Project 3 Objective 2: Investigation of Dijkstra's and Bellman-Ford Implementations for the Shortest Path Algorithm for a graphical network between the Middelburg railway station and UCR

Shaivan Bhagat

SCI COMP 302 Algorithms and Data Structures

This is all my own work. I have not knowingly allowed others to copy my work. This work has not been submitted for assessment in any other context.

1. Description

For this project, I am investigating the graph algorithm with the Shortest Path (SP) of the street networks between the Middelburg railway station and University College Roosevelt. For the Shortest Path, the chosen implementations are Dijkstra and Bellman Ford. They are both used to find the shortest paths in a weighted graph. Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a source node to all other nodes in the graph. It iteratively selects the node with the smallest tentative distance and updates the distances of its neighbors. Bellman-Ford algorithm works for graphs with both positive and negative weights. It iterates through all edges multiple times, relaxing the edges to find the shortest paths. It handles negative weights (unlike Dijkstra's) but can detect negative cycles in the graph. If there are no negative cycles, it provides the shortest paths.

2. Experimental Platform

The algorithms were run on a MacBook Air running on a M2 Chip. General specifications of the hardware are:

- Total Number of Cores: 8 (4 performance & 4 efficiency)
- Memory: 8 GB
- Chipset Model: Apple M2
- Type: CPU
- Bus: Built-In

The algorithms were run completely through Python. Python files were run via Spyder,

accessed through Anaconda-Navigator 2.4.2. The specific versions of the software are as follows:

- Spyder version: 5.4.3 (conda)
- Python version: 3.11.4 64-bit
- Qt version: 5.15.2
- PyQt5 version: 5.15.7
- Operating System: Darwin 22.5.0

Whilst the algorithms were being run, all cores were exploited, and CPU load was dedicated specifically towards enhancing the performance of the algorithms.

3. Experimental Components

a. Null Hypothesis

The implementation Dijkstra for the algorithm of the Shortest Path problem will perform better (time complexity) than the Bellman-Ford implementation for the same problem.

b. Time & Space Complexity

Number of runs for the two implementations: 10.

Big-Oh (Complexity):

Dijkstra's Algorithm: O((V + E) * log(V)) with a binary heap as the priority queue; when implemented with a binary heap as the priority queue, has a time complexity of O((V + E) * log(V)), where V is the number of vertices and E is the number of edges. This complexity arises from the need to update the priority queue and extract the minimum element during each iteration.

Bellman-Ford Algorithm: O(V * E) (worst case); where V is the number of vertices and E is the number of edges. The worst-case time complexity is often considered to be O(V * E) because the algorithm iterates over all edges for a maximum of V-1 passes. In practice, the algorithm often terminates earlier if no negative cycles are present.

Space Complexity: O(V) for both Dijkstra's and Bellman-Ford; primarily used for storing distances and predecessor information.

Time complexity in python was measured using the following pseudocode:

```
import time

start_time = time.time()
```

```
end_time = time.time()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time} seconds")
```

c. Experiment & Code

Independent Variable(s):

- Graph Structure (G): The structure of the graph, including nodes, edges, and their attributes. This is the input to the algorithm and is independent because you can choose or generate different graphs to analyze.
- Source and Target nodes: The nodes between which the shortest path is calculated.

Dependent Variables(s):

- Shortest Path: The sequence of nodes representing the shortest path from the source to the target. This is the main output of the algorithm, and it depends on the input graph structure and the selected source and target nodes.
- Elapsed Time: The time taken by the algorithm to calculate the shortest path. This time is dependent on factors such as the size and complexity of the graph.

Time was measured in microseconds (µs) as the amount taken to find and print the shortest possible path between the Middelburg railway station and UCR. This was then used as the basis for comparison of performance and complexity.

- In the following code, the nx.shortest_path function is used with the specified method set to "dijkstra." This function computes the shortest paths in the graph using Dijkstra's algorithm. The paths are then stored in the paths variable, and the elapsed time is measured for performance evaluation.
- The shortest_path function takes a graph (G), source and target nodes (source and target), and an optional weight parameter.
- The function calls nx.shortest_path with the specified method set to 'dijkstra'. This function computes the shortest paths in the graph using Dijkstra's algorithm.
- The calculated shortest path is stored in a variable and the elapsed time is measured for performance evaluation. The results, including the elapsed time and the shortest path, are printed.

