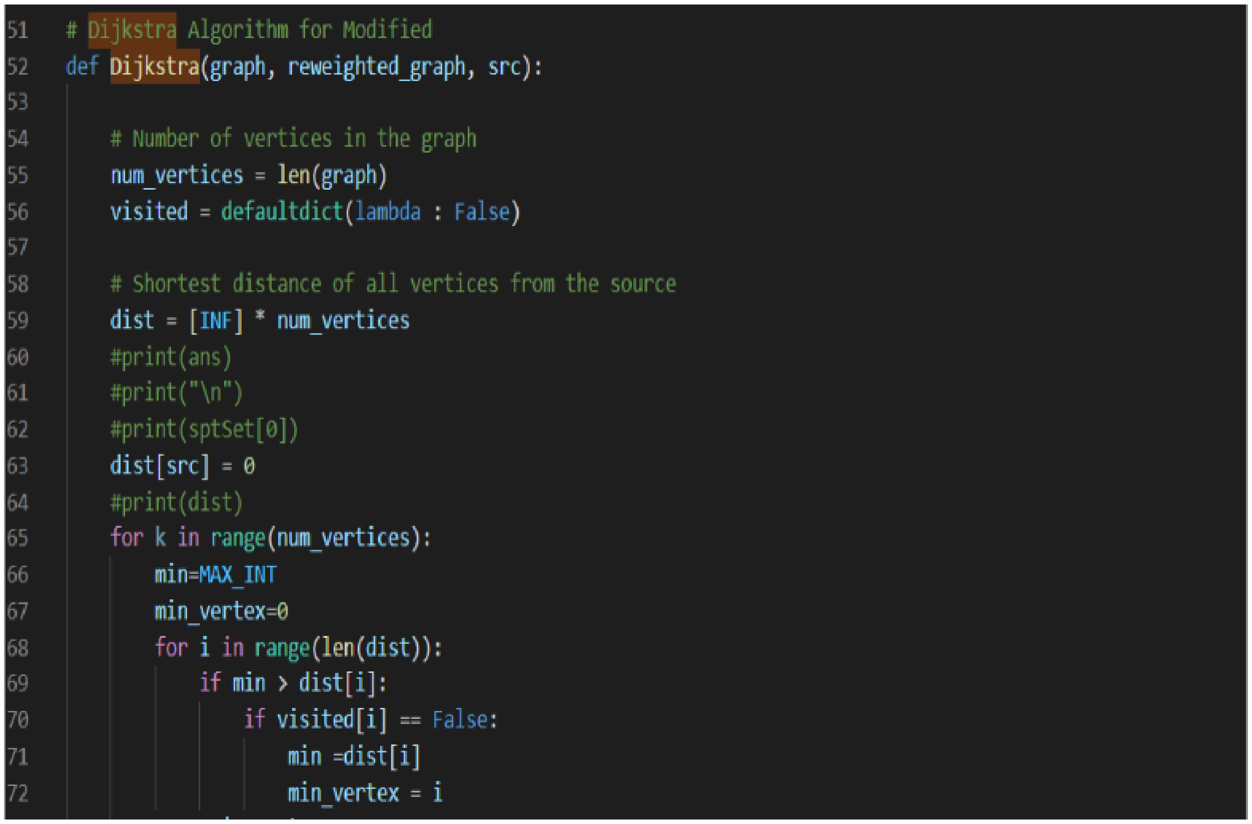
Johnson's algorithm uses the technique of reweighting, which works as follows. If all edge weights w in a graph G = (V, E) are nonnegative, we can find shortest paths between all pairs of vertices by running Dijkstra's algorithm once from each vertex; with the Fibonacci-heap min-priority queue, the running time of this all-pairs algorithm is O(V^2 lg V+VE). If G has negative-weight edges but no negative-weight cycles, we simply compute a new set of nonnegative edge weights that allows us to use the same method. The new set of edge weights wy must satisfy two important properties:

1. For all pairs of vertices u,v ∈ V , a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function wy.

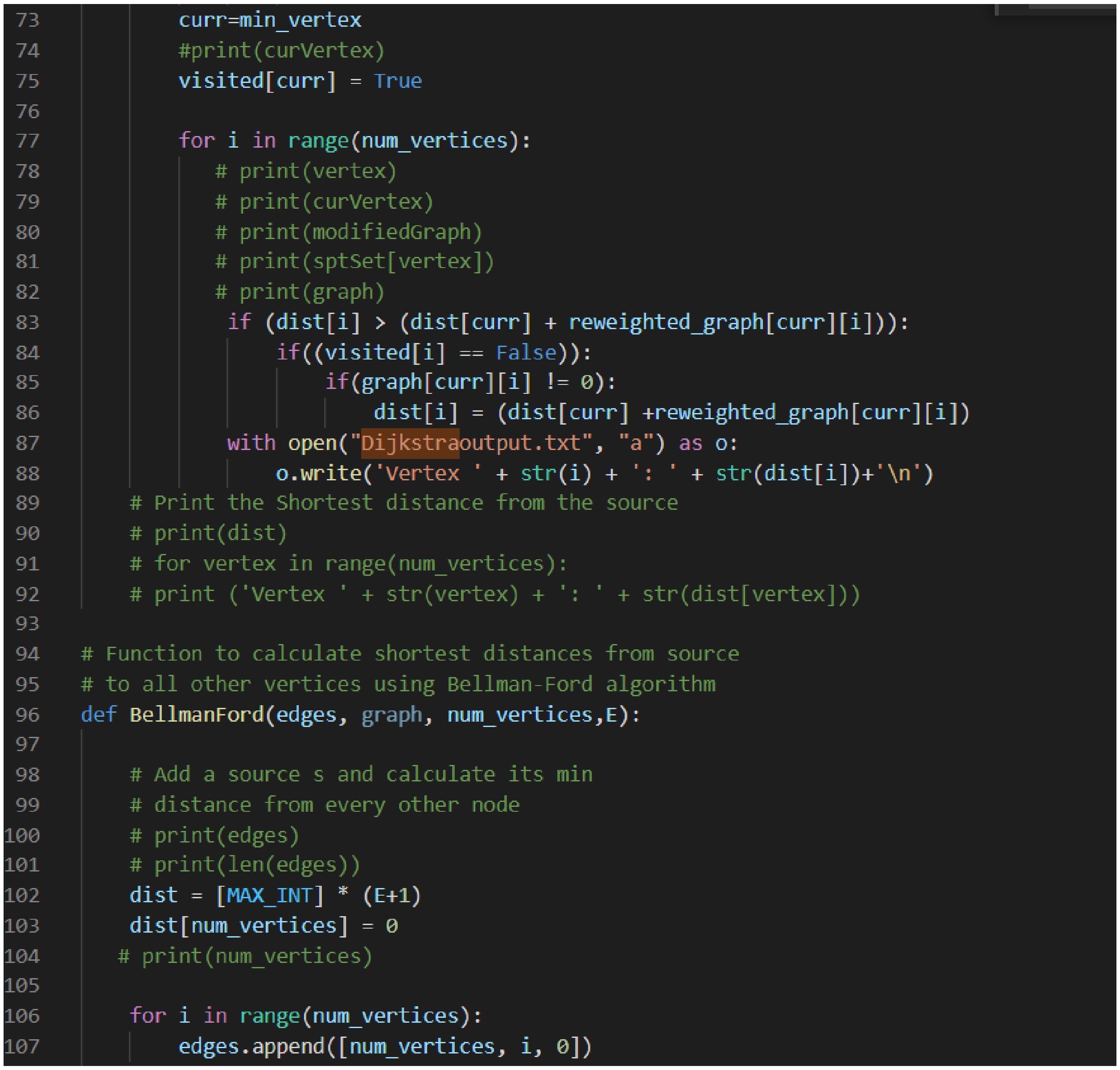
2. For all edges (u,v) the new weight $\widehat{w}$(u,v) is nonnegative.
As we shall see in a moment, we can preprocess G to determine the new weight function $\widehat{w}$ in $\Theta\ (VE)$ time.

