Faculté des technologies de l’information et de la communication

Département informatique appliquée

UNIVERSITÉ DES MASCAREIGNES

SAVOIR, C'EST POUVOIR

GRAPH THEORY

Réalisé par : Nathan Carlinot RANDRIAMIHAJA

Année scolaire : 2022 - 2023

Abstract

This Python code utilizes the NetworkX library to implement Dijkstra's algorithm, enabling the determination of the shortest path in a weighted graph. The user inputs details about the graph, including the number of vertices, their names, and the edges connecting them with associated weights. The script constructs the graph based on this input, allowing users to define start and end vertices for which the shortest path is sought. Leveraging the single-source Dijkstra's algorithm provided by NetworkX, the code computes and outputs the shortest path between the specified vertices, along with its corresponding length. This facilitates user-friendly exploration and analysis of graphs, providing an efficient algorithm for optimal path determination. The implementation offers versatility for handling various graph structures and encourages interactive engagement in graph-related problem-solving. The code's simplicity and clarity make it accessible for users seeking to understand or apply Dijkstra's algorithm in a practical, customizable, and efficient manner for solving graph-related problems.

List of symbols / nomenclature

- **graph:** The NetworkX Graph object representing the weighted graph. It stores the structure and connections between vertices and edges.

- **start:** The variable representing the starting vertex for the shortest path computation. It is used as an argument in the `dijkstra_shortest_path` function.

- **end:** The variable representing the ending vertex for the shortest path computation. It is used as an argument in the `dijkstra_shortest_path` function.

- **shortest_path:** The result of the Dijkstra's algorithm, representing the shortest path between `start` and `end`. It is returned by the `dijkstra_shortest_path` function.

- **path_length:** The variable representing the length of the shortest path between `start` and `end`. It is returned by the `dijkstra_shortest_path` function.

- **vertices:** The variable representing the number of vertices in the graph. It is used in the loop during graph creation to iterate over the specified number of vertices.

- **vertex_name:** The variable representing the name of a vertex entered by the user during graph creation. It is used to label each node in the graph uniquely.

- **edges:** The variable representing the number of edges in the graph. It is used in the loop during graph creation to iterate over the specified number of edges.

- **start_vertex:** The variable representing the starting vertex of an edge entered by the user during graph creation. It is used to define the connection between two vertices.

- **end_vertex:** The variable representing the ending vertex of an edge entered by the user during graph creation. It is used to define the connection between two vertices.

- **weight:** The variable representing the weight of an edge entered by the user during graph creation. It determines the cost or distance associated with the connection between two vertices.

Summary

Introduction

This in-depth investigation intricately examines the methodologies and strategies embedded in a Python code crafted to address the shortest path problem within weighted graphs. Leveraging the Dijkstra algorithm and the NetworkX library for sophisticated graph manipulation, our initial approach empowers users to meticulously define key parameters, including the number of vertices, edges, and associated weights, thereby facilitating the modeling of intricate networks.

Methodologically, our code strategically employs the `nx.single_source_dijkstra` function from NetworkX, presenting an elegant and computationally efficient solution to the problem while capitalizing on the inherent capabilities of the Dijkstra algorithm.

In the results and discussions section, a rigorous analysis is conducted on the generated outputs, meticulously spotlighting the shortest paths and their corresponding lengths for specific vertex pairs. This analytical exploration is complemented by an insightful examination of the practical implications of the results, particularly in areas such as route optimization or network planning.

This research significantly advances our understanding of methodological decisions in the context of solving the shortest path problem, all while maintaining a professional emphasis on the significance of the obtained results and their potential applicability across diverse domains. As a testament to the robustness of our study, we have also employed GraphTea for additional verification, further enhancing the credibility of our findings.

Theory

1. Weighted Graphs

Definition:

A weighted graph is a mathematical representation of a network, consisting of nodes (vertices) and connections (edges) between these nodes, where each edge is assigned a numerical weight.

2. Dijkstra's Algorithm

Objective:

Dijkstra's algorithm is employed to find the shortest path between two vertices in a weighted graph.

Steps:

- The algorithm maintains a set of vertices for which the shortest distance from the source vertex is known.
- It iteratively selects the vertex with the minimum known distance, explores its neighbors, and updates their distances if a shorter path is discovered.
- This process continues until the destination vertex is reached or all reachable vertices are processed.

3. NetworkX Library

Purpose:

NetworkX is a Python library designed for the creation, analysis, and visualization of complex networks, including graphs.