/Users/shaivan/Downloads/Project3/Project3_ShaivanBhagat.py

jkl.py | Project3_ShaivanBhagat.py | QWE.py | xyz.py

```
    # Implementation of Djikstra's algorithm
    def has_path(G, source, target):
        try:
            nx.shortest_path(G, orig, dest)
        except nx.NetworkXNoPath:
            return False
        return True

    def shortest_path(G, source=None, target=None, weight=None, method="dijkstra"):
        if method not in ("dijkstra", "bellman-ford"):
            # so we don't need to check in each branch later
            raise ValueError(f"method not supported: {method}")
        method = "unweighted" if weight is None else method
        if source is None:
            if target is None:
                # Find paths between all pairs.
                if method == "unweighted":
                    paths = dict(nx.all_pairs_shortest_path(G))
                elif method == "dijkstra":
                    paths = dict(nx.all_pairs_dijkstra_path(G, weight=weight))
                else:  # method == 'bellman-ford':
                    paths = dict(nx.all_pairs_bellman_ford_path(G, weight=weight))
            else:
                # Find paths from all nodes co-accessible to the target.
                if G.is_directed():
                    G = G.reverse(copy=False)
                if method == "unweighted":
                    paths = nx.single_source_shortest_path(G, target)
                elif method == "dijkstra":
                    paths = nx.single_source_dijkstra_path(G, target, weight=weight)
                else:  # method == 'bellman-ford':
                    paths = nx.single_source_bellman_ford_path(G, target, weight=weight)
                # Now flip the paths so they go from a source to the target.
                for target in paths:
                    paths[target] = list(reversed(paths[target]))
        else:
            if target is None:
                # Find paths to all nodes accessible from the source.
                if method == "unweighted":
                    paths = nx.single_source_shortest_path(G, source)
                elif method == "dijkstra":
                    paths = nx.single_source_dijkstra_path(G, source, weight=weight)
                else:  # method == 'bellman-ford':
                    paths = nx.single_source_bellman_ford_path(G, source, weight=weight)
            else:
                # Find shortest source-target path.
                if method == "unweighted":
                    paths = nx.bidirectional_shortest_path(G, source, target)
                elif method == "dijkstra":
                    _, paths = nx.bidirectional_dijkstra(G, source, target, weight)
                else:  # method == 'bellman-ford':
                    paths = nx.bellman_ford_path(G, source, target, weight)
        return paths

    # Timing the shortest path calculation
    start_time = time.time()
    shortest_path(G, source=orig, target=dest, weight="travel_time", method="dijkstra")
    elapsed_time = time.time() - start_time
    print(f"Shortest path time Djikstra: {elapsed_time} seconds")
```

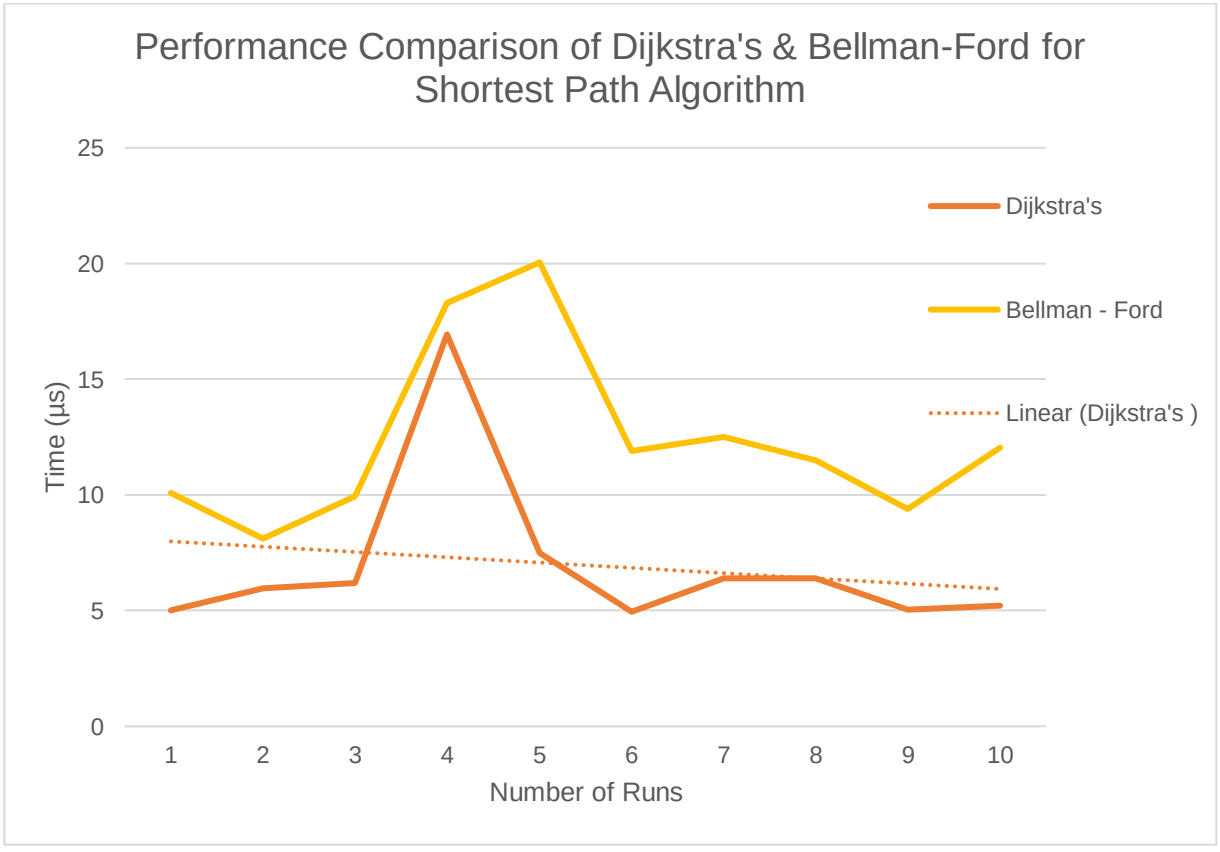
The bellman_ford_shortest_path function is defined, which uses the NetworkX function nx.shortest_path with the method='bellman-ford' parameter to find the shortest path between the specified source and target nodes in the graph G. The timing of the Bellman-Ford algorithm is measured by recording the start time (start_time_bellman_ford) before

calling the function and then calculating the elapsed time (elapsed_time_bellman_ford) after the function has been executed. The shortest path and the time taken by the Bellman-Ford algorithm are printed.

