

CSC541 SOFTWARE DEVELOPMENT IN PRACTISE
MARCH 2022

MID-MODULE ASSIGNMENT
FLOYD WARSHALL ALGORITHM

By

Zayd Vawda

Submitted to

The University of Liverpool

MASTER-OF-SCIENCE-TITLE

CSC541 SOFTWARE DEVELOPMENT IN PRACTISE MARCH 2022

Word Count: 1720 (excluding code excerpts)

02/05/2022

FLOYD WARSHALL ALGORITHM: MID-MODULE ASSIGNMENT

Submitted to
The University of Liverpool

Word Count: 1720 (excluding code excerpts)

02/05/2022

TABLE OF CONTENTS

	Page
LIST OF TABLES	2
LIST OF FIGURES	3
Chapter 1. Introduction	4
Chapter 2. development setup	5
2.1 Code Repository Directory	6
2.2 Floyd's Algorithm Original.py	6
2.2.1 Script Introduction and Structure	6
2.2.2 Imperative Code – PEP8 Upgrade	7
2.2.3 Code output	8
2.2.4 Code Testing	9
2.3 Mid_Module_Assignment.py	10
2.3.1 Script Introduction and Structure	10
2.3.2 Code Breakdown.....	12
2.3.3 Code Output.....	14
2.4 Code Testing.py	15
Chapter 3. Supporting documentation	18
Chapter 4. Code evaluation & Conclusions	19
APPENDICES	21

LIST OF TABLES

	Page
Figure 1. Snapshot of Github Repo of the project	6

LIST OF FIGURES

	Page
Figure 1. Snapshot of Github Repo of the project	6

Chapter 1. INTRODUCTION

This report is prepared for the Mid-Module Assignment for the CSCK 541 software development in practice March 2022 cohort. The report details the development and reasoning behind the implementation of a recursive based implementation of the Floyd Warshall Algorithm. The Floyd Warshal algorithm is a common problem in the field of mathematics theory used to estimate the shortest distance between nodes in a matrix.

The original code – in imperative format was provided by the module assignment guidance notes. The report details the work done beginning with the implementation of the imperative code, then the development of the recursive based approach, unit and performative testing of the recursive code as well as the software package supporting files for the newly created package.

Chapter 2. DEVELOPMENT SETUP

This project was developed in Python 3.7. Microsoft Visual Studio Code (VS Code) was used as the IDE for the development. Anaconda 3 was used for environment management. The project was hosted on a public Github repository newly created for this assignment. The link to all python scripts as well as supporting software documentation can be accessed via the repo link below.

Link to repository:

<https://github.com/ZMV-SA/Floyd-s-Algorithm-.git>

2.1 Code Repository Directory

The repository is hosted as a public repo, as snapshot of the included files and scripts are below:

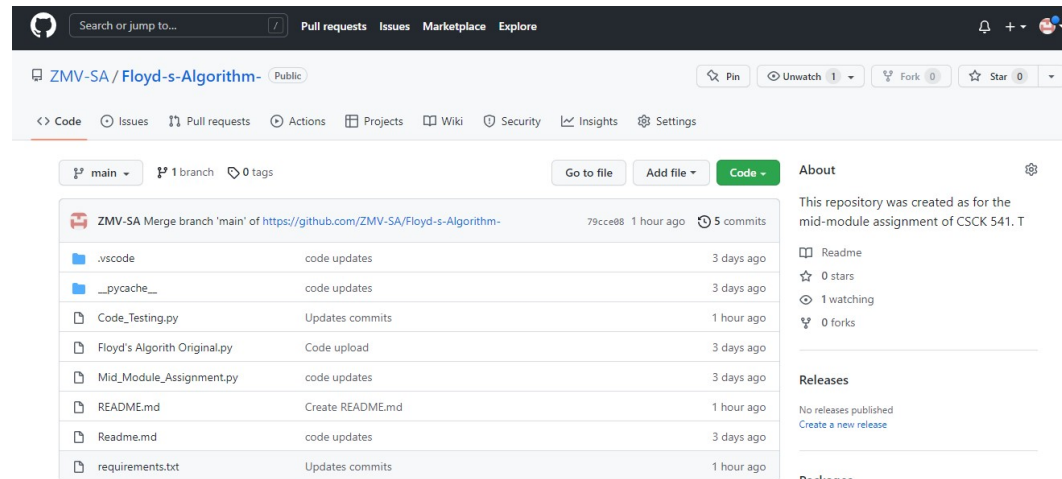


Figure 1. Snapshot of Github Repo of the project

The following files are included in the repo in accordance with the module assignment instructions:

1. Floyd's Algorithm Original.py
2. Mid_Module_Assignment.py
3. Code_Testing.py
4. README.md
5. requirement.txt

The following sections introduces the purpose of each code file and the how to use them.

