REPORT TITLE

**Floyd\_Warshall Algorithm**

By

Shorouq Aiman Telfah

Submitted to

The University of Liverpool

MASTER-OF-Data Science and Artificial intelligence

*Module-Name*

*CSCK541 Software Development in Practice January 2024 A*

Word Count: 1500

26/02/2024

TABLE OF CONTENTS

**Page**

[List of Figure 1](#list" \o "list)

[Chapter 1. Introduction 2](#INTRODUCTION)

[Chapter 2. Code implementation 2](#CODE_IMP)

[2.1. Floyd-Warshall Function Explanation 2](#FLOYD)

[**Chapter 3. Creating test and coverage**](#TEST)

**3.1unit test with coverage 5**

**3.2 Performance Test 6**

**3.3Hypothesis to explain the performance test result 7**

Chapter 4[. Conclusions](#CONCLUSION) 8

[References 9](#REFERENCE)

[**Appendix.A\_recursive.py 10**](#appa)

[**Appendix.B\_recursive +unittest with coverage.py 11**](#appb)

[**Appendix.C\_performance-test-recursive.py 13**](#appc)

[**Appendix.D\_iterative.py 14**](#appd)

[**Appendix.E** **\_iter unittest with cav.py 15**](#appe)

[**Appendix.F\_performance-test-iterative.py 17**](#appf)

**List of Figures**

**Page**

[**Figure 1. Import ‘sys’**](#fig1) **3**

[**Figure 2. Input graph**](#fig2) **3**

[**Figure 3. floyd\_algorithm**](#fig3)

[**Figure 4. Code running**](#fig4) **4**

[**Figure 5. The unittest with coverage for recursive code**](#fig5) **5**

[**Figure 6. Importing timeit**](#fig6) **6**

[**Figure 7. Performance test**](#fig7) **7**

# INTRODUCTION

In 1962 Floyd\_Warshall algorithm was proposed by Robert Floyd and Stephen Warshall independently, it is a dynamic programming algorithm that is used to find the shortest paths between every pair of vertices (nodes) in the graph as it was designed in a weighted graph, The Floyd\_Warshall algorithm iteratively updates the data which represent the shortest path between all pairs of vertices, in which each vertex considered as potential intermediate vertex in the paths between pairs of vertices and calculate if the path through this vertex is shorter. This could be applied in wide real-life applications such as in networking devices(Aini. A.,2012) and (Floyd Warshall Algorithm | DP-16, 2023)

This report is to meet the requirements of the mid-module assignment, so in this essay, the recursive version of Floyd\_Warshall code will be proposed based on the given imperative version, then a unit test for each function will be created in addition to a performance test after that the performance for both iterative version and recursive version will be compared and a hypothesis to explain the result will be suggested. The work will be under source control (https://github.com/shorouq-bot/Mid\_Module\_Assignment.git). To improve readability, simplicity and clarity, the code will be written to Python Enhancement Proposal-8 standards (PEP-8). All codes were added as appendices.

# code implementation

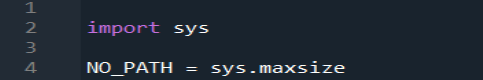
Floyd-Warshall Function Explanation

Here is an explanation of how the application was built:

2.1.a Importing the necessary module:

The ‘sys’ module in Python was used to access the maximum representable integer (infinite value), which indicates that there is no path between the two nodes. Then the maximum integer was assigned to the NO\_PATH variable.

**Figure (1) \_Import ‘sys’:**

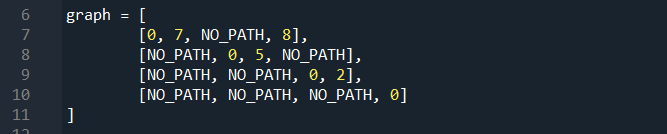


[**See Appendix A**](#appa)**,** [**Appendix D**](#appd)

**2.1.b Initialize the input graph:**

The first step is to initialize the solution matrix as an input graph, which will represent the initial matrix in the form of a list of the lists, each inner list represents an arrow, and the graph variable illustrates the weight of the edge from vertex ‘I’ to vertex ‘j’ based on the given data on the assignment (Floyd Warshall Algorithm | DP-16, 2023).