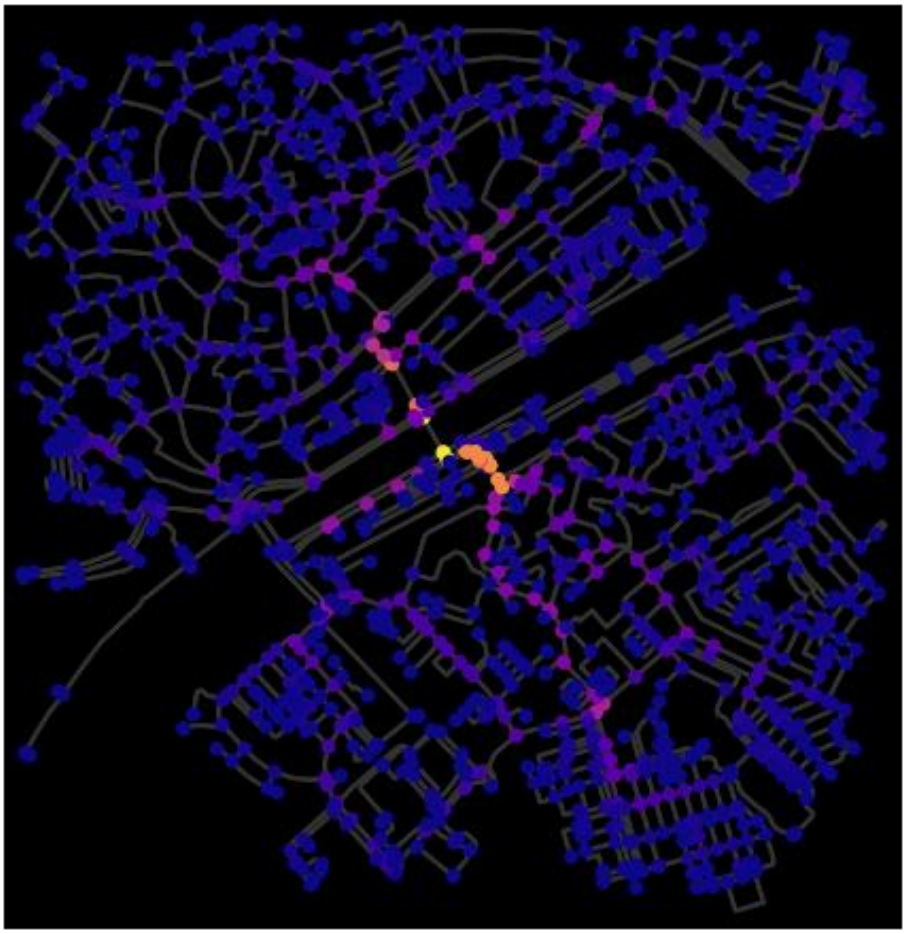
```
152
153     # Implementation of Bellman-Ford algorithm
154     def bellman_ford_shortest_path(G, source, target, weight=None):
155         return nx.shortest_path(G, source=source, target=target, weight=weight, method='bellman-ford')
156
157     # Timing the shortest path calculation using Bellman-Ford algorithm
158     start_time_bellman_ford = time.time()
159     bellman_ford_path = bellman_ford_shortest_path(G, source=orig, target=dest, weight='travel_time')
160     elapsed_time_bellman_ford = time.time() - start_time_bellman_ford
161     print(f"Bellman-Ford Shortest path time: {elapsed_time_bellman_ford} seconds")
162     print("Bellman-Ford Shortest path:", bellman_ford_path)
163
```