2.2 Floyd's Algorithm Original.py

2.2.1 Script Introduction and Structure

The module assignment provided the code for an imperative implementation of the Floyd Warshall algorithm. The code was rewritten PEP8 style and run as a benchmark for the other testing. Also taken from assignment notes was the "input matrix" which was used for all implementations in this assignment.

Assignment Input Matrix:

The following 4 x 4 embedded list matrix was used as the input for the code. This was taken from the assignment notes. The input was hard-coded in the code and no user input prompt was implemented to gain the data without modification of the code.

```
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]]
```

2.2.2 Imperative Code – PEP8 Upgrade

This script is best viewed online in an appropriate IDE. A requirement.txt file was also generated and included in the repository. The requirements file has a list of all packages that were used.

```
#This is the original code implementation which has been updated to  
PEP8 standard  
  
import datetime #used to determine the runtime as part of com-  
partive performative testing  
  
start_time =datetime.datetime.now() #initiate timestamp before  
starting of function  
  
import sys # used to check the largest int size for infinite dis-  
tances  
import itertools  
  
NO_PATH = sys.maxsize # used for the infite size distances  
graph = [[0, 7, NO_PATH, 8], # This is the assignement input matrix  
[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(distance):  
    """  
    #A simple implementation of Floyd's algorithm  
    """
```

```

    for intermediate, start_node, end_node in iter-
tools.product(range(MAX_LENGTH), range(MAX_LENGTH),
range(MAX_LENGTH)):
    # Assume that if start_node and end_node are the same
    # then the distance would be zero
    if start_node == end_node:
        distance[start_node][end_node] = 0
        continue
    #return all possible paths and find the minimum
    distance[start_node][end_node] =
min(distance[start_node][end_node],
        distance[start_node][intermediate] + dis-
tance[intermediate][end_node] )
    #Any value that have sys.maxsize has no path
    print (distance)
floyd(graph)

end_time=datetime.datetime.now() #end time stamp one algorithm has
completed running
print ("the Algorithm runtime is \n")
print (end_time-start_time) # print algorithm runtime

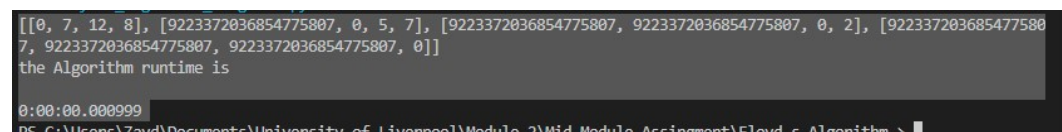
```

Major Code Update:

- Added comments for code including definition of variables
- Added doc strings major modules
- Added indentations
- Added performative testing code to compare to iterative version

2.2.3 Code output

Below is the out of the script:



```

[[0, 7, 12, 8], [9223372036854775807, 0, 5, 7], [9223372036854775807, 9223372036854775807, 0, 2], [9223372036854775807, 9223372036854775807, 9223372036854775807, 0]]
the Algorithm runtime is
0:00:00.000999

```

```

[[0, 7, 12, 8], [9223372036854775807, 0, 5, 7], [9223372036854775807,
9223372036854775807, 0, 2], [9223372036854775807, 9223372036854775807,
9223372036854775807, 0]]

```

The Algorithm runtime is:

0:00:00.000999

The following 4 x 4 nested list matrix was produced as the output which contained the minimized distances between nodes:

```
[[0, 7, 12, 8],
```

```
[9223372036854775807, 0, 5, 7],
```

```
[9223372036854775807, 9223372036854775807, 0, 2],
```

```
[9223372036854775807, 9223372036854775807, 9223372036854775807, 0]]
```

In this code implementation, the 9223372036854775807 value represents an infinite or no-solution distance.

2.2.4 Code Testing

Only performative testing and some manual testing was implemented in this script.

1. Manual testing included a print function for the output.

```
print (distance)
```

Note the manual testing did not format the matrix for easy of reading as a matrix. The output was still reading from the console to benchmark against other Floyd Algorithms developed in this assignment.