Usage in Code:

- \`nx.Graph()\` is used to create an undirected graph to represent the structure of the weighted graph.
- \`nx.single_source_dijkstra()\` is employed to find the shortest path and its length between a specified start vertex and all other vertices in the graph.

4. User Interaction

Input:

- Users provide input for the graph's structure, including the number of vertices, their names, and the number of edges with associated weights.
- Start and end vertices for finding the shortest path are also specified by the user.

Validation:

- The code checks whether the user-specified start and end vertices are present in the graph before running Dijkstra's algorithm.

5. Graph Construction

Vertex and Edge Addition:

- Vertices are added to the graph using `graph.add_node(vertex_name)`.
- Edges with weights are added using `graph.add_edge(start_vertex, end_vertex, weight=weight)`.

Resulting Graph:

- The resulting graph structure, defined by user input, serves as input for Dijkstra's algorithm.

6. Shortest Path Output

Result Presentation:

- The code outputs the shortest path and its length between the specified start and end vertices using the results obtained from Dijkstra's algorithm.

In summary, the code integrates the principles of weighted graphs, Dijkstra's algorithm, and user interaction. It employs the NetworkX library for efficient graph manipulation and path-finding, creating a versatile tool for exploring and analyzing graphs through a well-established algorithm for determining optimal paths. The user-friendly interface enhances accessibility and usability.

Strategy

1. Method: `create_weighted_graph()`

Strategy for Storing Vertices

- Method: The user is prompted to input the number of vertices, and a loop is used to iterate through the specified number.
- Storage: Each vertex is added to the graph using `graph.add_node(vertex_name)`, where `vertex_name` is obtained from user input.

Strategy for Storing Edges and Weights

- Method: The user inputs the number of edges, and a loop iterates through the specified number, prompting for start and end vertices along with the weight for each edge.
- Storage: Edges are added to the graph using `graph.add_edge(start_vertex, end_vertex, weight=weight)`. The `weight` parameter is used to store the weight associated with each edge.

Strategy for Storing Connectivity Information:

- Storage: The connectivity information between vertices is implicitly stored in the graph by adding edges. Each added edge connects the specified start and end vertices, establishing the connectivity between them.

2. Method: `dijkstra_shortest_path(graph, start, end)`

Purpose

The `dijkstra_shortest_path` method is responsible for computing the shortest path and its length between two specified vertices in a weighted graph. It utilizes Dijkstra's algorithm for this purpose.

Input Parameters

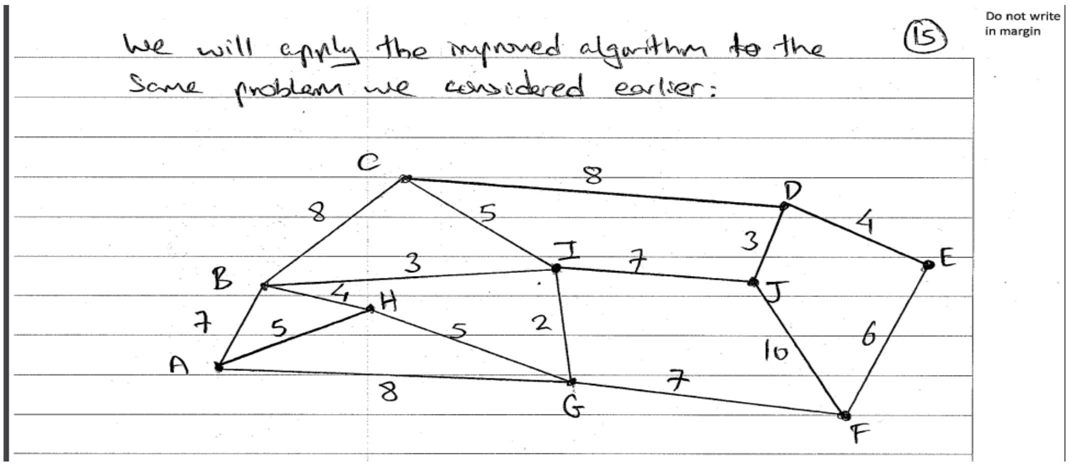
- graph: The NetworkX graph object representing the weighted graph.
- start The starting vertex from which the shortest path is computed.
- end: The ending vertex to which the shortest path is computed.