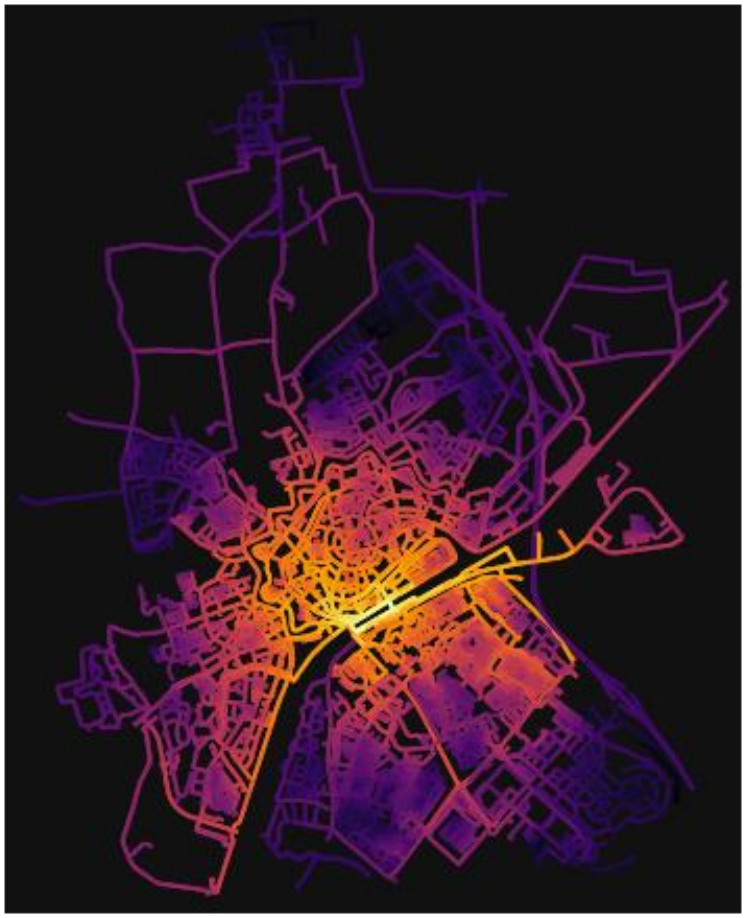
```
def shortest_path(G, source=None, target=None, weight=None, method="bellman-ford"):
    if method not in ("dijkstra", "bellman-ford"):
        # so we don't need to check in each branch later
        raise ValueError(f"method not supported: {method}")
    method = "unweighted" if weight is None else method
```

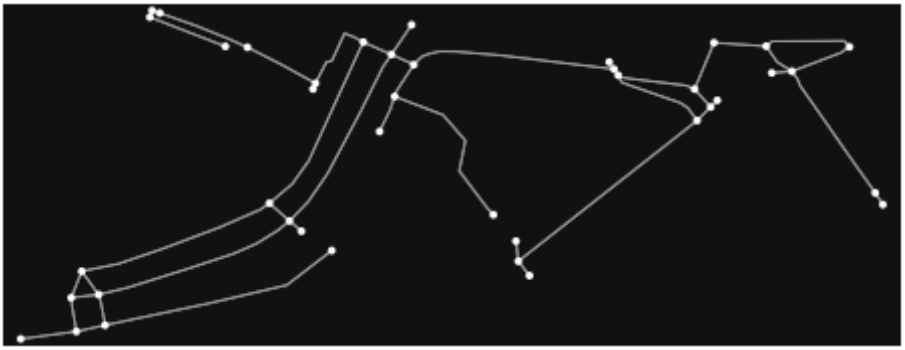
Results



Run	Dijkstra's	Bellman - Ford
1	5,0067	10,0804
2	5,9604	8,1106
3	6,1988	9,948
4	16,9380	18,29
5	7,4997	20,048
6	4,9508	11,89579
7	6,408	12,4897
8	6,4038	11,4809
9	5,048	9,4039
10	5,209	12,049







d. Static Analysis & Refactoring

Through Source > Run code analysis, static analysis can be performed for the python file using pylint.

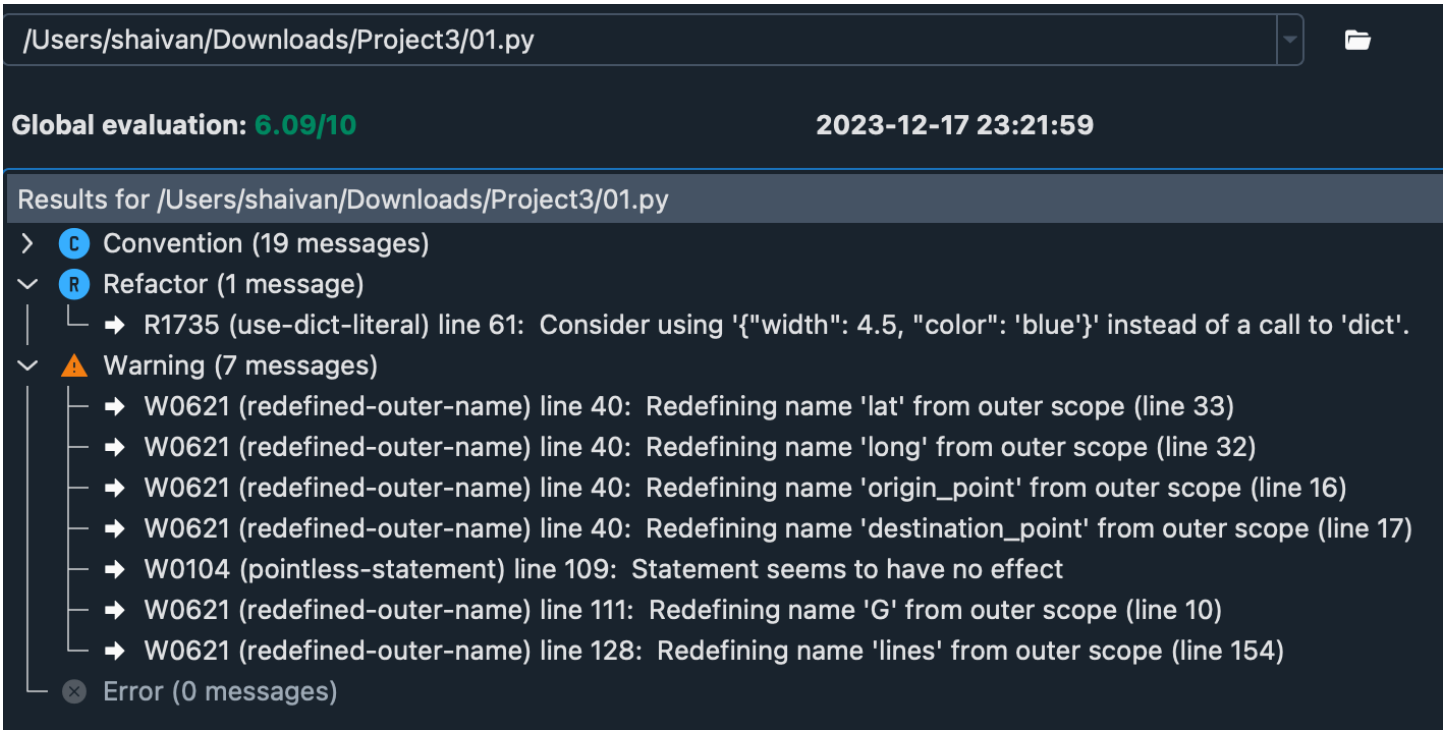
Most of the errors are convention errors which requires to name the attributes, variables, and arguments according to the python naming convention. The refractor error can be ignored. However, the error message can lead to issues with implementation.