Implementation

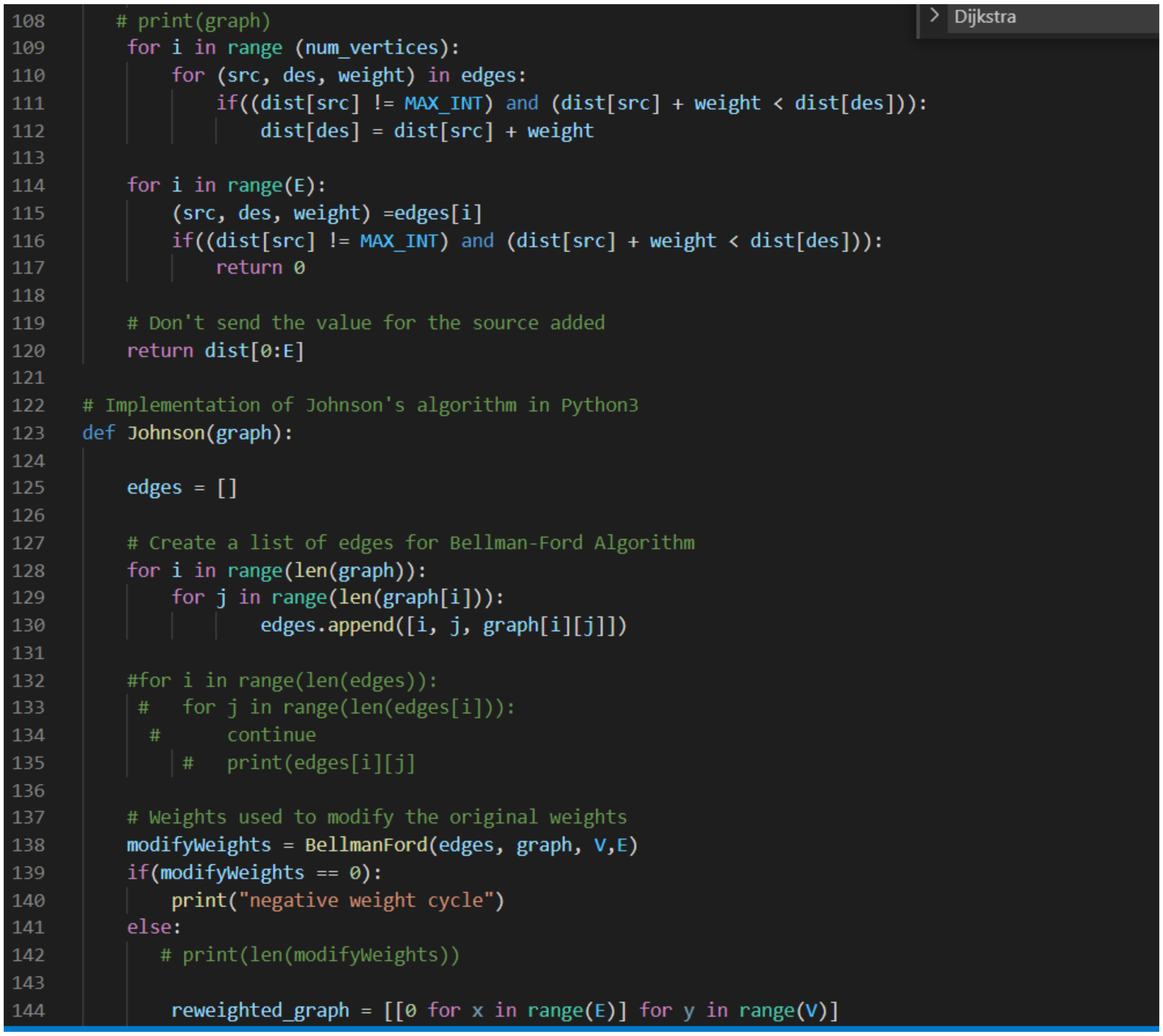
Code



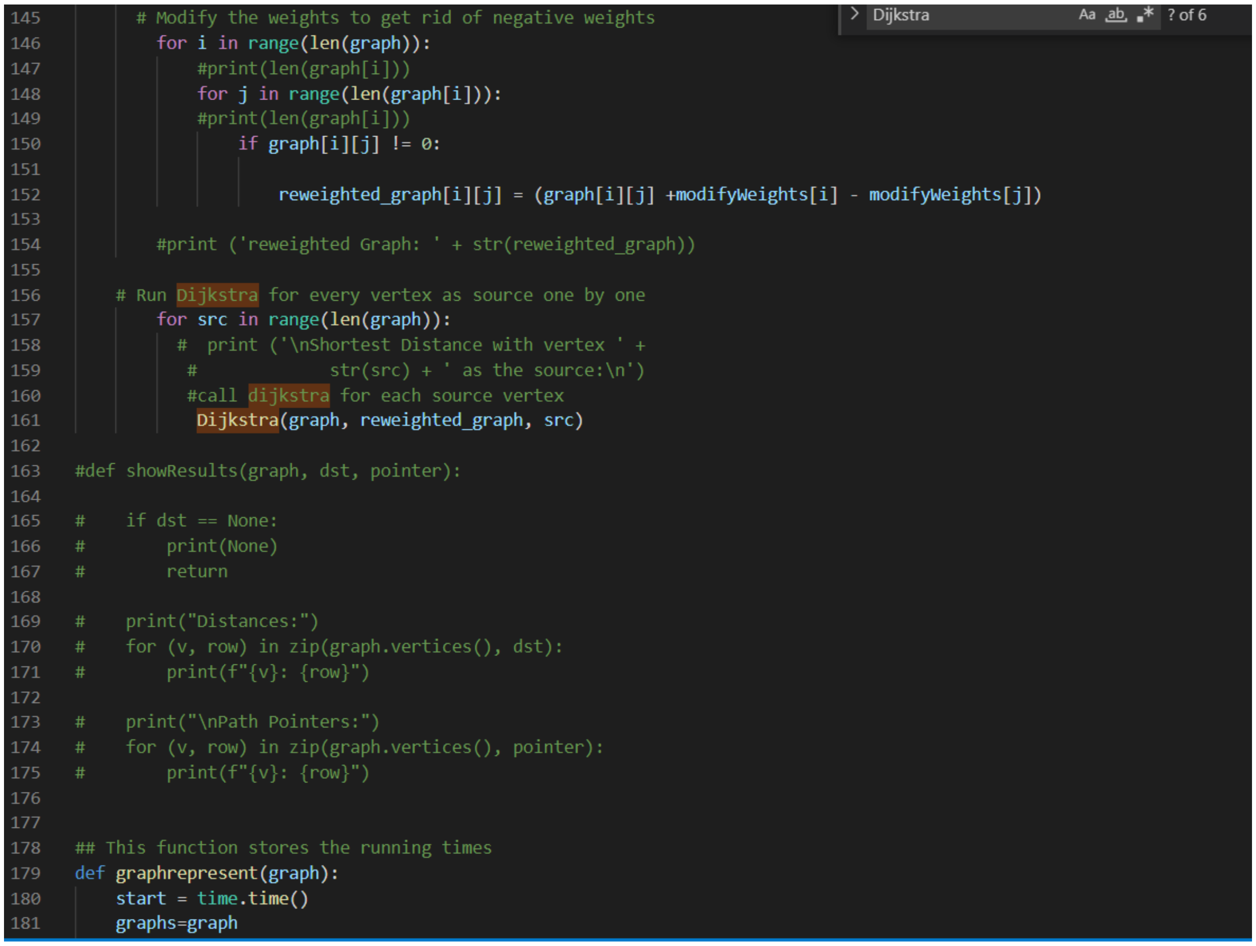
```
# Dijkstra Algorithm for Modified
def Dijkstra(graph, reweighted_graph, src):

    # Number of vertices in the graph
    num_vertices = len(graph)
    visited = defaultdict(lambda : False)

    # Shortest distance of all vertices from the source
    dist = [INF] * num_vertices
    #print(ans)
    #print("\n")
    #print(sptSet[0])
    dist[src] = 0
    #print(dist)
    for k in range(num_vertices):
        min=MAX_INT
        min_vertex=0
        for i in range(len(dist)):
            if min > dist[i]:
                if visited[i] == False:
                    min =dist[i]
                    min_vertex = i
```