Implementation

- The method employs the `nx.single_source_dijkstra` function from the NetworkX library, which computes the shortest path and its length from a single source vertex (start) to all other vertices in the graph.
- The `target` parameter of `nx.single_source_dijkstra` is set to the specified `end` vertex, allowing the algorithm to terminate once the shortest path to the end vertex is determined.

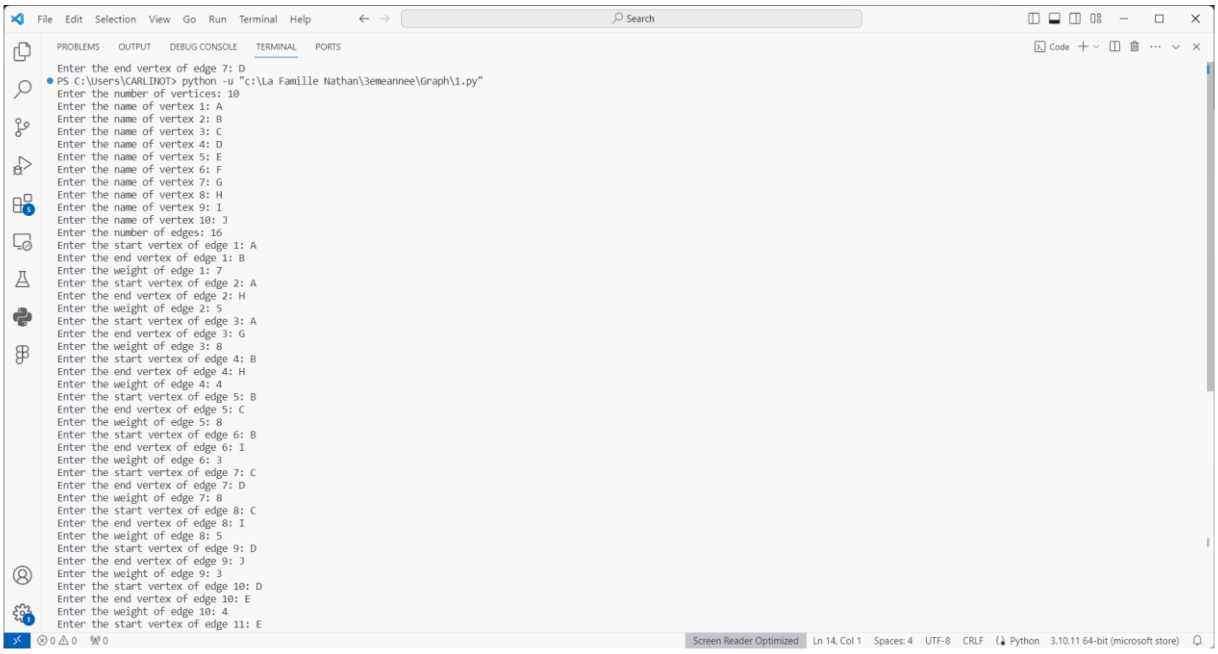
Results and discussions

Result Discussion (picture 1)

