PRACTICAL ASSESSMENT: PROGRAMMING – FLOYD'S ALGORITHM

By

Maikel Handersonn Michelon

Submitted to

The University of Liverpool

MASTER-OF-SCIENCE-DATA SCIENCE AND ARTIFICIAL INTELLIGENCE

Software development in practice

Word Count: 2512

10/07/2023

PRACTICAL ASSESSMENT: PROGRAMMING – FLOYD'S ALGORITHM

Submitted to

The University of Liverpool

Word Count: 2512

10/07/2023

TABLE OF CONTENTS

LIST OF FIGURES

Page

Chapter 1. INTRODUCTION

The Floyd Warshall algorithm is a widely used algorithm for finding the shortest paths between all pairs of vertices in a graph. It is known for its simplicity and effectiveness in solving the all-pairs shortest path problem. The algorithm works by iteratively considering all intermediate vertices as potential stops along the paths between pairs of vertices. It updates the distance between two vertices if a shorter path is found through the intermediate vertex. By repeatedly applying this process for all possible intermediate vertices, the algorithm gradually builds up the shortest paths between all pairs of vertices.

In this report, the goal is to implement a recursive version of Floyd Warshall algorithm, write unit tests for each function, and compare the performance of the recursive version with the iterative version mentioned in the provided resource. To ensure code quality and version control, the report will be placed under source control using GitHub. The directory tree will be organized, and code documentation will be provided, including a readme.md file and a requirements.txt file for easy setup of project dependencies.

The report will discuss the process of building the application, including the steps taken to implement the recursive version of Floyd's algorithm. The report will also cover the creation of unit tests, the test, and the performance tests conducted and compare the performance of the recursive version with the iterative version mentioned in the resource and provide a hypothesis to explain any differences observed. To ensure accessibility, the code and documentation will be hosted on an online service (GitHub), and settings will be configured to allow anyone with the direct link to access the repository.Throughout the project, best practices will be followed, maintaining clear documentation, and providing a comprehensive report to showcase implementation and findings.

Chapter 2. FLOYD WARSHALL ALGORITHM AND RECURSION

Recursion is a problem-solving strategy in computer programming that involves functions calling themselves repeatedly until a base case is reached. It is a powerful technique used in various disciplines, including mathematics, bioinformatics, linguistics, art, and nature. Recursion and iteration are equivalent in solving problems, and choosing between them depends on factors like the problem at hand, efficiency, programming language, and paradigm.

Recursive algorithms have several advantages. They are often simpler to design and understand compared to iterative algorithms, as they closely resemble the logical approach humans would take to solve a problem. Recursive algorithms also make use of the program stack implicitly, eliminating the need for explicit stack management in iterative algorithms. This makes them intuitive, concise, and easier to comprehend, especially when the problem or data structure resembles a tree.

However, recursive algorithms generally have drawbacks in terms of efficiency and memory usage. Since each function call allocates memory on the program stack, recursive algorithms may be slower and consume more memory compared to iterative versions. This overhead is inherent to the use of the program stack.

Figure 1 illustrates the recursive implementation of Floyd's algorithm; The following synopsis outlines the step-by-step code pertaining to this algorithm:

1. The code begins by importing the sys module and the itertools module. Sys.maxsize is used to represent infinity, indicating that there is no direct path between two nodes in the graph.
2. The graph variable is a 2d list representing the weighted graph. Each element graph [i][j] represents the weight of the edge from node i to node j. If there is no edge between the nodes, the value is set to sys.maxsize.
3. Max_length is set to the length of the graph (number of nodes).
4. The floyd_recursive function is a recursive implementation of floyd's algorithm. It takes four parameters: distance, intermediate, start_node, and end_node.
5. The base case of the recursive function is when start_node is equal to end_node. In this case, the distance between the two nodes is set to 0.

6. Otherwise, the function updates the distance between start_node and end_node by taking the minimum of the current distance and the sum of the distances from start_node to intermediate and from intermediate to end_node.
7. If the value of intermediate is less than max_length - 1, the function calls itself recursively with an incremented value of intermediate.
8. The floyd function is the main implementation of floyd's algorithm. It takes the distance matrix as input.
9. The function uses itertools.product to iterate over all possible combinations of intermediate, start_node, and end_node values.
10. For each combination, the floyd_recursive function is called to update the distance matrix.
11. After the algorithm completes, the resulting distance matrix is printed row by row.
12. Finally, the floyd(graph) function is called with the given graph to execute the algorithm and print the resulting distance matrix.

```python
#recursiveversion_floydsalgorithn

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_recursive(distance, intermediate, start_node, end_node):
    """
    Recursive implementation of Floyd's algorithm
    """
    if start_node == end_node:
        distance[start_node][end_node] = 0
        return

    distance[start_node][end_node] = min(
        distance[start_node][end_node],
        distance[start_node][intermediate] + distance[intermediate][end_node]
    )

    if intermediate < MAX_LENGTH - 1:
        floyd_recursive(distance, intermediate + 1, start_node, end_node)


def floyd(distance):
    """
    Implementation of Floyd's algorithm
    """
    for intermediate, start_node, end_node in itertools.product(range(MAX_LENGTH),
range(MAX_LENGTH), range(MAX_LENGTH)):
        floyd_recursive(distance, intermediate, start_node, end_node)

    # Print the resulting distance matrix
    for row in distance:
        print(row)


# Call the function with the graph
floyd(graph)
```

