RECUSRIVE IMPLEMENTATION OF FLOYD'S ALGORITHM

By

Edward Halsall

Submitted to

The University of Liverpool

MASTER-OF-SCIENCE-COMPUTER-SCIENCE

Software Development in Practice

Word Count: 1000

26/02/2024

LIST OF FIGURES

Page

Chapter 1. INTRODUCTION

Floyd-Warshall method is famous for its capacity to find the shortest pathways between any pair of vertices in a weighted graph. Using the concepts of dynamic programming, this approach iteratively updates a distance matrix in a methodical manner (Hougardy, 2010). Although iterative constructions are typically employed to implement algorithms, recursion presents a different approach to describing the solution to the problem. Recursive methods occasionally help to clarify the rationale of the algorithm and may also disclose various aspects of its performance. An algorithmic recursive implementation is shown in this report. A comprehensive description of the recursive formulation is given together with a testing protocol to verify its accuracy. In addition, a performance analysis contrasts the traditional iterative version with the recursive implementation, providing information on the computational trade-offs between these methods.

Repository: https://github.com/edwardh98/floyd-warshall-finished-assignment

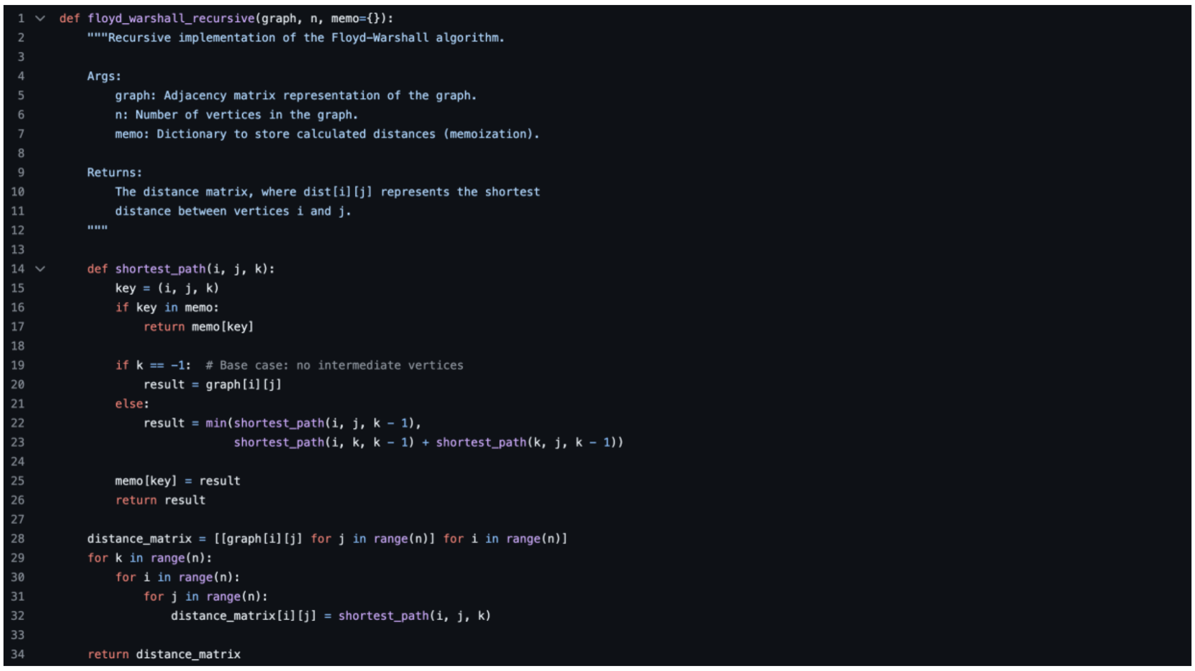
Chapter 2. FLOYD-WARSHALL RECURSIVE AND ITERATIVE ALGORITHM

2.1 Explanation of recursive code

The provided code implements the Floyd-Warshall algorithm using a recursive approach, the code is divided into two core functions.

Floyd_warshall_recursive: This function establishes the overall structure of the solution. It initialises a distances matrix iterates through potential intermediate vertices, and coordinates the core recursive calculations performed by shortest_path

shortest_path: This function is the heart of the recursion. It calculates the shortest path between two vertices (i and j) while considering intermediate vertices up to a given index (k). It strategically employs memoization to store and reuse results, optimizing the calculation.



```python
def floyd_warshall_recursive(graph, n, memo={}):
    """Recursive implementation of the Floyd-Warshall algorithm.

    Args:
        graph: Adjacency matrix representation of the graph.
        n: Number of vertices in the graph.
        memo: Dictionary to store calculated distances (memoization).

    Returns:
        The distance matrix, where dist[i][j] represents the shortest
        distance between vertices i and j.
    """

    def shortest_path(i, j, k):
        key = (i, j, k)
        if key in memo:
            return memo[key]

        if k == -1:  # Base case: no intermediate vertices
            result = graph[i][j]
        else:
            result = min(shortest_path(i, j, k - 1),
                         shortest_path(i, k, k - 1) + shortest_path(k, j, k - 1))

        memo[key] = result
        return result

    distance_matrix = [[graph[i][j] for j in range(n)] for i in range(n)]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                distance_matrix[i][j] = shortest_path(i, j, k)

    return distance_matrix
```

Figure 1. Recursive Floyd-Warshall Algorithm

The floyd_warshall_recursive function
Purpose: The outer function initialises the framework for the recursive solution.

graph: Takes the adjacency matrix of the input graph.
n: The number of vertices in the graph.
memo: A dictionary for memoization, storing calculated results.