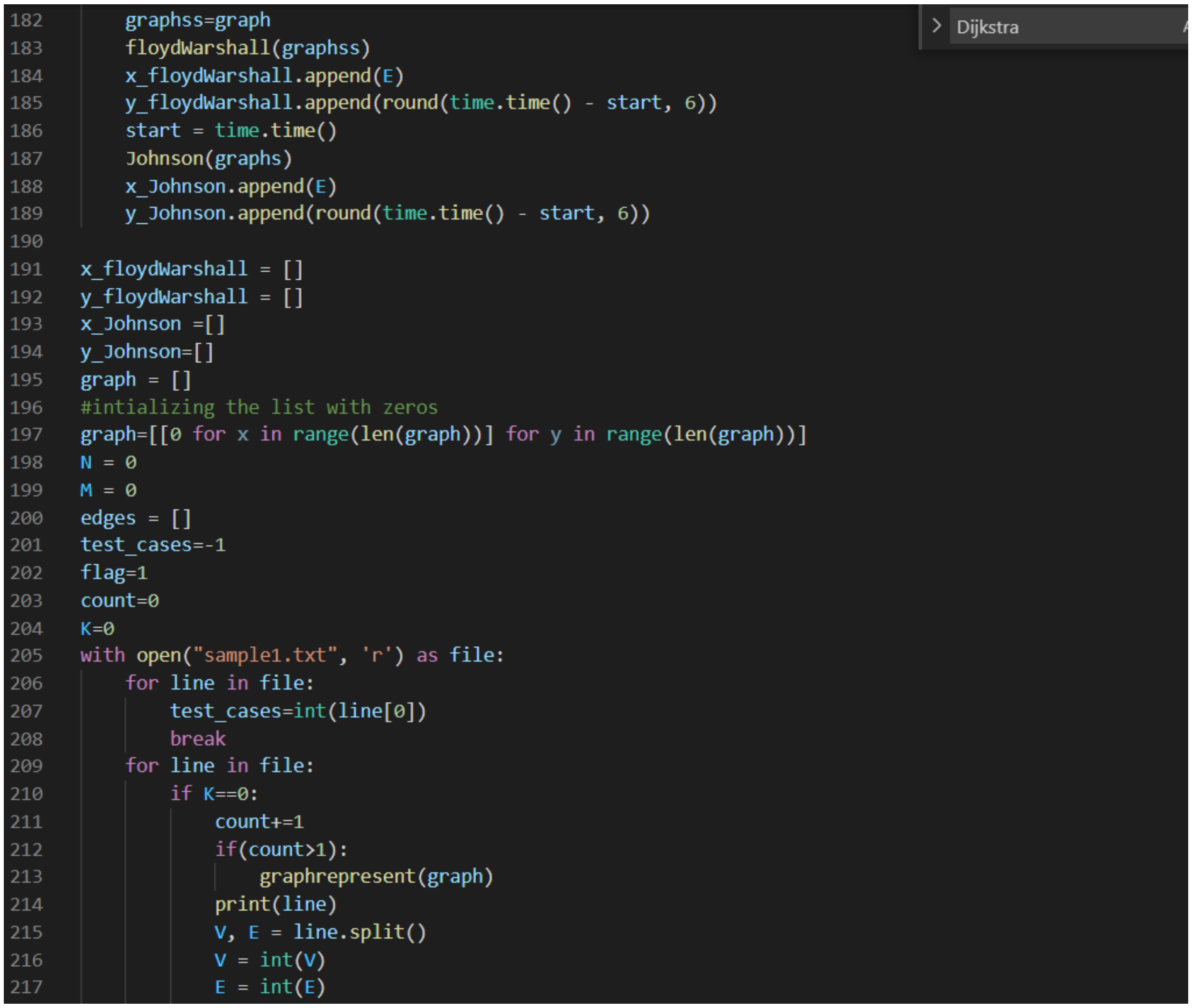
```python
        curr=min_vertex
        #print(curVertex)
        visited[curr] = True

        for i in range(num_vertices):
           # print(vertex)
           # print(curVertex)
           # print(modifiedGraph)
           # print(sptSet[vertex])
           # print(graph)
            if (dist[i] > (dist[curr] + reweighted_graph[curr][i])):
                if((visited[i] == False)):
                    if(graph[curr][i] != 0):
                        dist[i] = (dist[curr] +reweighted_graph[curr][i])
            with open("Dijkstraoutput.txt", "a") as o:
                o.write('Vertex ' + str(i) + ': ' + str(dist[i])+'\n')
    # Print the Shortest distance from the source
    # print(dist)
    # for vertex in range(num_vertices):