**Figure (2)-** **Input graph:**



**2.1.c Defining the ‘Floyd-algorithm’ function with nested ‘recursive\_floyd\_algorithm’ function**

The ‘floyd\_algorithm’ function was built to take the input from the ‘graph’ and return it as the shortest path between all pairs of vertices in a two-dimensional graph.

The function determines the number of vertices based on the number of rows in the graph. The length of the graph will be executed and then assigned to the ‘n’ variable.

Then the nested function ‘recursive\_floyd\_algorithm’ was built within the ‘floyed\_algorithm’ function to encapsulate the recursive function.

The ‘recursive\_floyd\_algorithm’ function will calculate the shortest paths between individual pairs of vertices up to index k by using the recursive approach in which three arguments will be passed: ‘i’ as the starting point of the path ‘j’ is the destination, both represent the weight of each edge and ‘k’ will represent the potential intermediate vertex in the shortest path between ‘i’-‘j’.

The termination case (base case) is determined when k is zero, which means that there are no intermediate vertices between ‘i’ and ‘j’, so the function will return the weight edge between ‘i’ and ‘j’ in the input graph.

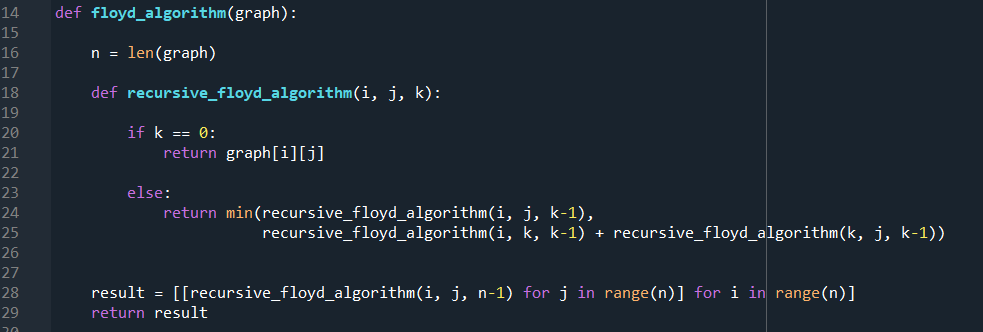
But if k is not zero (recursive case), the function computes the shortest path between vertices ‘i’ and ‘j’ considering the possibility of going through vertex k, and then it recursively calls itself for three cases:

Directly from ’i’ to ‘j’

From ‘i’ to ‘k’ then from ‘k’ to ‘j’

The path from ‘i’ to ‘j’ through intermediate vertices up to ‘k’

**Figure (3) \_floyd\_algorithm**:



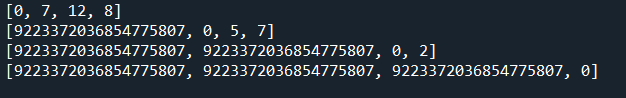
**2.1.d Running the algorithm and printing the result**

The code calls the ‘floyed\_algorithm’ function on the provided graph, iteration over the resulted matrix and prints each row, representing the shortest path between all pairs of vertices.

**Figure (4) \_Code running:**



Updated final graph “solution”. This code will be printed as a two-dimensional list, after treating all the nodes as intermediate nodes:

****

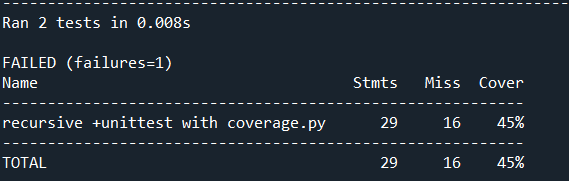
**Chapter** **3. CREATING TESTS AND COVERAGE**

**3.1-unit test with coverage**

‘Unittest’ is used in Python to check a small part of the code in isolation from the rest of the system. (Okken, B.,2017). It was imported from the Pytest package

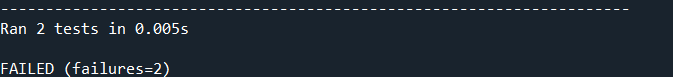
Also, coverage was done for the recursive version and iterative version of both codes by using ‘unittest’ and coverage modules the result for the recursive case was as follows:

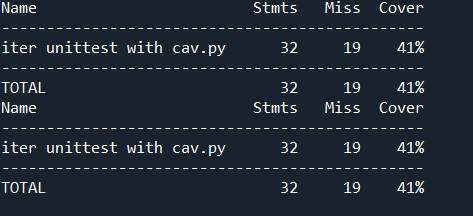
**Figure (5)\_****The unittest with coverage for recursive code**



Two tests were conducted in 0.008 seconds, and one of them filed due to at least one condition in the test was not passed correctly Only 45% of the code was covered by the test, and the test could be reviewed for better results.

**Figure (6)\_The unittest with coverage for iterative code**



****

Two tests were done in 0.005 seconds both failed, the test only covered 41% of the code.

**See** [**Appendix B**](#appb)[**Appendix E**](#appe)

**3.2 performance test**

The performance test was conducted for both iterative and recursive codes through the ‘timeit’ module to measure the excursion time to the ‘floyd\_algorithm function’.

**Figure (6) \_Importing timeit**



Then as described in Figure (3) the functions were defined then

the variable's iteration number was set to 1000 iterations for this test

In the performance test the timeit.time function takes the lambda function that calls the ‘floyd\_algorihm’ function and executes it 1000 times.

**Figure (7) \_Performance test**





The average excursion time per iteration was calculated by dividing the total execution time by the number of iterations.

so, the performance test provides an estimate of the execution time per iteration of floyd\_algorithm.

The result for the recursive version was approximately 133.143 milliseconds.

****

While the iterative version was approximately 0.026699 milliseconds.****

**See** [**Appendix C**](#appc)**,** [**Appendix F**](#appf)

**3.3 Hypothesis to explain the performance test result**

Although the readability and the expression of the recursive code are superior to the iterative coding process the performance test for the iterative gives better results than the recursive process for the same code. In my point of view and after reviewing the reading material and the lecture casts for the previous weeks the difference in the performance test between the recursive and iterative way of coding could be explained by the mechanism of action for each one.

The recursive version of ‘floyd\_algorithm’ shows a longer execution time per iteration than the iterative version, the hypotheses behind this could be due to higher function calls in the recursive version, each recursive call involves an additional function call for the stack, operation in each call needs an extra pushing and popping data onto and off the call stack, which will add overhead compared to the execute of the loop in the iterative code. Moreover, catch locality is better in iterative solutions because it always uses the same block of code. As a result, more memory is used by the recursive code which could impact the performance test, The central processing unit (CPU) in the computer depends on Random access memory (RAM) to treat any input so the operation system calculates the run-time stack and heap on the use of RAM so each recursive function will use this stack for storing each function under Last\_In\_First\_Out (LIFO) (Lee, K.D.,2015) *Chapter 2.*

# Conclusions

This project was to describe the implementation of a code for Floyd\_Warshall algorithm in a recursive way based on the imperative version that was provided, then to do unittest with coverage and the performance test for both the iterative and recursive version and hypothesize the reason behind this result which showed the efficiency of iterative way over the recursive version, all details were provided on the control resources (https://github.com/shorouq-bot/Mid\_Module\_Assignment.git).

REFERENCES

* Aini, A. and Salehipour, A. (2012) “Speeding up the Floyd–Warshall algorithm for the cycled shortest path problem,” *Applied mathematics letters*, 25(1), pp. 1–5. Available at: <https://doi.org/10.1016/j.aml.2011.06.008>.
* Okken, B. (2017) *Python testing with Pytest : simple, rapid, effective, and Scalable / Brian Okken ; edited by Katharine Dvorak.* Version: P1.0 (September 2017). Edited by K. Dvorak. Raleigh, North Carolina: The Pragmatic Bookself.
* Lee, K.D. (2015) *Chapter 2: Computational Complexity*, *Data structures and algorithms with Python*. Cham : Springer,.
* Floyd Warshall Algorithm | DP-16 (2023), available at: https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16

**Appendex.A recursive.py**

# floyd's Warshall code in recursion way without negative value

import sys

# assign the infinity (maximum value )to the NO\_PATH variable to represent the absence of direction

NO\_PATH = sys.maxsize

# the input graph matrex

graph = [

[0, 7, NO\_PATH, 8],

[NO\_PATH, 0, 5, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, 2],

[NO\_PATH, NO\_PATH, NO\_PATH, 0]

]

""" defining the floyd\_algorithm function to calculate the shortest distance between all pairs of vertices in the graph

- argument is the graph which is a list of lists, each element is an integer representing the distance

between two vertices in the graph 'i' and 'j'.