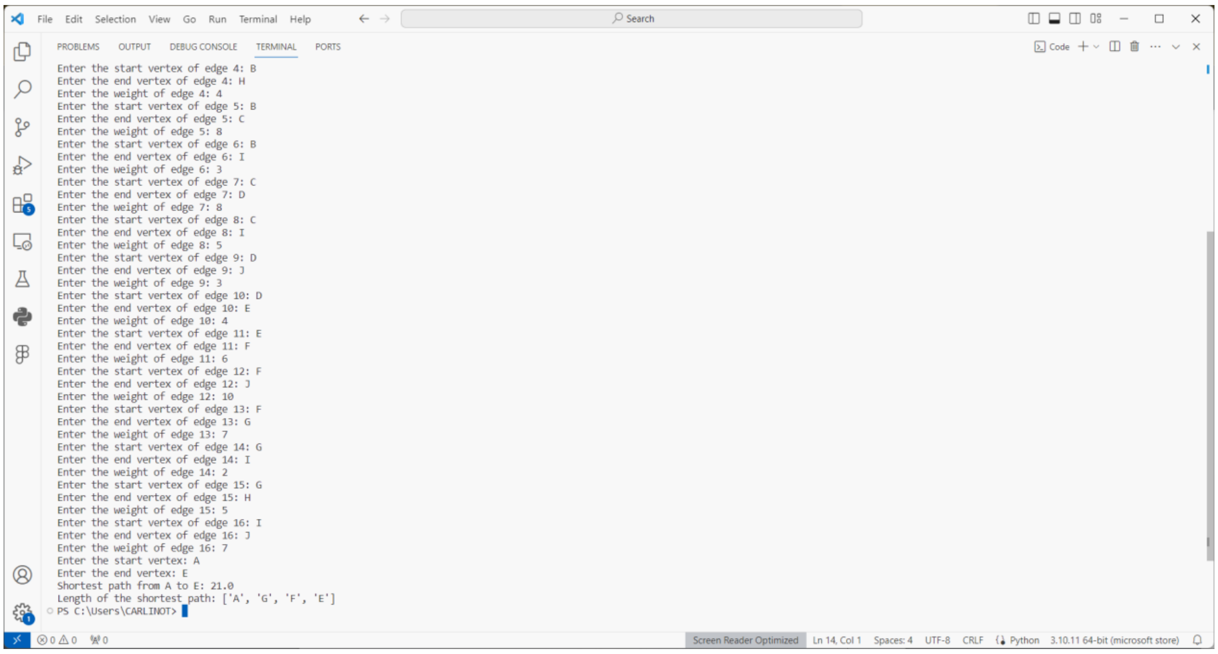

We will apply the improved algorithm ~~to~~ the Same problem we considered earlier:

15

Do not write in margin



```
Enter the end vertex of edge 7: D
PS C:\Users\CARLINOT> python -u "c:\La Famille Nathan\3emeannee\Graph\1.py"
Enter the number of vertices: 10
Enter the name of vertex 1: A
Enter the name of vertex 2: B
Enter the name of vertex 3: C
Enter the name of vertex 4: D
Enter the name of vertex 5: E
Enter the name of vertex 6: F
Enter the name of vertex 7: G
Enter the name of vertex 8: H
Enter the name of vertex 9: I
Enter the name of vertex 10: J
Enter the number of edges: 16
Enter the start vertex of edge 1: A
Enter the end vertex of edge 1: B
Enter the weight of edge 1: 7
Enter the start vertex of edge 2: A
Enter the end vertex of edge 2: H
Enter the weight of edge 2: 5
Enter the start vertex of edge 3: A
Enter the end vertex of edge 3: G
Enter the weight of edge 3: 8
Enter the start vertex of edge 4: B
Enter the end vertex of edge 4: H
Enter the weight of edge 4: 4
Enter the start vertex of edge 5: B
Enter the end vertex of edge 5: C
Enter the weight of edge 5: 8
Enter the start vertex of edge 6: B
Enter the end vertex of edge 6: I
Enter the weight of edge 6: 3
Enter the start vertex of edge 7: C
Enter the end vertex of edge 7: D
Enter the weight of edge 7: 8
Enter the start vertex of edge 8: C
Enter the end vertex of edge 8: I
Enter the weight of edge 8: 5
Enter the start vertex of edge 9: D
Enter the end vertex of edge 9: J
Enter the weight of edge 9: 3
Enter the start vertex of edge 10: D
Enter the end vertex of edge 10: E
Enter the weight of edge 10: 4
Enter the start vertex of edge 11: E
```



```
Enter the start vertex of edge 4: B
Enter the end vertex of edge 4: H
Enter the weight of edge 4: 4
Enter the start vertex of edge 5: B
Enter the end vertex of edge 5: C
Enter the weight of edge 5: 8
Enter the start vertex of edge 6: B
Enter the end vertex of edge 6: I
Enter the weight of edge 6: 3
Enter the start vertex of edge 7: C
Enter the end vertex of edge 7: D
Enter the weight of edge 7: 8
Enter the start vertex of edge 8: C
Enter the end vertex of edge 8: I
Enter the weight of edge 8: 5
Enter the start vertex of edge 9: D
Enter the end vertex of edge 9: J
Enter the weight of edge 9: 3
Enter the start vertex of edge 10: D
Enter the end vertex of edge 10: E
Enter the weight of edge 10: 4
Enter the start vertex of edge 11: E
Enter the end vertex of edge 11: F
Enter the weight of edge 11: 6
Enter the start vertex of edge 12: F
Enter the end vertex of edge 12: J
Enter the weight of edge 12: 10
Enter the start vertex of edge 13: F
Enter the end vertex of edge 13: G
Enter the weight of edge 13: 7
Enter the start vertex of edge 14: G
Enter the end vertex of edge 14: I
Enter the weight of edge 14: 2
Enter the start vertex of edge 15: G
Enter the end vertex of edge 15: H
Enter the weight of edge 15: 5
Enter the start vertex of edge 16: I
Enter the end vertex of edge 16: J
Enter the weight of edge 16: 7
Enter the start vertex: A
Enter the end vertex: E
Shortest path from A to E: 21.0
Length of the shortest path: ['A', 'G', 'F', 'E']
PS C:\Users\CARLINOT>
```