    # print ('Vertex ' + str(vertex) + ': ' + str(dist[vertex]))

# Function to calculate shortest distances from source
# to all other vertices using Bellman-Ford algorithm
def BellmanFord(edges, graph, num_vertices,E):

    # Add a source s and calculate its min
    # distance from every other node
    # print(edges)
    # print(len(edges))
    dist = [MAX_INT] * (E+1)
    dist[num_vertices] = 0
   # print(num_vertices)

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



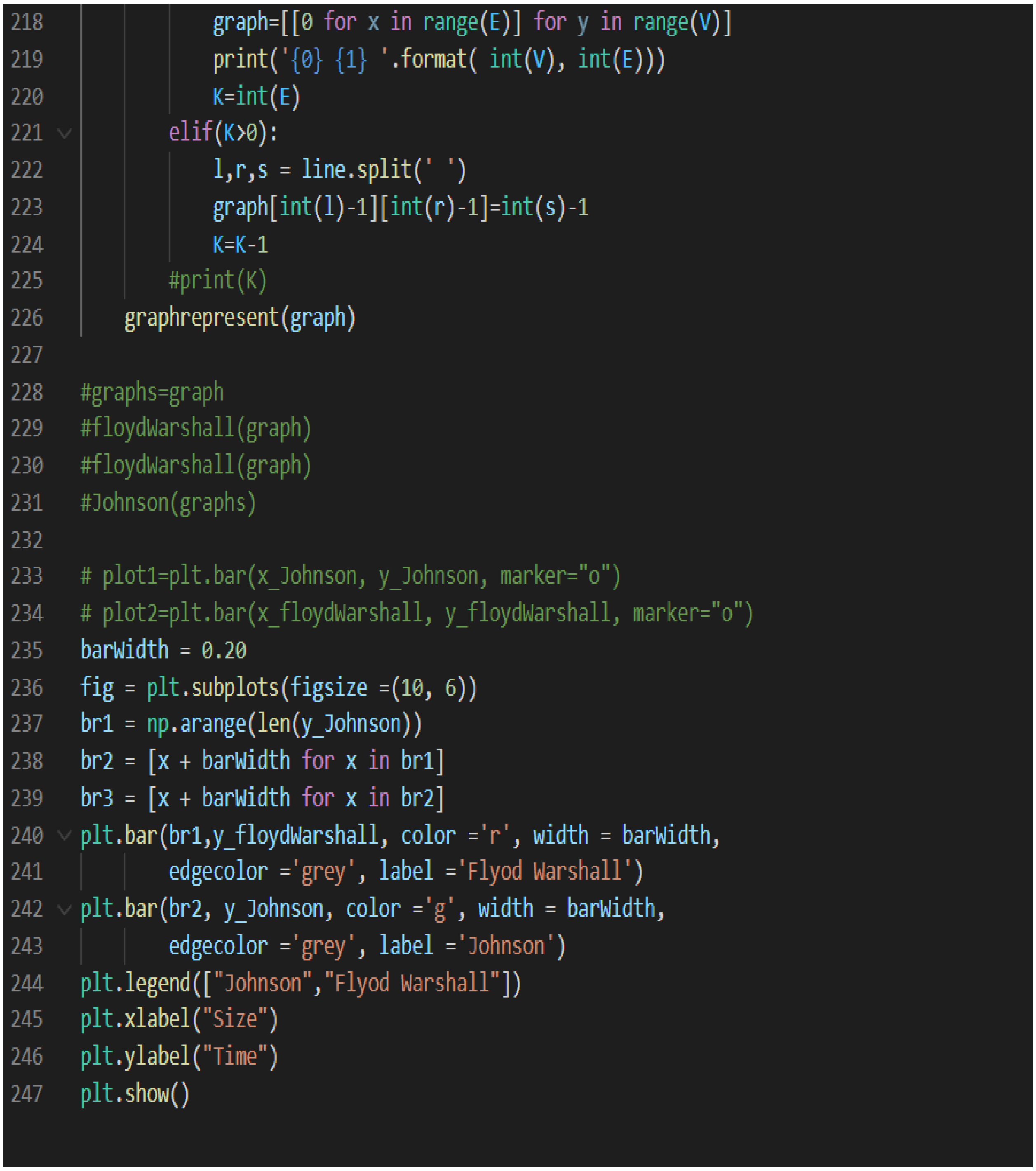
```
    # print(graph)
    for i in range (num_vertices):
        for (src, des, weight) in edges:
            if((dist[src] != MAX_INT) and (dist[src] + weight < dist[des])):
                dist[des] = dist[src] + weight