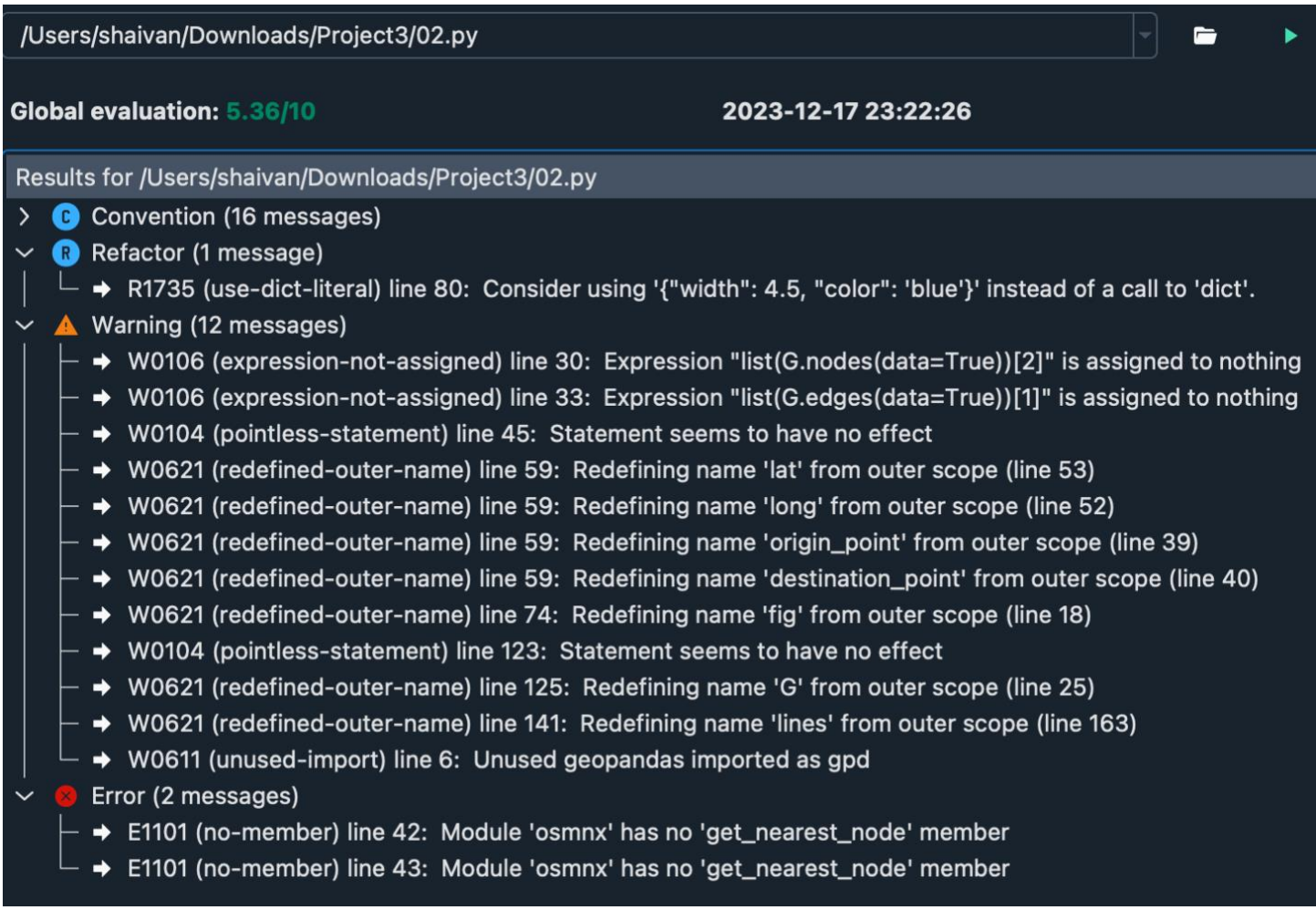
Refactoring would involve restructuring the code to enhance its readability, maintainability, and/or performance without modifying its behavior. Redundant code could be made more modular.