Figure 1: Floyd's Algorithm using recursion (Developed and adapted by the author)

2.1 Floyd Warshall algorithm (recursion) - Unit test

In unit testing, the goal is to test the fundamental components of an application at a granular level. Each individual unit of code, usually a method, is tested in isolation to determine if it behaves as expected under specific conditions. By breaking testing down to this level, it ensures that every part of the application functions correctly and allows for the handling of unexpected scenarios. Unit testing provides confidence in the application's behavior and helps uncover and address edge cases.

The code on figure 2 represents a unit test case for the floyd algorithm, which is a recursive implementation of floyd's algorithm. The test steps are:

1. The code starts with importing the required modules: sys, itertools, and unittest. These modules provide necessary functionality for testing and defining test cases.
2. The no_path constant is set to sys.maxsize, representing infinity or the absence of a direct path between nodes in the graph.
3. The graph variable is a 2d list that represents the weighted graph on which the algorithm will be tested. It contains a matrix of weights representing the edges between nodes.
4. The max_length variable is set to the length of the first row of the graph, representing the number of nodes in the graph.
5. Next, two functions are defined: floyd_recursive and floyd, implementing the recursive version of floyd's algorithm.
6. The floyd_recursive function updates the distance matrix by finding the shortest path between two nodes. It uses a recursive approach to iterate through all possible intermediate nodes and update the distances accordingly.
7. The floyd function is the main implementation of floyd's algorithm. It iterates over all combinations of intermediate, start_node, and end_node values using itertools.product and calls the floyd_recursive function to update the distance matrix.
8. The floydtestcase class is defined, which inherits from unittest.testcase. It contains a single test method, test_floyd_algorithm, that verifies the correctness of the floyd algorithm.
9. In the test_floyd_algorithm method, a copy of the graph is created, and the floyd algorithm is applied to the copy. The expected result is defined as the matrix representing the shortest distances between all pairs of nodes.
10. The self.assertequal assertion is used to compare the obtained distance matrix with the expected result.
11. Finally, the unittest.main() function is called to run the test case.

In summary, this unit test verifies the correctness of the floyd algorithm by comparing the obtained distances with the expected distances for a given graph. It ensures that the algorithm produces the desired results and helps in detecting any regressions or errors.

```
#unittest_forrecursiveversion

import sys
import itertools
import unittest

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_recursive(distance, intermediate, start_node, end_node):
    """
    Recursive implementation of Floyd's algorithm
    """
    if start_node == end_node:
        distance[start_node][end_node] = 0
        return

    distance[start_node][end_node] = min(
        distance[start_node][end_node],
        distance[start_node][intermediate] + distance[intermediate][end_node]
    )

    if intermediate < MAX_LENGTH - 1:
        floyd_recursive(distance, intermediate + 1, start_node, end_node)


def floyd(distance):
    """
    Implementation of Floyd's algorithm
    """
    for intermediate, start_node, end_node in itertools.product(range(MAX_LENGTH),
range(MAX_LENGTH), range(MAX_LENGTH)):
        floyd_recursive(distance, intermediate, start_node, end_node)


class FloydTestCase(unittest.TestCase):

    def test_floyd_algorithm(self):
        distance = [row[:] for row in graph]  # Create a copy of the original graph
        floyd(distance)
        expected_result = [
            [0, 7, 12, 8],
            [NO_PATH, 0, 5, 7],
            [NO_PATH, NO_PATH, 0, 2],
            [NO_PATH, NO_PATH, NO_PATH, 0]
        ]
        self.assertEqual(distance, expected_result)


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

Figure 1. Floyd's Algorithm Unit Test(Developed and adapted by the author)

2.2 Floyd Warshall algorithm (recursion) - Performance Test

It is valuable to evaluate metrics, such as program speed, responsiveness to user input, and memory efficiency. These metrics fall under the category of Performance Testing. In the Python programming language, there are several excellent tools available for conducting manual Performance Testing.

One such tool is the timeit function, which enables the evaluation of function performance. It intelligently determines the appropriate number of times to execute a piece of code in order to calculate a reliable average execution time. Notably, slower functions typically require fewer iterations compared to faster ones.

By utilizing the timeit function, developers can assess the performance of their functions and measure factors like execution time. This enables the identification of bottlenecks, optimization opportunities, and the overall efficiency of the program. Performance testing provides insights into system resource consumption, responsiveness, and speed, contributing to the development of optimized and high-performing software solutions.

The algorithm on figure 3, showcases the performance test using timeit function which is explained as follow:

1. The code starts by importing the necessary modules: sys, itertools, and timeit. These modules enable system-specific operations, iteration tools, and performance measurement, respectively.
2. The no_path constant is set to sys.maxsize, representing infinity or the absence of a direct path between nodes in the graph.
3. The graph variable is a 2d list that represents the weighted graph on which the algorithm will be executed. It contains a matrix of weights indicating the edges between nodes.
4. The max_length variable is set to the length of the first row of the graph, which represents the number of nodes in the graph.
5. The floyd_recursive function implements the recursive version of floyd's algorithm. It updates the distance matrix by finding the shortest path between two nodes. The function makes recursive calls and uses dynamic programming to compute the shortest distances.
6. The floyd function is the main implementation of floyd's algorithm. It iterates over all combinations of intermediate, start_node, and end_node values using itertools.product and calls the floyd_recursive function to update the distance matrix.

7. The test_performance function is defined to measure the performance of the algorithm. It creates a copy of the original graph, then uses timeit.timeit to measure the execution time of the floyd function. The function is executed 10,000 times to obtain an accurate average execution time.
8. Finally, the test_performance function is called within the if __name__ == '__main__' block to run the performance test.

In summary, the code measures the performance of the recursive version of floyd's algorithm using the timeit module. It creates a copy of the graph, executes the algorithm multiple times, and calculates the average execution time. This performance testing provides insights into the efficiency of the algorithm, aiding in evaluating and optimizing its execution time for a given graph.