Initialisation: Creates a distance_matrix initially filled with direct edge weights from the graph.

Loops: Iterates through all potential intermediate vertices (k), and for each combination of vertices i and j, calls the shortest_path function to find the shortest path.

Return: Returns the final distance_matrix containing shortest distances between all pairs.

The shortest_path function
Purpose: The core recursive function that calculates the shortest path between vertices i and j with a maximum intermediate vertex index of k.

Key: Creates a unique tuple to represent a specific subproblem (i, j, k) for memoization.

Memoization Check: If the result for this subproblem is already in memo, it's directly returned.

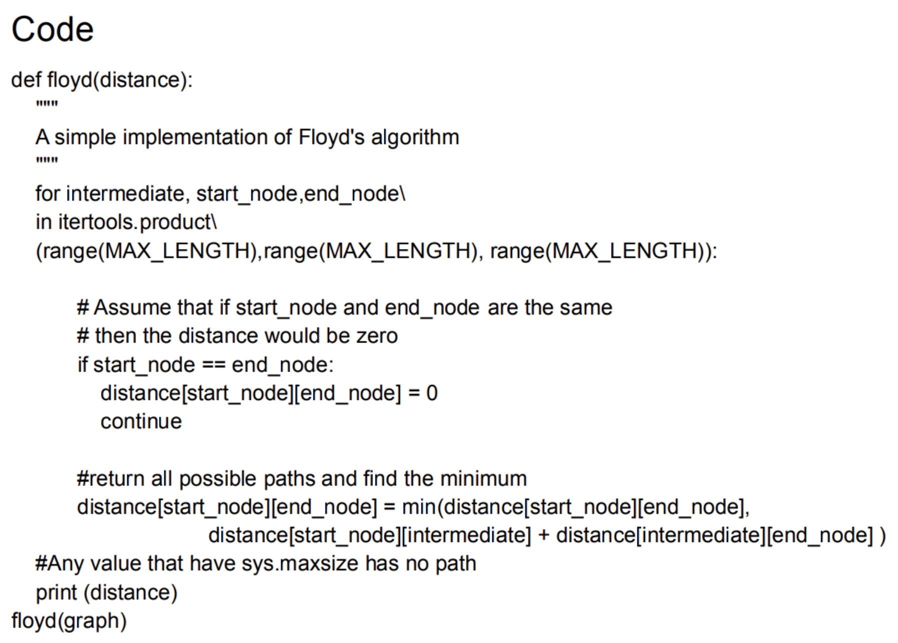
Base Case (k == -1): If no intermediate vertices are allowed, the shortest path is the direct edge weight in the graph.

Recursive Case:
Compares the current shortest path (shortest_path(i, j, k-1)) to the potential path obtained by going through vertex k. Stores the shorter path in result.

Memoization Update: Stores the calculated result in the memo dictionary before returning.

2.2 Explanation of Iterative code

The iterative version systematically updates a distance matrix with the shortest known distances between all pairs of vertices. It does this by iterating through all potential intermediate vertices and checking if using a specific intermediate vertex would shorten the existing path between a pair of nodes.



Code

```
def floyd(distance):
    """
    A simple implementation of Floyd's algorithm
    """
    for intermediate, start_node,end_node\
    in itertools.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] + distance[intermediate][end_node] )
    #Any value that have sys.maxsize has no path
    print (distance)
floyd(graph)
```

Figure 2. Iterative floyd-warshall Algorithm

floyd(distance) Function:
Takes a distance matrix. It's assumed this matrix is properly initialized, likely with direct edge weights and 'infinity' values representing unconnected nodes.

itertools.product Loop:

Creates combinations of all possible (intermediate, start_node, end_node) triples representing potential paths.