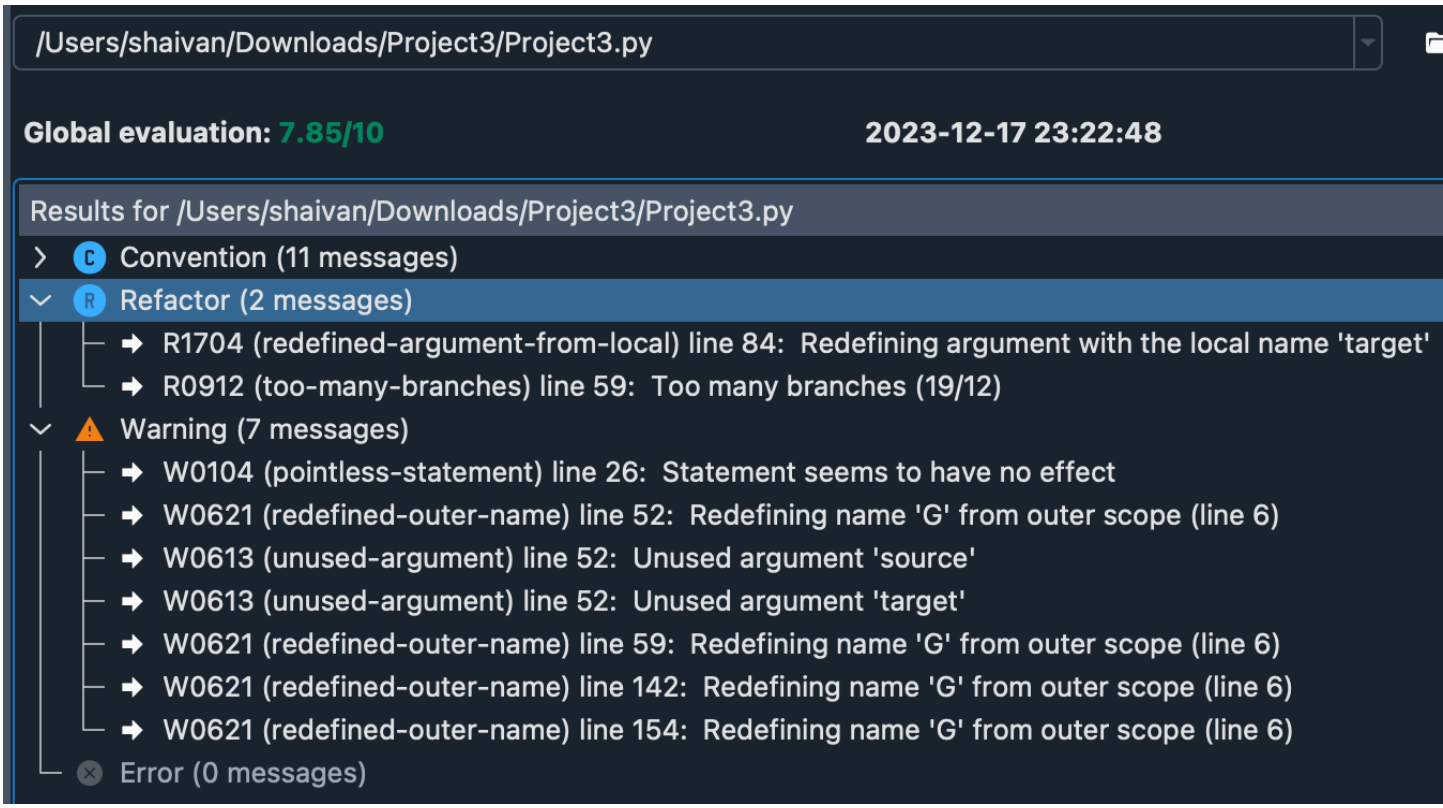
Techniques that could be applied to refactor code could possibly include simplifying conditional statements, modularization, adhering to naming conventions and PEP 8 standards, reducing complexity, and improving readability.