- the return value is the shortest distance between the two vertices after

all vertices have been treated as intermediate nodes"""

def floyd\_algorithm(graph):

# to calculate the number of vertices in the graph

n = len(graph)

""" defining a nested loop to find the shortest distance between all pairs of vertices in the graph

where the function recursively finds the shortest distance between 'i': the starting vertex and 'j': the destination

vertex considering 'k' as an intermediate"""

def recursive\_floyd\_algorithm(i, j, k):

# base case

if k == 0:

return graph[i][j]

# recursive case

else:

return min(recursive\_floyd\_algorithm(i, j, k-1),

recursive\_floyd\_algorithm(i, k, k-1) + recursive\_floyd\_algorithm(k, j, k-1))

# initialise the result matrix

result = [[recursive\_floyd\_algorithm(i, j, n-1) for j in range(n)] for i in range(n)]

return result

# calling the floyd\_algorithm

for row in floyd\_algorithm(graph):

print(row)

**Appendex.B recursive +unittest with coverage.py**

import unittest

import sys

import coverage

NO\_PATH = sys.maxsize

graph1= [

[0, 7, NO\_PATH, 8],

[NO\_PATH, 0, 5, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, 2],

[NO\_PATH, NO\_PATH, NO\_PATH, 0]

]

def floyd\_algorithm(graph):

n = len(graph)

def recursive\_floyd\_algorithm(i, j, k):

if k == 0:

return graph[i][j]

else:

return min(recursive\_floyd\_algorithm(i, j, k - 1),

recursive\_floyd\_algorithm(i, k, k - 1) + recursive\_floyd\_algorithm(k, j, k - 1))

result = [[recursive\_floyd\_algorithm(i, j, n - 1) for j in range(n)] for i in range(n)]

return result

class TestFloydAlgorithm(unittest.TestCase):

def test\_floyd\_algorithm(self):

expected\_result1 = [

[0, 7, 12, 8],

[NO\_PATH, 0, 5, 7],

[NO\_PATH, NO\_PATH, 0, 2],

[NO\_PATH, NO\_PATH, NO\_PATH, 0]

]

result = floyd\_algorithm(graph1)

self.assertEqual(result, expected\_result1)

def test\_floyd\_algorithm2(self):

expected\_result2 = [

[0, 7, 12, 8],

[NO\_PATH, 0, 3, 7],

[NO\_PATH, NO\_PATH, 0, 2],

[NO\_PATH, NO\_PATH,NO\_PATH, 0]

]

result2 = floyd\_algorithm(graph1)

self.assertEqual(result2, expected\_result2)

if \_\_name\_\_ == '\_\_main\_\_':

cov = coverage.Coverage()

cov.start()

unittest.main()

cov.stop()

cov.save()

cov.report()

**Appendex.C performance-test-recursive.py**

import timeit

import sys

NO\_PATH = sys.maxsize

graph = [

[0, 7, NO\_PATH, 8],

[NO\_PATH, 0, 5, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, 2],

[NO\_PATH, NO\_PATH, NO\_PATH, 0]

]

def floyd\_algorithm(graph):

n = len(graph)

def recursive\_floyd\_algorithm(i, j, k):

if k == 0:

return graph[i][j]

else:

return min(recursive\_floyd\_algorithm(i, j, k-1),

recursive\_floyd\_algorithm(i, k, k-1) + recursive\_floyd\_algorithm(k, j, k-1))

result = [[recursive\_floyd\_algorithm(i, j, n-1) for j in range(n)] for i in range(n)]

return result

iteration\_num = 1000

execution\_time = timeit.timeit(lambda: floyd\_algorithm(graph), number=iteration\_num)

print("Average execution time per iteration:", execution\_time / iteration\_num, "seconds")