2. Performative testing was done via a brute force approach. The datetime module was imported and a time stamp was taken before and after the algorithm was implemented. The difference in the time stamp was used to calculate the execution time.

```
import datetime  
  
start time =datetime.datetime.now() #initiate timestamp before  
starting of function
```

```
end time=datetime.datetime.now() #end time stamp one algorithm has
completed running
```

This brute force method, although valid, does not implement repeat testing, which could have been done with a different test module. However as the code was very simple (a single function), repeated running of the code by the tester, did not alter the test time significantly, this validating the test method.

2.3 Mid_Module_Assignment.py

2.3.1 Script Introduction and Structure

The Mid-Module Assignment primary instruction was to rewrite the above imperative version of the Floyd-Warshall algorithm using a recursion approach. Many such example are available in online literature.

This code used an implementation from (Shivali Bhadaniya, 2022), but was heavily modified for this assignment. The code was further augmented and hardcoded with the assignment input matrix from the module instructions.

The full script can be found in the Github repository, however has been reproduced below.

```
# This Code is an Iterative Solution to the Floyd Warshal Problem
# The code has been adapted from (Shivali Bhadaniya, 2022)
# Which was published on https://favtutor.com/blogs/floyd-warshall-
algorithm.
# The code was sigifanctly modified to conform to
# PEP8 as well as additional changes to code structure and varai-
bles.

import sys # As per Assignement instuctive code, sys needs to be im-
ported to access sys.maxsize.
            # However a very large number could be defined instead.

nv = 4 # Number of vertices
INF = sys.maxsize # Use a large number instead inplace of infinity
aslo equivalent to sys
```

```

# Define the Floyd Warshal Algorithm
def floyd(*args):
    """
        This function is an interative implementation of Floyd Warshall
        algorithm
        p and q and invoked and are used as varaibles indecies for colm
        and row respectivly

        Paremeters:
        arugment1 (list): The input is a 4 x 4 matrix comprised of the 4
        nested loops within a list

        Returns:
        4 x 4 matrix list of integers with the shortes distance between
        nodes
    """
    dist = list(map(lambda p: list(map(lambda q: q, p)), *args))

    # Adding vertices individually using recursion
    for r in range(nv): # p and q and invoked and are used as
        varaibles indecies for colm and row respectivly
            for p in range(nv):
                for q in range(nv):
                    dist[p][q] = min(dist[p][q], dist[p][r] +
dist[r][q])
    sol(dist)

# Printing the output
def sol(dist):
    """
        This fucntion prints the output from the Floyd Warshall Aglo-
        rithm defined above this
        n and p and invoked and are used as varaibles indecies for colm
        and row respectivly

        Parameters:
        The input is a list map of generated by Floyd Aloroghtm

        Returns:
        The function will iterate through the mapped list produced by
        the Floyd function and print it

    """
    for p in range(nv): #p and q and invoked and are used as
        varaibles indecies for colm and row respectivly
            for q in range(nv):
                if(dist[p][q] == INF):
                    print("INF", end=" ")

```

```

        else:
            print(dist[p][q], end=" ")
        print(" ")

G = [[0, 7, INF, 8],    # This is the input Matrix from the assign-
ment
      [INF, 0, 5, INF],
      [INF, INF, 0, 2],
      [INF, INF, INF, 0]]

H = [[0, 5, INF, INF], #This is an input Matrix from (Shivali
Bhadaniya, 2022), it is also used in unit testing. The correct out-
put is known for this input and is used for validation. This is in-
cluded here as for of manual testing.
      [50, 0, 15, 5],
      [30, INF, 0, 15],
      [15, INF, 5, 0]]

print("\nTest 1:\n" )
floyd(G)
print ("\nTest 2:\n")
floyd(H)
print("\n")

```

2.3.2 Code Breakdown

Line 1-6:

Code introduction as well as reference and credit to sources utilized.

Line 7-13:

The sys library was imported to make use of the .maxsize method. This allowed us to define infinity or the situation where the distance between the nodes was non-existent. In addition nv was declared to indicate the number of vertices or nodes. This was made static at 4 as per the assignment instructions and dimensions of the input matrix. INF was used a variable stand-in for output of the sys.maxsize out.

Line 14-24:

The main algorithm called Floyd was defined here. The docstring for the function was also implemented using the PEP-8 Standard.

Line 25-27:

Distance between nodes is created here using a recursive function.