```
#performancetest_recursiveversion

import sys
import itertools
import timeit

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_recursive(distance, intermediate, start_node, end_node):
    """
    Recursive implementation of Floyd's algorithm
    """
    if start_node == end_node:
        distance[start_node][end_node] = 0
        return

    distance[start_node][end_node] = min(
        distance[start_node][end_node],
        distance[start_node][intermediate] + distance[intermediate][end_node]
    )

    if intermediate < MAX_LENGTH - 1:
        floyd_recursive(distance, intermediate + 1, start_node, end_node)


def floyd(distance):
    """
    Implementation of Floyd's algorithm
    """
    for intermediate, start_node, end_node in itertools.product(range(MAX_LENGTH),
range(MAX_LENGTH), range(MAX_LENGTH)):
        floyd_recursive(distance, intermediate, start_node, end_node)


# Performance test
def test_performance():
    distance = [row[:] for row in graph]  # Create a copy of the original graph

    # Measure the execution time of the floyd function - (function is executed 10000 times)
    execution_time = timeit.timeit(lambda: floyd(distance), number=10000)
    print(f"Execution time: {execution_time} seconds")


if __name__ == '__main__':
    test_performance()
()
```

Figure 3. Floyd's Algorithm Performance Test(Developed and adapted by the author)

Chapter 3. CONCLUSIONS

The report showcases the implementation of the recursive version of Floyd's algorithm, performs unit tests to verify its correctness, and conducts performance tests to evaluate its execution time, demonstrating the importance of software testing for ensuring the quality, reliability, and performance of software systems.

In addition, the results highlight the superiority of the imperative version in terms of speed, with significantly shorter execution times observed. It is important to acknowledge that the exact execution times may vary slightly across different runs due to factors like system load and hardware capabilities.

REFERENCES

Agarwal, B. and Baka, B. (2018) ‘Python Data Types and Structures’, in Hands-On Data Structures and Algorithms with Python. United Kingdom: Packt Publishing, Limited.

Lott, S.F. (2018) Functional python programming discover the power of functional programming, generator functions, lazy evaluation, the built-in itertools library, and monads, second edition / Steven F. Lott. 2nd ed. S.l: Packt Publishing.

Pajankar, A. (2017) Python unit test automation : practical techniques for Python developers and testers / Ashwin Pajankar. United States: Apress.

Pajankar, A. (2021) Python unit test automation : automate, organize, and execute unit tests in Python / Ashwin Pajankar. 2nd ed. New York, NY: Apress. Available at: https://doi.org/10.1007/978-1-4842-7854-3.

Rubio-Sánchez, M. (2018) Introduction to recursive programming / Manuel Rubio-Sánchez. 1st edition. Boca Raton, FL: CRC Press, Taylor & Francis Group.

Sale, D. (2014) Testing Python : applying unit testing, TDD, BDD, and acceptance testing / David Sale. Chichester, England: Wiley.

Toroslu, I.H. (2023) ‘The Floyd-Warshall all-pairs shortest paths algorithm for disconnected and very sparse graphs’, Software, practice & experience, 53(6), pp. 1287–1303. Available at: https://doi.org/10.1002/spe.3188.