recursion version of Floyd’s algorithm

By

Moustafa S. Fouda

Submitted to

The University of Liverpool

MASTER-OF-SCIENCE

Data science and Artificial Intelligence

*Software development in practice*

Word Count: XXX

26/02/2024

TABLE OF CONTENTS

Page

[LIST OF TABLES 2](#_Toc159558363)

[LIST OF FIGURES 3](#_Toc159558364)

[Chapter 1. Introduction 4](#_Toc159558365)

[Chapter 2. Floyd Warshall Algorithm (Python) 4](#_Toc159558366)

[2.1 What is Floyd algorithm 4](#_Toc159558367)

[2.2 Floyd Marshall Recursive 4](#_Toc159558368)

[2.3 Testing and coverage 6](#_Toc159558369)

[2.3.1 Testing 6](#_Toc159558370)

[2.3.2 Coverage 7](#_Toc159558371)

[2.3.3 iterative vs recursive performance 8](#_Toc159558372)

[chapter 3. Conclusion 9](#_Toc159558373)

[REFERENCES 10](#_Toc159558374)

LIST OF FIGURES

Page

[Figure 1. recursive in single function (reference) 4](#_Toc159581748)

Figure 2. recursive algorithm in separate functions to enhance readibility. 6

Figure 3. Time testing of the algorithm. 6

Figure 4. Unit test of Floyd-Warshall algorithm. 7

Figure 5. coverage of the algorithm. 8

# Introduction

In this work we will discuss the use of recursive version of Floyd algorithm. We will also define the limitations and advantages of the recursive version over the iterative one while giving some deep insights of the performance of the code. The 2nd chapter of the code will discuss mainly how the code was built in detail. While the 3rd part will investigate the performance testing and coverage of this code. In later part will present the core differences between the recursive and iterative approaches while the last one will be the conclusion.

# Floyd Warshall Algorithm (Python)

What is Floyd algorithm

“The Floyd–Warshall algorithm is a simple and widely used algorithm to compute shortest paths between all pairs of vertices in an edge weighted directed graph.” (Hougardy, 2010). This is done by considering all vertices as intermediate ones and finding the shortest path. Each time the function calls itself to perform the same steps on the remaining points, the start of the new call is the end of the last one till an end which we call the base case.

Recursion relies on three main rules:

* Function that calls itself recursively.
* With movement and change of state to a;
* base code (where it stops)

in the following part, we will try to find the shortest path recursively.

Floyd Marshall Recursive

The core of the code is the function that finds the shortest path between two vertices. While this can be done in one function as follows:

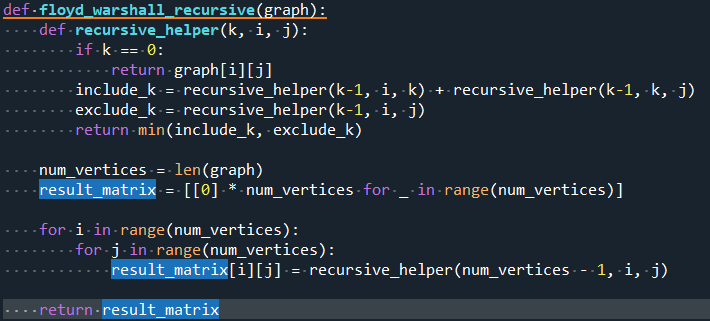


Figure 1. recursive in single function (reference)

It was found clearer and more readable to split the code in two functions where one for recursion itself and the other is defining the shortest path as follows:

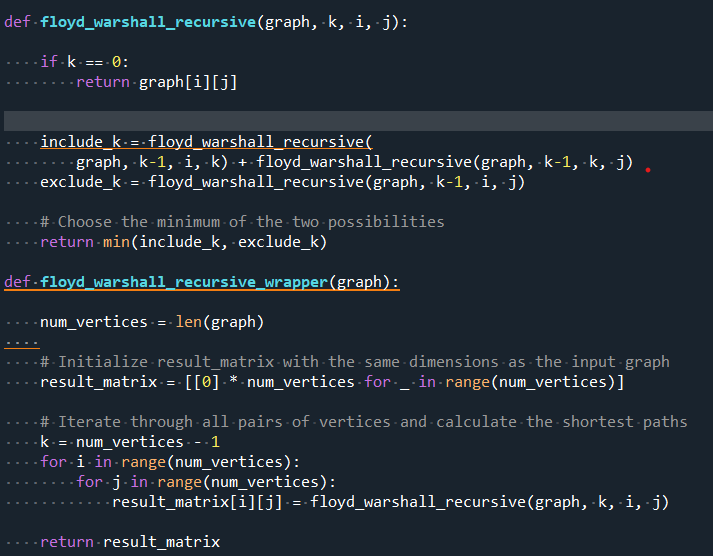


Fig.2 recursive algorithm in separate functions to enhance readibility.

Where;

- graph: The adjacency matrix representation of the graph.

- k: Intermediate vertex

- i, j: Source and destination vertices

The first function is to take the minimum output if we consider an intermediate vertex. while the second function iterates through all veertices considering each as intermediate.

The implementaiton using separate functions increase readabilty and make the code easy to maintain in case of adding features. However, we kept both functions in Git utilizing two different branchs main and seperatefunction, respectively.

Breaking down the code:

First part simply taking the minimum value between direct distance between vertices or considering a midpoint. While the function keeps calling it self in recursion till all points are consumed.

Second part, starting with filling matrix with zeros while matrix size is dependent on the number of vertices. Then iterate through all pairs of vertices with the same procedure.

Testing and coverage

### Testing

It is essential to test the code to make sure that it is functioning as expected and apply the relative adjustment if it is not.

The two types of tests that we followed in this work were: performance test and unittest.

1st performance test: In this test we examine the time required for the function to be executed on the example sample. The output is time consumed for execution.

Implementing this code is via setting the start time and end time before and after execution then perform calculations. While the best practice for the implementation is to test more that one test set and take the average time for better accuracy utilizing “timeit” package for the execution

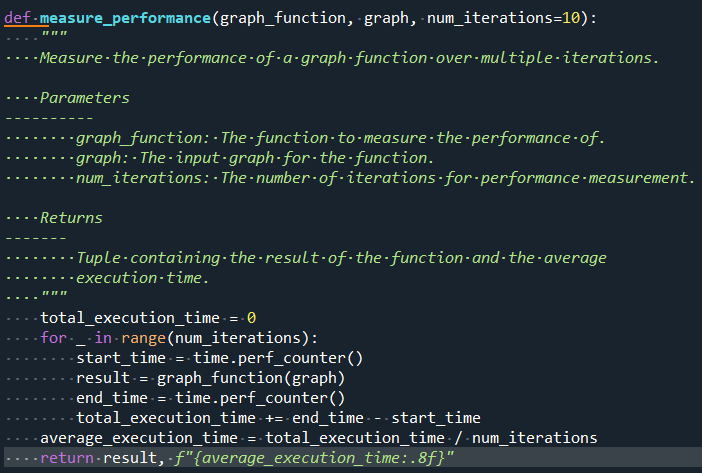


Fig.3 Time testing of the algorithm

2nd unit test

Here, we test the function of the code and check it the results are as expected. The criteria are to assert that a known result using known input and output. If the output match with no errors, then the code will give Ok as a result otherwise it will show that it failed showing the expectation against output. Note that float(“inf”) is considered as no path between the points.