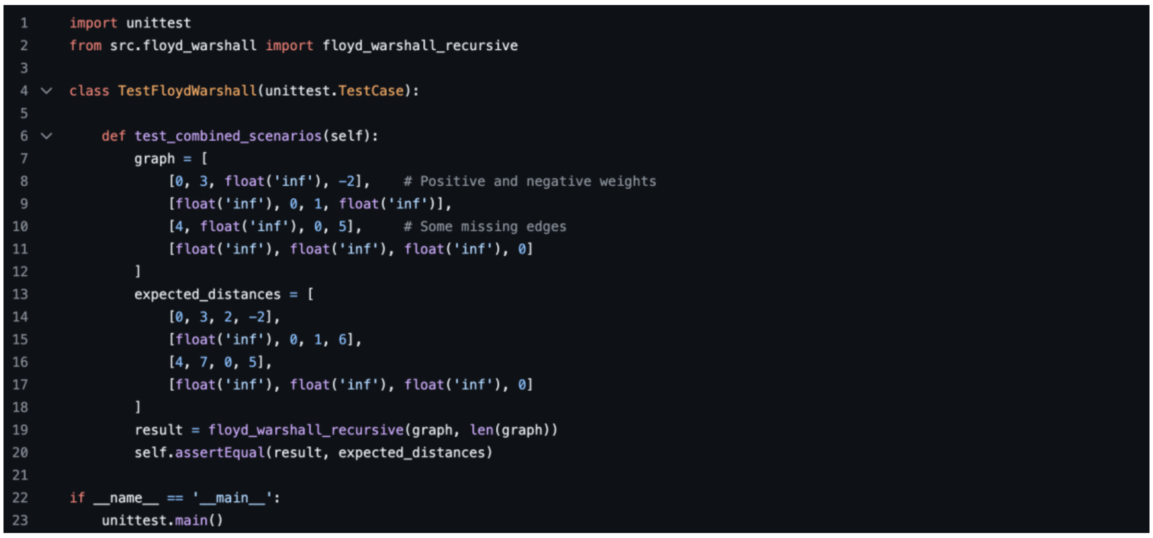
Zero Distance Check:
A node to itself has a distance of 0. This check establishes that base case.

Path Update:
distance[start_node][end_node] = min(...): This is the critical step. It compares:
The current known distance between start_node and end_node.
The potential distance if you go through intermediate (distance[start_node][intermediate] + distance[intermediate][end_node]).

The shorter of these distances is stored back into distance[start_node][end_node].

2.3 Testing

The primary goal of this testing strategy is to verify the correctness of the recursive implementation of the algorithm. This test verifies the algorithm's output by comparing the calculated distances against a pre-determined expected_distances array. This array was used to ensure it represents the true shortest paths for the input graph. The test includes positive and negative weights to ensure the algorithm handles these scenarios, which are fundamental to shortest path problems. Missing edges, represented as 'inf', test the algorithm's ability to work with partially connected graphs.



```python
import unittest
from src.floyd_warshall import floyd_warshall_recursive

class TestFloydWarshall(unittest.TestCase):

    def test_combined_scenarios(self):
        graph = [
            [0, 3, float('inf'), -2],    # Positive and negative weights
            [float('inf'), 0, 1, float('inf')],
            [4, float('inf'), 0, 5],     # Some missing edges
            [float('inf'), float('inf'), float('inf'), 0]
        ]
        expected_distances = [
            [0, 3, 2, -2],
            [float('inf'), 0, 1, 6],
            [4, 7, 0, 5],
            [float('inf'), float('inf'), float('inf'), 0]
        ]
        result = floyd_warshall_recursive(graph, len(graph))
        self.assertEqual(result, expected_distances)

if __name__ == '__main__':
    unittest.main()
```

Figure 3. Testing

Function Behaviour:
The test demonstrates the core behaviour of the shortest_path recursive function. By including a mix of weights and connectivity, it implicitly exercises how the function evaluates potential paths through different intermediate vertices and updates distances when shorter paths are discovered.

Edge Cases:
While this test provides a coverage of basic scenarios, a more comprehensive test suite could include cases with larger graphs to analyse.

2.4 Iterative vs Recursive

This will discuss the theoretical differences between recursive and iterative implementations of the Floyd-Warshall algorithm.

Recursion Overhead:
Recursive function calls involve overhead for setting up the call stack and managing the return process. This overhead can be significant, especially for larger graphs or deeper recursion.

Memoization:
The iterative version doesn't explicitly use memoization, while the recursive version utilises a memo dictionary to store intermediate results. This can significantly improve the performance of the recursive version by avoiding redundant calculations.

Data Access Patterns:
The iterative version has simpler data access patterns, iterating directly through the distance matrix. The recursive version might involve more complex memory accesses due to the recursive calls and potential cache misses.

In general, the iterative version would be expected to be faster than the recursive version due to the lower overhead and simpler data access patterns. However, the memoization used in the recursive version can mitigate the overhead and sometimes make it competitive, especially for larger graphs where redundant calculations become more frequent (Cormen, 2022).

Chapter 3. CONCLUSION

This report presented a recursive implementation of the Floyd-Warshall algorithm, detailed its structure and use of memoization, and outlined a testing strategy to verify its correctness. While a direct performance comparison was outside the scope, a theoretical analysis was conducted, examining potential performance differences between recursive and iterative approaches. The analysis highlighted factors such as recursion overhead and memoization, which can influence the performance characteristics of each implementation. Implementing Floyd's algorithm recursively offered a different perspective on the algorithm's core logic. The recursive formulation highlights the way the problem can be decomposed into smaller subproblems.

REFERENCES

Cormen, T.H. et al. (2022) Introduction to algorithms Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Fourth edition. Cambridge, Massachusett: The MIT Press. (pp 519 – 524)

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.