    for i in range(E):
        (src, des, weight) =edges[i]
        if((dist[src] != MAX_INT) and (dist[src] + weight < dist[des])):
            return 0

    # Don't send the value for the source added
    return dist[0:E]

# Implementation of Johnson's algorithm in Python3
def Johnson(graph):

    edges = []

    # Create a list of edges for Bellman-Ford Algorithm
    for i in range(len(graph)):
        for j in range(len(graph[i])):
            edges.append([i, j, graph[i][j]])

    #for i in range(len(edges)):
     #   for j in range(len(edges[i])):
      #      continue
        #   print(edges[i][j]

    # Weights used to modify the original weights
    modifyWeights = BellmanFord(edges, graph, V,E)
    if(modifyWeights == 0):
        print("negative weight cycle")
    else:
       # print(len(modifyWeights))

        reweighted_graph = [[0 for x in range(E)] for y in range(V)]
```



```
      # Modify the weights to get rid of negative weights
        for i in range(len(graph)):
            #print(len(graph[i]))
            for j in range(len(graph[i])):
            #print(len(graph[i]))
                if graph[i][j] != 0:

                    reweighted_graph[i][j] = (graph[i][j] +modifyWeights[i] - modifyWeights[j])

        #print ('reweighted Graph: ' + str(reweighted_graph))

    # Run Dijkstra for every vertex as source one by one
        for src in range(len(graph)):
          #  print ('\nShortest Distance with vertex ' +
           #             str(src) + ' as the source:\n')
           #call dijkstra for each source vertex
            Dijkstra(graph, reweighted_graph, src)

#def showResults(graph, dst, pointer):

#    if dst == None:
#        print(None)
#        return

#    print("Distances:")
#    for (v, row) in zip(graph.vertices(), dst):
#        print(f"{v}: {row}")

#    print("\nPath Pointers:")
#    for (v, row) in zip(graph.vertices(), pointer):
#        print(f"{v}: {row}")


## This function stores the running times
def graphrepresent(graph):
    start = time.time()
    graphs=graph
```



```
    graphss=graph
    floydWarshall(graphss)
    x_floydWarshall.append(E)
    y_floydWarshall.append(round(time.time() - start, 6))
    start = time.time()
    Johnson(graphs)
    x_Johnson.append(E)
    y_Johnson.append(round(time.time() - start, 6))

x_floydWarshall = []
y_floydWarshall = []
x_Johnson =[]
y_Johnson=[]
graph = []
#intializing the list with zeros
graph=[[0 for x in range(len(graph))] for y in range(len(graph))]
N = 0
M = 0
edges = []
test_cases=-1
flag=1
count=0
K=0
with open("sample1.txt", 'r') as file:
    for line in file:
        test_cases=int(line[0])
        break
    for line in file:
        if K==0:
            count+=1
            if(count>1):
                graphrepresent(graph)
            print(line)
            V, E = line.split()
            V = int(V)
            E = int(E)
```