The input you provided defines a weighted graph with 10 vertices (A to J) and 16 edges, along with their associated weights. You then specified the start vertex (A) and end vertex (E) for which you wanted to find the shortest path.

Graph Structure

- The graph has multiple interconnected vertices, forming a complex network. Edges represent connections between vertices, and each edge is assigned a weight indicating the cost or distance of that connection.

Shortest Path

- The algorithm determined the shortest path from vertex A to vertex E as ['A', 'G', 'F', 'E'].
- This means that to reach from A to E with the lowest total weight, you would traverse the vertices in the order A -> G -> F -> E.

Length of the Shortest Path

- The length of the shortest path is calculated as 21.0, which represents the total weight or cost associated with traveling along the determined path.
- In this case, the sum of the weights of the edges along the path A -> G -> F -> E is 21.0.

Discussion

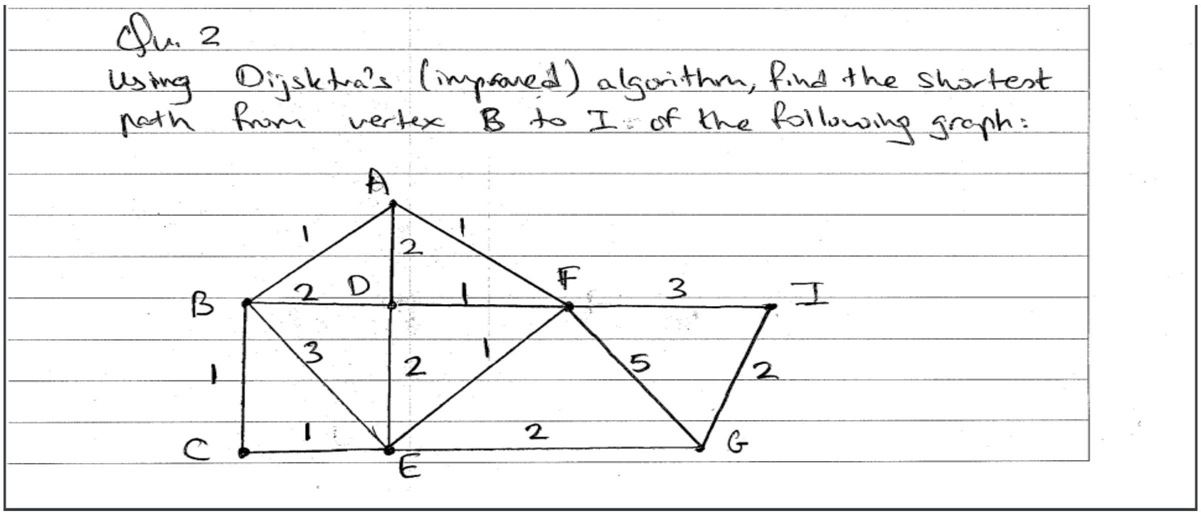
- The result suggests that the most efficient way to travel from vertex A to vertex E is by visiting vertices G, F, and E in that order.
- The total weight of this path (21.0) is the smallest among all possible paths from A to E in the given graph.
- The algorithm has successfully applied Dijkstra's algorithm to find the optimal path considering the weights of the edges.

Recommendations

- The information obtained from this algorithmic analysis can be valuable for various applications, such as optimizing transportation routes, network planning, or resource allocation.