1. The variable dist is created to hold the output.
2. The variable is defined as a list
3. The map iterator is used to apply a function to every item in list iterable
4. Within the map iterator is two nested lambda functions, one for p and q which then reads in the columns and rows on in the input matrix. The input matrix is then mapped to dist

Line 28 – 33:

Three nested for loops are then called which use the search space created by the nv (matrix dimension) variable to loop through the input matrix (dist)

The dist list is then modified iteratively by using the min function which finds the min distance between nodes. The nv variable limits the number of steps to 4 ensuring there arnt an infinite amount of dimensioned searched. nv is parsed to variable r which is used in this loop.

The dist list is now updated with the min distances.

List 34-55:

A new function called sol defined to print the output of the floyd algorithm.

A docstring was written for this simple function.

The sol function steps through the dist nested list. Variables p and q are again used to defined the column and rows respectively.

Two nested for statements are used to loop through the elements within the list.

INF is defined as a dedicated print statement provided for those items which are equal to the max system size. This is used for ease of reading the output.

New line print points are included to pad the printed output for ease of reading and display the output as a matrix instead of linear string of characters.

Line 56 – 65:

G and H are defined here. There are input matrixes to implement in the floyd algorithm and are hard coded.

G and H are both 4 x 4 embedded list type variables.

G is defined as the input matrix from the assignment instruction which is also used in the original floyd algorithm script described above.

H was as input matrix with a known correct output from (Shivali Bhadaniya, 2022). This is used as a manual unit test to validate that the function.

Line 65-70:

These lines were used as the wrapped to parse the input matrixes (assignment and manual unit test) as well as print the output the to terminal.

2.3.3 Code Output

Once run the code output is as below:

```
PS C:\Users\Zayd\Documents\University of Liverpool\Module 2\Mid Module Assingment\Floyd-s-Algorithm-> & C:\Users\Zayd\anaconda3\envs\Floyd-s-Algorithm-/python.exe "c:/Users/Zayd/Documents/University of Liverpool/Module 2/Mid Module Assingment/Floyd-s-Algorithm-/Mid_Module_Assignment.py"

Test 1:
0 7 12 8
INF 0 5 7
INF INF 0 2
INF INF INF 0

Test 2:
0 5 15 10
20 0 10 5
30 35 0 15
15 20 5 0
```


The output of Test 1 is from the assignment and is the same as the output from the imperative code with the exception that INF was defined in place of the long string which is a numeric value of the system max size.

The output of Test 2 is the same as the validated output from (Shivali Bhadaniya, 2022), indicating that the algorithm works.

Further unit testing is described in the Code_Testing script, see section below.

2.4 Code Testing.py

Unit testing and performative testing was done via a dedicated testing script. The same Mid-Module_Assignment code was clones but only a single input matrix was used.

Unit testing:

Unit testing was done in two ways: Manual testing and as well as unit testing via the unittest library. Both indicated that the code was successful.

Unit testing was done using the following code extract:

```
34 # Testing Code: Unit Test of Floyd Function -----
35 # Unit test of Floyd Algorithm floyd
36 Test_Output = [[0, 5, 15, 10], [20, 0, 10, 5], [30, 35, 0, 15], [15, 20, 5, 0]] # This is the known correct output from the Floyd Algorithm f
37 assert dist == Test_Output, "Test failed"
38 # Testing Code: Unit Test of Floyd Function -----
39
```

The asset statement was hardcoded with the expected output from the known input/output pair from (Shivali Bhadaniya, 2022).

The manual unit test printed the output to the terminal which was just validated by comparing the terminal read out to the expected output from (Shivali Bhadaniya, 2022),

```
Manual Unit Test:
0 5 15 10
20 0 10 5
30 35 0 15
15 20 5 0

Algorithm runtime is:0:00:00.000366
PS C:\Users\Zayd\Documents\University of Liverpool\Module 2\Mid Module Assingment\Floyd-s-Algorithm-> |
```

Performative testing

Performative testing was carried out in a similar fashion as described in for the imperative script. A brute force method was employed using the datetime function. A time stamp was created before and after the algorithm was called and the difference in time was printed.

```
end_time=datetime.datetime.now() #end time stamp one algorithm has completed running
print("Algorithm runtime is:",end= "")
print (end_time-start_time) # print algorithm runtime
```

The code had a runtime of approximately 0.00036 ms which did not vary significantly when repeated run.