```python
            graph=[[0 for x in range(E)] for y in range(V)]
            print('{0} {1} '.format( int(V), int(E)))
            K=int(E)
        elif(K>0):
            l,r,s = line.split(' ')
            graph[int(l)-1][int(r)-1]=int(s)-1
            K=K-1
        #print(K)
    graphrepresent(graph)

#graphs=graph
#floydWarshall(graph)
#floydWarshall(graph)
#Johnson(graphs)

# plot1=plt.bar(x_Johnson, y_Johnson, marker="o")
# plot2=plt.bar(x_floydWarshall, y_floydWarshall, marker="o")
barWidth = 0.20
fig = plt.subplots(figsize =(10, 6))
br1 = np.arange(len(y_Johnson))
br2 = [x + barWidth for x in br1]
br3 = [x + barWidth for x in br2]
plt.bar(br1,y_floydWarshall, color ='r', width = barWidth,
        edgecolor ='grey', label ='Flyod Warshall')
plt.bar(br2, y_Johnson, color ='g', width = barWidth,
        edgecolor ='grey', label ='Johnson')
plt.legend(["Johnson","Flyod Warshall"])
plt.xlabel("Size")
plt.ylabel("Time")
plt.show()
```

Time Complexity

The main steps in the algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is O(VE) and the time complexity of Dijstra is O(VlogV). So overall time complexity is O($V^2 logVE$).