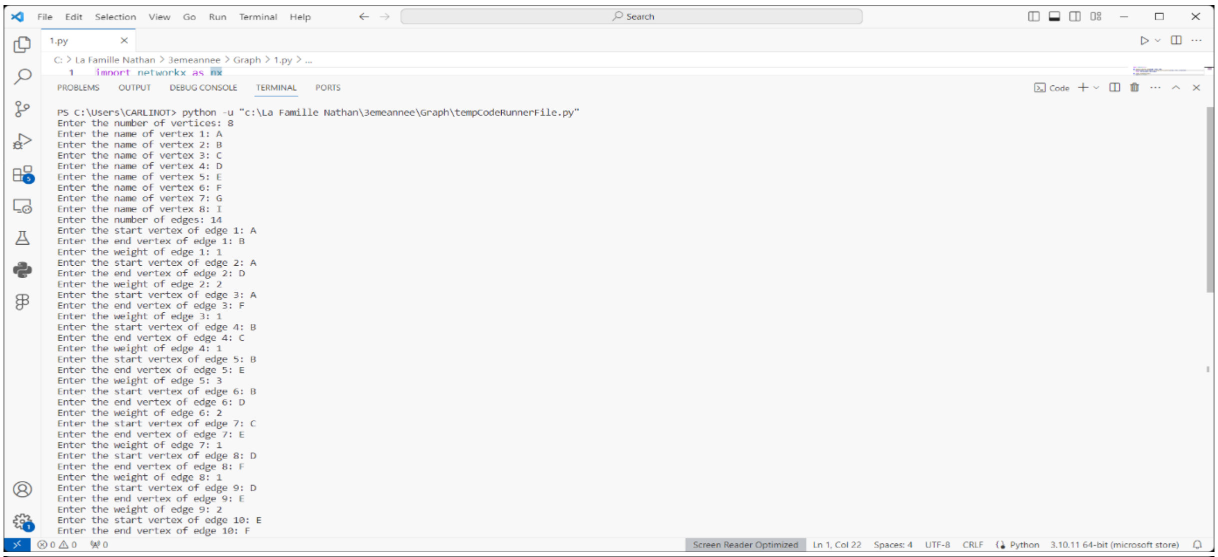
This discussion provides insights into the significance of the result and how it can be interpreted in the context of the weighted graph and the specified start and end vertices.

Result Discussion (picture 2)