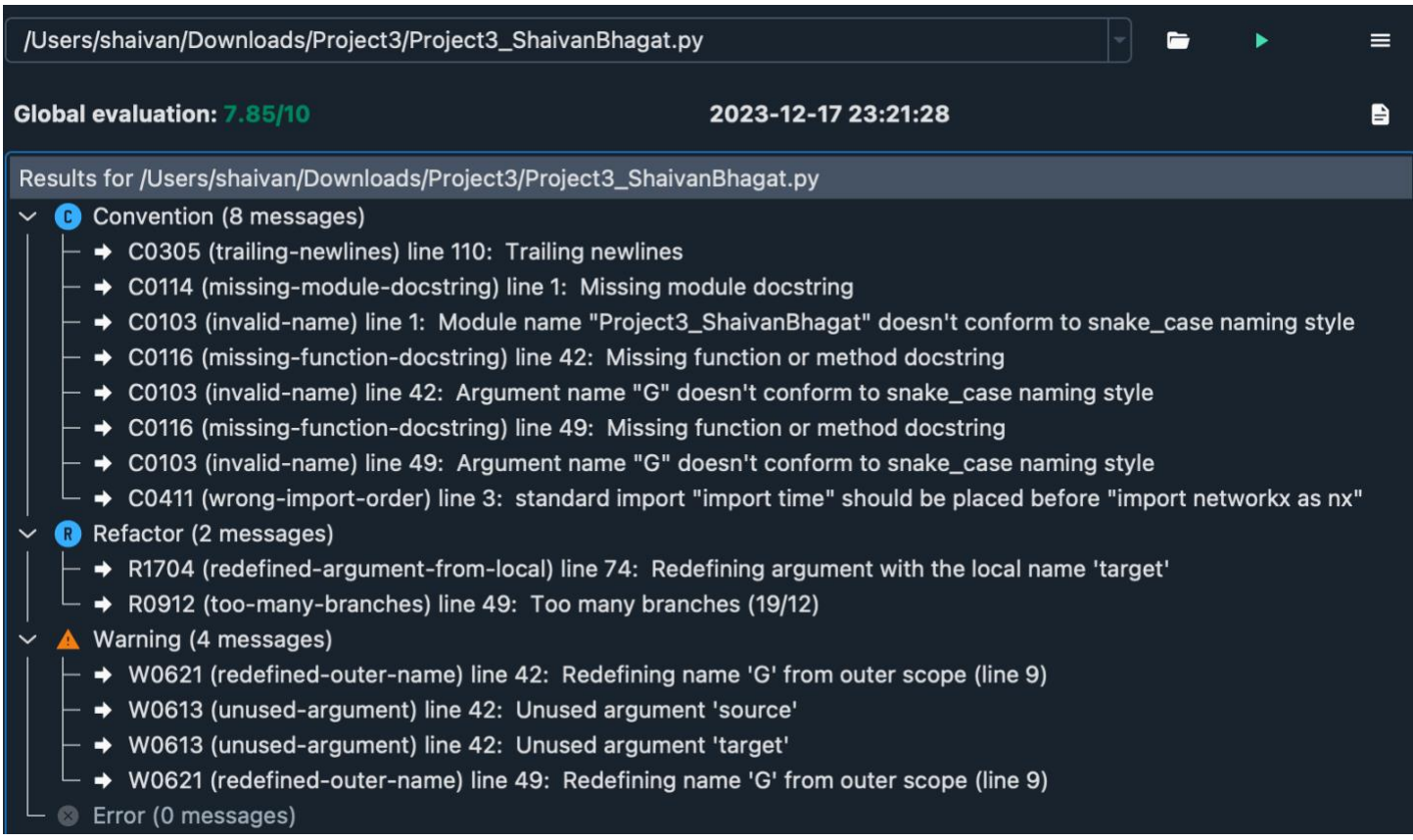
- Error handling: ‘has_path’ function is defined but not used. Removing this is suggested.
- Code duplication: The graph is being saved in different formats, which can be avoided by using only one specific format to avoid duplication.
- Time measurement: instead of manually converting the time taken for the implementations to return the answers, the ‘timeit’ module can and should be used for more accurate and convenient timing measurements
- Function parameter: The shortest_path function takes parameters named source and target, but the function body uses orig and dest. It would be more consistent and clear to use source and target throughout.
- Although there are errors present in the code, as seen from the sensitivity analysis, most errors can be neglected. The only one which needs to be taken into consideration is for the second file of code with 2 errors where the module does not recognize the required function.

However, a solution for that is presented below as well.


/Users/shaivan/Downloads/Project3/01.py
Global evaluation: 6.09/10
2023-12-17 23:21:59
Results for /Users/shaivan/Downloads/Project3/01.py
Convention (19 messages)
Refactor (1 message)
R1735 (use-dict-literal) line 61: Consider using '{"width": 4.5, "color": 'blue'}' instead of a call to 'dict'.
Warning (7 messages)
W0621 (redefined-outer-name) line 40: Redefining name 'lat' from outer scope (line 33)
W0621 (redefined-outer-name) line 40: Redefining name 'long' from outer scope (line 32)
W0621 (redefined-outer-name) line 40: Redefining name 'origin_point' from outer scope (line 16)
W0621 (redefined-outer-name) line 40: Redefining name 'destination_point' from outer scope (line 17)
W0104 (pointless-statement) line 109: Statement seems to have no effect
W0621 (redefined-outer-name) line 111: Redefining name 'G' from outer scope (line 10)
W0621 (redefined-outer-name) line 128: Redefining name 'lines' from outer scope (line 154)
Error (0 messages)