The code for the unit and performative testing is presented below:

```
# This Python Notebook is used to do unit and performative testing on
on the Mid-Module Assignment Floyd Warshall Code
# The test input and output were taken from (Shivali Bhadaniya,
2022), https://favtutor.com/blogs/floyd-warshall-algorithm.

import unittest
import datetime #Used to conduct performative testing
import sys # As per Assignment instructive code, sys needs to be im-
ported to access sys.maxsize.
            # However a very large number could be defined instead.

start_time =datetime.datetime.now() #initiate timestamp before
starting of function

nv = 4 # Number of vertices
INF = sys.maxsize # Use a large number instead inplace of infinity
also equivalent to sys

def floyd(*args):
    """
    This function is an iterative implementation of Floyd Warshall
    algorithm

    Parameters:
    argument1 (list): The input is a 4 x 4 matrix comprised of the 4
    nested loops within a list

    Returns:
    4 x 4 matrix list of integers with the shortest distance between
    nodes
```

```

"""
dist = list(map(lambda p: list(map(lambda q: q, p)), H))

# Adding vertices individually using recursion
#p and q and invoked and are used as variables indices for column
and row respectively
for r in range(nv):
    for p in range(nv):
        for q in range(nv):
            dist[p][q] = min(dist[p][q], dist[p][r] +
dist[r][q])
    sol(dist)

# Testing Code: Unit Test of Floyd Function -----
-----
# Unit test of Floyd Algorithm floyd
Test_Output = [[0, 5, 15, 10], [20, 0, 10, 5], [30, 35, 0, 15],
[15, 20, 5, 0]] # This is the known correct output from the Floyd
Algorithm from (Shivali Bhadaniya, 2022)
assert dist == Test_Output, "Test failed"
# Testing Code: Unit Test of Floyd Function -----
-----

# Printing the output, This is a form of Manual testing also check
the output is correct
def sol(dist):
    """
    This function prints the output from the Floyd Warshall Algo-
rithm defined above this

    Parameters:
    The input is a list map of generated by Floyd Algorithm

    Returns:
    The function will iterate through the mapped list produced by
the Floyd function and print it

    """
    for p in range(nv):
        for q in range(nv):
            if(dist[p][q] == INF):
                print("INF", end=" ")
            else:
                print(dist[p][q], end=" ")
        print(" ")

```

```

H = [[0, 5, INF, INF], #This is an input Matrix from (Shivali
Bhadaniya, 2022), it is also used in unit testing. The correct out-
put is known for this input and is used for validation. This is in-
cluded here as for of manual testing.
      [50, 0, 15, 5],
      [30, INF, 0, 15],
      [15, INF, 5, 0]]

print ("\nManual Unit Test:\n")
floyd(H)
print("\n")

end_time=datetime.datetime.now() #end time stamp one algorithm has
completed running
print("Algorithm runtime is:",end= "")
print (end_time-start_time) # print algorithm runtime

# The matrix below is the known correct output from (Shivali
Bhadaniya, 2022)
# 0  5  15  10
# 20  0  10  5
# 30  35  0  15
# 15  20  5  0

```

Chapter 3. SUPPORTING DOCUMENTATION

Two supporting documents were created for this project. A readme file (which is a clone of this report) as well as requirements file.

Chapter 4. CODE EVALUATION & CONCLUSIONS

The imperative code (Floyd's Algorithm Original) was successfully implemented with the unit test confirming the same input and output.

The recursive script (Mid_Module_Assigement) was also successful with the same input and out from the imperative being produced. Unit testing from an external source also indicate that the code does work.

The imperative code had an execution runtime of 0.001003 ms while the recursive version had a runtime of 0.001997 ms, a difference of 0,000994 ms. The difference is very small and could likely be due to the fact that the recursive version requires extra instruction cycles to define the INF variable (instead of the system maxsize) as well as outputs the results in a ordered 4 x 4 matrix instead of a simple string of characters. This extra "padding" to make the output for reading may increase the runtime.

REFERENCES

Bhadaniya, S., 2022. *Floyd Warshall Algorithm (Python) | Dynamic Programming | FavTutor*. [online] FavTutor. Available at: <<https://favtutor.com/blogs/floyd-warshall-algorithm>> [Accessed 1 May 2022]. Bhadaniya, S., 2022. *Floyd Warshall Algorithm (Python) | Dynamic Programming | FavTutor*. [online] FavTutor. Available at: <<https://favtutor.com/blogs/floyd-warshall-algorithm>> [Accessed 1 May 2022].

APPENDICES