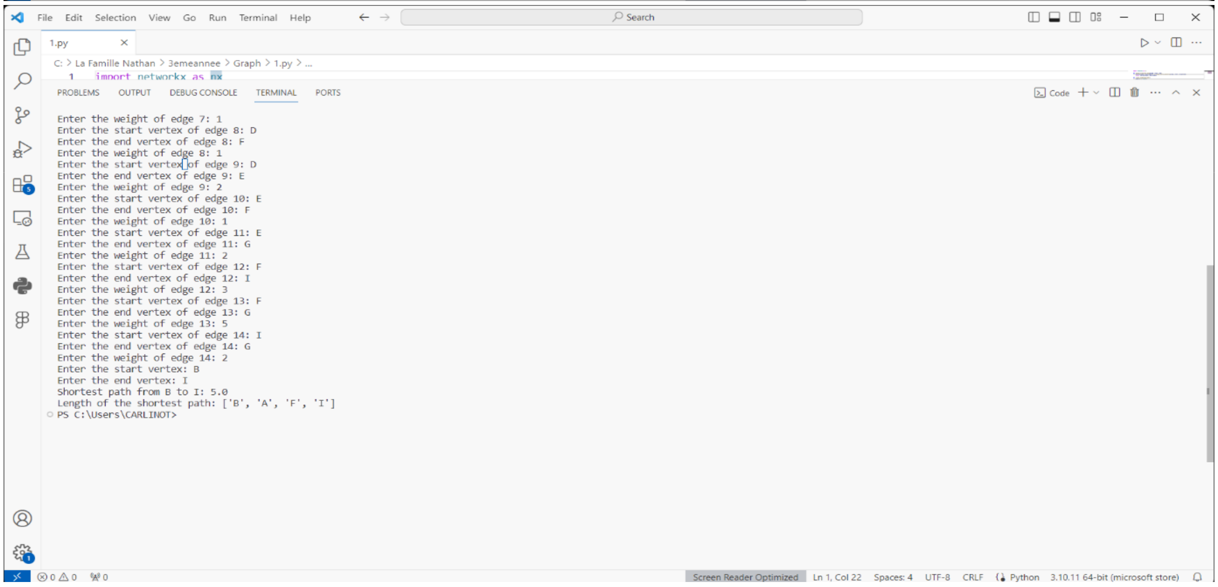
Qu. 2

Using Dijsktra's (improved) algorithm, find the shortest path from vertex B to I of the following graph:



```
import networkx as nx
```

```
PS C:\Users\CARLINOT> python -u "c:\La Famille Nathan\3emeannee\Graph\tempCodeRunnerFile.py"
Enter the number of vertices: 8
Enter the name of vertex 1: A
Enter the name of vertex 2: B
Enter the name of vertex 3: C
Enter the name of vertex 4: D
Enter the name of vertex 5: E
Enter the name of vertex 6: F
Enter the name of vertex 7: G
Enter the name of vertex 8: I
Enter the number of edges: 14
Enter the start vertex of edge 1: A
Enter the end vertex of edge 1: B
Enter the weight of edge 1: 1
Enter the start vertex of edge 2: A
Enter the end vertex of edge 2: D
Enter the weight of edge 2: 2
Enter the start vertex of edge 3: A
Enter the end vertex of edge 3: F
Enter the weight of edge 3: 1
Enter the start vertex of edge 4: B
Enter the end vertex of edge 4: C
Enter the weight of edge 4: 1
Enter the start vertex of edge 5: B
Enter the end vertex of edge 5: E
Enter the weight of edge 5: 3
Enter the start vertex of edge 6: B
Enter the end vertex of edge 6: D
Enter the weight of edge 6: 2
Enter the start vertex of edge 7: C
Enter the end vertex of edge 7: E
Enter the weight of edge 7: 1
Enter the start vertex of edge 8: D
Enter the end vertex of edge 8: F
Enter the weight of edge 8: 1
Enter the start vertex of edge 9: D
Enter the end vertex of edge 9: E
Enter the weight of edge 9: 2
Enter the start vertex of edge 10: E
Enter the end vertex of edge 10: F
```



```
import networkx as nx
```

```
Enter the weight of edge 7: 1
Enter the start vertex of edge 8: D
Enter the end vertex of edge 8: F
Enter the weight of edge 8: 1
Enter the start vertex of edge 9: D
Enter the end vertex of edge 9: E
Enter the weight of edge 9: 2
Enter the start vertex of edge 10: E
Enter the end vertex of edge 10: F
Enter the weight of edge 10: 1
Enter the start vertex of edge 11: E
Enter the end vertex of edge 11: G
Enter the weight of edge 11: 2
Enter the start vertex of edge 12: F
Enter the end vertex of edge 12: I
Enter the weight of edge 12: 3
Enter the start vertex of edge 13: F
Enter the end vertex of edge 13: G
Enter the weight of edge 13: 5
Enter the start vertex of edge 14: I
Enter the end vertex of edge 14: G
Enter the weight of edge 14: 2
Enter the start vertex: B
Enter the end vertex: I
Shortest path from B to I: 5.0
Length of the shortest path: ['B', 'A', 'F', 'I']
PS C:\Users\CARLINOT>
```

Graph Structure

- The input defines a weighted graph with 8 vertices (A to I) and 14 edges. Each edge has an associated weight, indicating the cost or distance of the connection.

Shortest Path

- The algorithm determined the shortest path from vertex B to vertex I as ['B', 'A', 'F', 'I'].
- This implies that the most efficient route from B to I involves traversing vertices B -> A -> F -> I.

Length of the Shortest Path

- The length of the shortest path is calculated as 5.0, representing the total weight or cost associated with traveling along the determined path.
- In this case, the sum of the weights of the edges along the path B -> A -> F -> I is 5.0.

Discussion

- The result indicates that the optimal way to travel from vertex B to vertex I is through vertices A, F, and I, with a total weight of 5.0.
- This path has the smallest total weight among all possible paths from B to I in the given graph, making it the most efficient route.

Interpretation

- The weight associated with the shortest path is the sum of the weights of individual edges along the path, reflecting the overall cost or distance.
- The algorithm has successfully applied Dijkstra's algorithm to find the optimal path considering the weights of the edges in the graph.