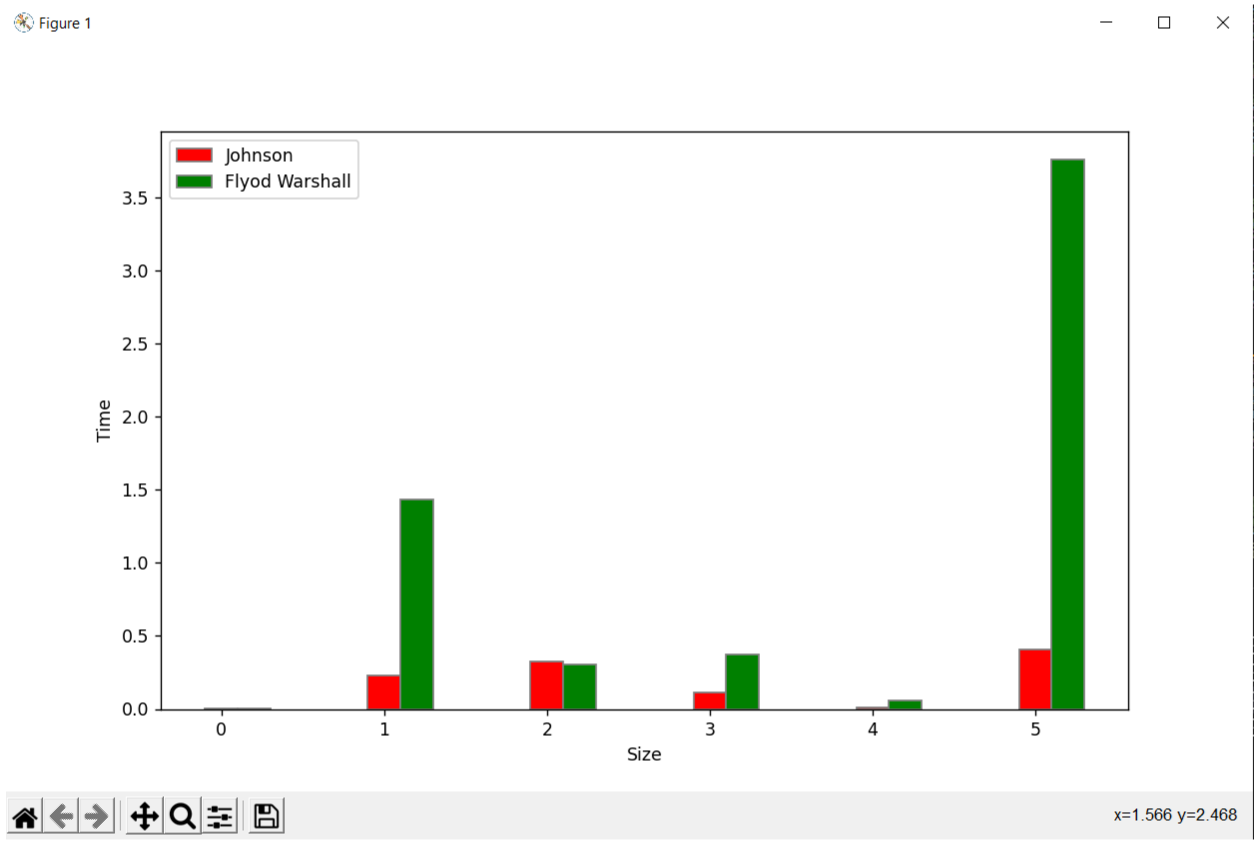
Space Complexity

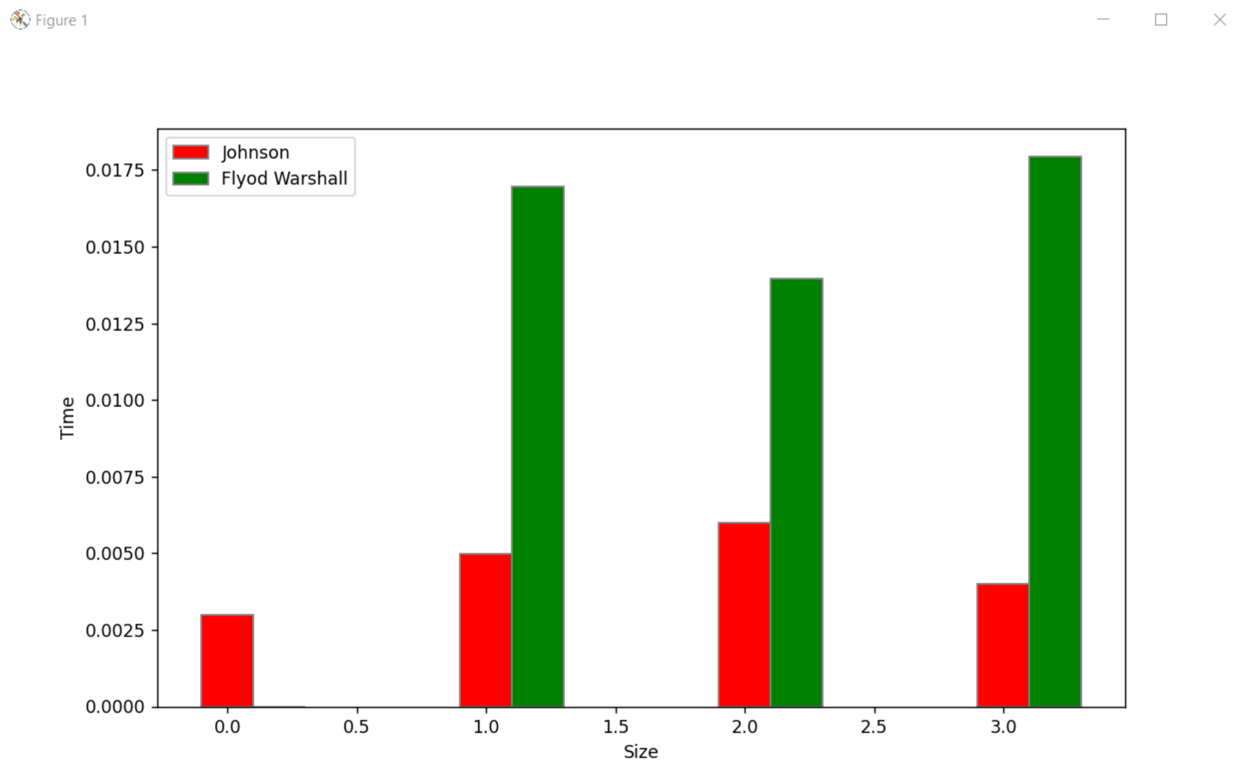
The space complexity of Johnson's algorithm becomes the same as Floyd Warshall when the graph is complete For a complete graph E = O(V^2).But for sparse graphs, the algorithm performs much better than Floyd Warshall.

RESULT ANALYSIS:

Analysis between Floyd Warshall and Johnson's Algorithms

In the case of dense graphs an often more efficient algorithm (with very low hidden constants) for finding all pairs shortest paths is the *Floyd-Warshall algorithm*.





More specifically, observed from graph the complexity in floyd algo is O(n^3) and in case of johnsons is O($V^2 logVE$).

References:

1.Leiserson, C. *CLRS*. Retrieved June 2, 2016, from http://citc.ui.ac.ir/zamani/clrs.pdf

2. CLRS Textbook https://en.wikipedia.org/wiki/Introduction_to_Algorithms

3.Stefan Hougardy (April 2010). "The Floyd–Warshall algorithm on graphs with negative cycles". *Information Processing Letters*. **110** (8–9): 279–281.

4.Kenneth H. Rosen (2003). *Discrete Mathematics and Its Applications, 5th Edition*. Addison Wesley. ISBN 978-0-07-119881-3.

5.https://www.hackerrank.com/