**Appendex.D iterative.py**

"""floyd warshall algorithm in iterative way without negative """

import sys

import itertools

"""assign the the maximum value (infinite) to NO\_PATH variable """

NO\_PATH = sys.maxsize

# the weighted graph matrix

graph = [[0, 7, NO\_PATH, 8],

[NO\_PATH, 0, 5, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, 2],

[NO\_PATH, NO\_PATH, NO\_PATH, 0]]

# git the number of vertices in the graph

MAX\_LENGTH = len(graph[0])

"""define floyd warshall function to find the shortest distance """

def floyd\_algorithm(distance):

"""iterate over all possible pairs"""

for intermediate, start\_node, end\_node \

in itertools.product\

(range(MAX\_LENGTH), range(MAX\_LENGTH), range(MAX\_LENGTH)):

if start\_node == end\_node:

distance[start\_node][end\_node] = 0

continue

"""to update the shortest distance between the start\_node and the

end\_node via intermediate node """

distance[start\_node][end\_node] = min(distance[start\_node][end\_node], distance[start\_node][intermediate] + distance[intermediate][end\_node] )

# print the result

print(distance)

# call the function

floyd\_algorithm(graph)

**Appendex.E iter unittest with cav.py**

import unittest

import sys

import coverage

import itertools

NO\_PATH = sys.maxsize

graph1 = [[0, 7, NO\_PATH, 8],

[NO\_PATH, 0, 5, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, 2],

[NO\_PATH, NO\_PATH, NO\_PATH, 0]]

def floyd\_algorithm(distance):

MAX\_LENGTH = len(graph1[0])

for intermediate, start\_node, end\_node \

in itertools.product \

(range(MAX\_LENGTH), range(MAX\_LENGTH), range(MAX\_LENGTH)):

if start\_node == end\_node:

distance[start\_node][end\_node] = 0

continue

distance[start\_node][end\_node] = min(distance[start\_node][end\_node],

distance[start\_node][intermediate] + distance[intermediate][end\_node])

print(distance)

class TestFloydAlgorithm(unittest.TestCase):

def test\_floyd\_algorithm(self):

expected\_result1 = [

[0, 7, 12, 8],

[sys.maxsize, 0, 5, 7],

[sys.maxsize, sys.maxsize, 0, 2],

[sys.maxsize, sys.maxsize, sys.maxsize, 0]

]

result = floyd\_algorithm(graph1)

self.assertEqual(result, expected\_result1)

def test\_floyd\_algorithm2(self):

expected\_result2 = [

[0, 7, 12, 8],

[sys.maxsize, 0, 3, 7],

[sys.maxsize, sys.maxsize, 0, 2],

[sys.maxsize, sys.maxsize, sys.maxsize, 0]

]

result2 = floyd\_algorithm(graph1)

self.assertEqual(result2, expected\_result2)

if \_\_name\_\_ == '\_\_main\_\_':

cov = coverage.Coverage()

cov.start()

unittest.main()

cov.stop()

cov.save()

cov.report()

cov.save()

cov.report()

**AppendexF performance-test-iterative.py**

import timeit

import sys

import itertools

NO\_PATH = sys.maxsize

graph = [[0, 7, NO\_PATH, 8],

[NO\_PATH, 0, 5, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, 2],

[NO\_PATH, NO\_PATH, NO\_PATH, 0]]

MAX\_LENGTH = len(graph[0])

def floyd\_algorithm(distance):

for intermediate, start\_node, end\_node \

in itertools.product\

(range(MAX\_LENGTH), range(MAX\_LENGTH), range(MAX\_LENGTH)):

if start\_node == end\_node:

distance[start\_node][end\_node] = 0

continue

distance[start\_node][end\_node] = min(distance[start\_node][end\_node],

distance[start\_node][intermediate] + distance[intermediate][end\_node])

print(distance)

iteration\_num = 1000

execution\_time =timeit.timeit(lambda: floyd\_algorithm(graph), number=iteration\_num)

print("Average execution time per iteration:", execution\_time / iteration\_num, "seconds")