/Users/shaivan/Downloads/Project3/02.py
Global evaluation: 5.36/10
2023-12-17 23:22:26
Results for /Users/shaivan/Downloads/Project3/02.py
Convention (16 messages)
Refactor (1 message)
R1735 (use-dict-literal) line 80: Consider using '{"width": 4.5, "color": 'blue'}' instead of a call to 'dict'.
Warning (12 messages)
W0106 (expression-not-assigned) line 30: Expression "list(G.nodes(data=True))[2]" is assigned to nothing
W0106 (expression-not-assigned) line 33: Expression "list(G.edges(data=True))[1]" is assigned to nothing
W0104 (pointless-statement) line 45: Statement seems to have no effect
W0621 (redefined-outer-name) line 59: Redefining name 'lat' from outer scope (line 53)
W0621 (redefined-outer-name) line 59: Redefining name 'long' from outer scope (line 52)
W0621 (redefined-outer-name) line 59: Redefining name 'origin_point' from outer scope (line 39)
W0621 (redefined-outer-name) line 59: Redefining name 'destination_point' from outer scope (line 40)
W0621 (redefined-outer-name) line 74: Redefining name 'fig' from outer scope (line 18)
W0104 (pointless-statement) line 123: Statement seems to have no effect
W0621 (redefined-outer-name) line 125: Redefining name 'G' from outer scope (line 25)
W0621 (redefined-outer-name) line 141: Redefining name 'lines' from outer scope (line 163)
W0611 (unused-import) line 6: Unused geopandas imported as gpd
Error (2 messages)
E1101 (no-member) line 42: Module 'osmnx' has no 'get_nearest_node' member
E1101 (no-member) line 43: Module 'osmnx' has no 'get_nearest_node' member


/Users/shaivan/Downloads/Project3/Project3.py
Global evaluation: 7.85/10
2023-12-17 23:22:48
Results for /Users/shaivan/Downloads/Project3/Project3.py
Convention (11 messages)
Refactor (2 messages)
R1704 (redefined-argument-from-local) line 84: Redefining argument with the local name 'target'
R0912 (too-many-branches) line 59: Too many branches (19/12)
Warning (7 messages)
W0104 (pointless-statement) line 26: Statement seems to have no effect
W0621 (redefined-outer-name) line 52: Redefining name 'G' from outer scope (line 6)
W0613 (unused-argument) line 52: Unused argument 'source'
W0613 (unused-argument) line 52: Unused argument 'target'
W0621 (redefined-outer-name) line 59: Redefining name 'G' from outer scope (line 6)
W0621 (redefined-outer-name) line 142: Redefining name 'G' from outer scope (line 6)
W0621 (redefined-outer-name) line 154: Redefining name 'G' from outer scope (line 6)
Error (0 messages)


/Users/shaivan/Downloads/Project3/Project3_ShaivanBhagat.py
Global evaluation: 7.85/10
2023-12-17 23:21:28
Results for /Users/shaivan/Downloads/Project3/Project3_ShaivanBhagat.py
Convention (8 messages)
C0305 (trailing-newlines) line 110: Trailing newlines
C0114 (missing-module-docstring) line 1: Missing module docstring
C0103 (invalid-name) line 1: Module name "Project3_ShaivanBhagat" doesn't conform to snake_case naming style
C0116 (missing-function-docstring) line 42: Missing function or method docstring
C0103 (invalid-name) line 42: Argument name "G" doesn't conform to snake_case naming style
C0116 (missing-function-docstring) line 49: Missing function or method docstring
C0103 (invalid-name) line 49: Argument name "G" doesn't conform to snake_case naming style
C0411 (wrong-import-order) line 3: standard import "import time" should be placed before "import networkx as nx"
Refactor (2 messages)
R1704 (redefined-argument-from-local) line 74: Redefining argument with the local name 'target'
R0912 (too-many-branches) line 49: Too many branches (19/12)
Warning (4 messages)
W0621 (redefined-outer-name) line 42: Redefining name 'G' from outer scope (line 9)
W0613 (unused-argument) line 42: Unused argument 'source'
W0613 (unused-argument) line 42: Unused argument 'target'
W0621 (redefined-outer-name) line 49: Redefining name 'G' from outer scope (line 9)
Error (0 messages)

.

```
origin_point = (51.495446, 3.618127)
destination_point = (51.499318, 3.610658)

origin_node = ox.distance.nearest_nodes(G, X=origin_point[1], Y=origin_point[0])
destination_node = ox.distance.nearest_nodes(G, X=destination_point[1], Y=destination_point[0])
```

4. Conclusion

Based on the results that I have obtained; I can verify the null hypothesis and conclude that Dijkstra's algorithm for the Shortest Path (SP) problem performs better compared to the Bellman-Ford implementation. Sensitivity and static code analysis has been presented as well, along with some possible options for code refactoring. Results along with the graph show that a shortest path between the Middelburg railway station and UCR is indeed present and can be calculated using the two implementation algorithms used in this case. Techniques like edge contraction, node ordering, and other preprocessing steps can be applied to simplify the graph structure and improve algorithm efficiency. For scenarios where the graph is dynamic (edges or nodes change over time), algorithms designed to handle dynamic graphs efficiently can be beneficial. On current hardware, leveraging GPU or specialized hardware can accelerate graph algorithms. Although here only the shortest path (SP) was considered, utilizing Minimum Spanning Tree (MST) to solve similar problems and to create an optimal graph network between the Middelburg railway station and UCR should be considered.

References

- https://apurv.page/plotthepath.html
- https://osmnx.readthedocs.io/en/stable/
- https://geoffboeing.com/2016/11/osmnx-python-street-networks/
- https://github.com/gboeing/osmnx-examples/blob/main/notebooks/00-osmnx-features-demo.ipynb
- https://networkx.org/documentation/stable//auto_examples/geospatial/plot_osmnx.html#sphx-glr-download-auto-examples-geospatial-plot-osmnx-py
- https://networkx.org/
- https://www.pylint.org/