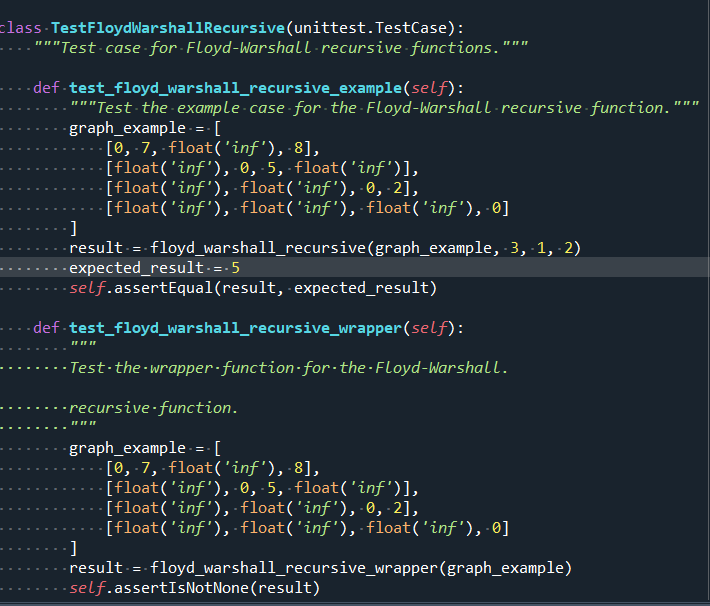


Fig.4 Unit test of Floyd-Warshall algorithm

Finaly numerous tests can be applied depending on the code and the function required. Testing can be included in the main code as a part of the code lines and can also be separate by modulating the recursion and produce separate python file for testing. In this work we have shown both methods in two separate branches: “testoncode” and “testoffcode” respectively.

### Coverage

In this section we provide the testing coverage for both the main function and the testing functionality.

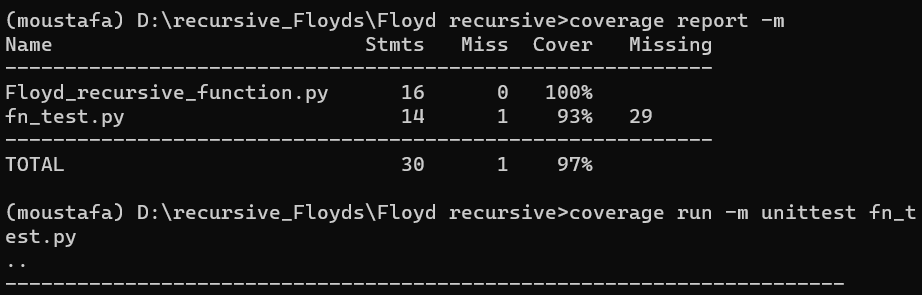


Fig.5 coverage of the algorithm

while the results are showing 97% due to the presence of the last line



Other than that the coverage of the tests are 100% which is good result.

### iterative vs recursive performance

while we have implemented both alogrithms the reuslts showed that the iterative approach can be faster for larger numbers of inputs while for less input recursive can be faster.

For this time performance iterative algorithm has a polynomial time complexity with respcet to the inputs O(n^3) while the recursive version has an exponential time complexity O(n^k) where n is the input size and k is the number of intermediate vertices.

Applying the test for both large input size and lower input size considering O notation above we can find that in case of low input size recursive approach can be faster, however due to the exponential time complexity of the recursive approach it becomes impractical in larger sizes. This have been tested in the “time\_test\_iterative\_vs\_recursive.py”

# chapter 3. Conclusions

Recursive approach and iterative approaches can be used interchangeably based on the context and the best for the specific case while recursion can be easier in breaking down the code to more readable form, increased inputs would compromise the performance due to exponential complexity. Testing and coverage checks are important to ensure that the code is not only functional but also optimized. For this specific case, increasing the input above 10 would make the recursive method slower however it is faster for the lower input size. We have pushed our work into GitHub utilizing different branches and multiple commits for the ease of maintenance.

REFERENCES

Hougardy, S. (2010) ‘The Floyd–Warshall algorithm on graphs with negative cycles’, *Information processing letters*, 110(8), pp. 279–281. Available at: https://doi.org/10.1016/j.ipl.2010.02.001.