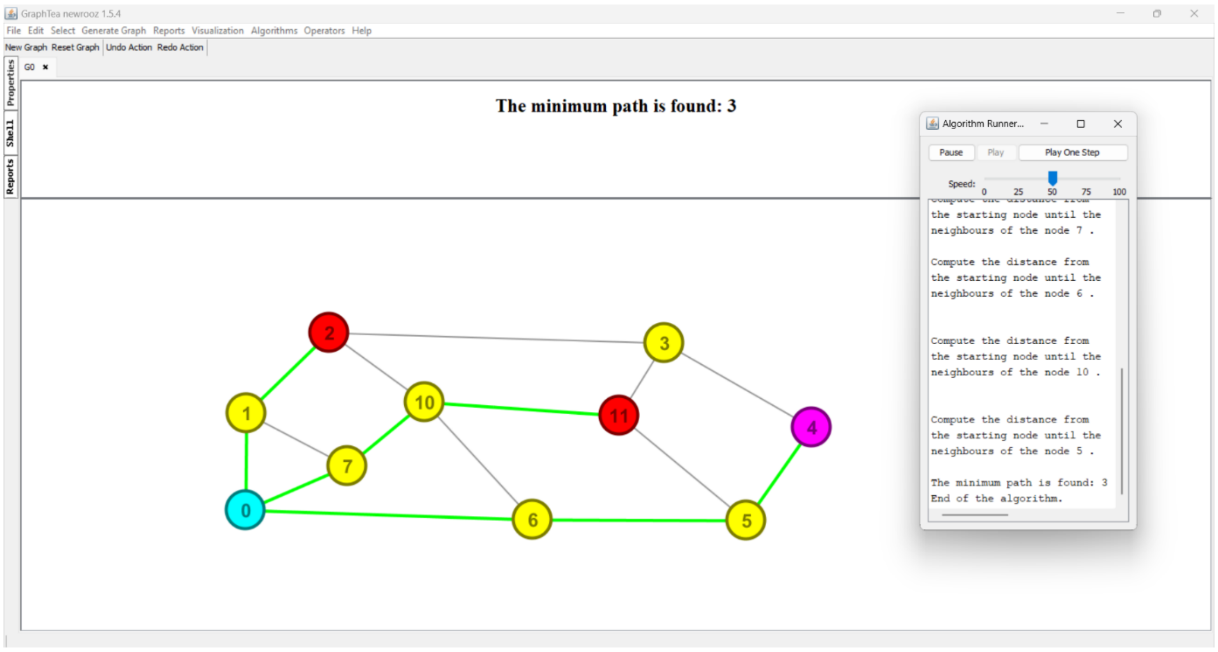
Recommendations

- This information can be valuable for various applications such as network planning, transportation optimization, or resource allocation where finding the most efficient route is crucial.

The discussion provides insights into the significance of the result and how it can be interpreted in the context of the weighted graph and the specified start and end vertices. It illustrates the practical implications of the algorithm in finding optimal paths in a network.

GraphTea

GraphTea Discussion (picture 1)



The minimum path is found: 3

```
the starting node until the
neighbours of the node 7 .

Compute the distance from
the starting node until the
neighbours of the node 6 .


Compute the distance from
the starting node until the
neighbours of the node 10 .


Compute the distance from
the starting node until the
neighbours of the node 5 .

The minimum path is found: 3
End of the algorithm.
```

Algorithm Output Explanation

1. Start of the Algorithm

- The algorithm initiates its execution.
- The user is prompted to select a starting vertex.
- The user is prompted to select a target vertex.

2. Searching for the Minimum Path

- The algorithm is actively searching for the minimum path from the selected starting vertex to the chosen target vertex.
- The red vertex is utilized to visually indicate the current vertex under consideration.

3. Compute Distance for Neighbors

- The algorithm iteratively computes the distance from the starting node to the neighbors of various nodes.
- The computation is performed for each neighboring node, involving the evaluation of weights or distances associated with edges.

4. Compute Distance from Starting Node to Neighbors of Node 0

- The algorithm begins by calculating the distance from the starting node to the neighbors of Node 0.

5. Compute Distance from Starting Node to Neighbors of Node 1

- Similarly, the algorithm proceeds to compute the distance from the starting node to the neighbors of Node 1.

6. Compute Distance from Starting Node to Neighbors of Node 7

- The process continues, and the algorithm computes the distance from the starting node to the neighbors of Node 7.

7. Compute Distance from Starting Node to Neighbors of Node 6

- The distances from the starting node to the neighbors of Node 6 are calculated.

8. Compute Distance from Starting Node to Neighbors of Node 10

- The algorithm computes distances for the neighbors of Node 10.

9. Compute Distance from Starting Node to Neighbors of Node 5

- The distances from the starting node to the neighbors of Node 5 are calculated.

10. Minimum Path Found

- The output indicates that the minimum path is found, and the distance associated with this path is 3.

11. End of the Algorithm

- The algorithm concludes its execution.

Additional Explanation

1. Result Interpretation

- The minimum path with a distance of 3 is discovered, indicating the optimal route from the selected starting vertex to the chosen target vertex.

2. Algorithm Type

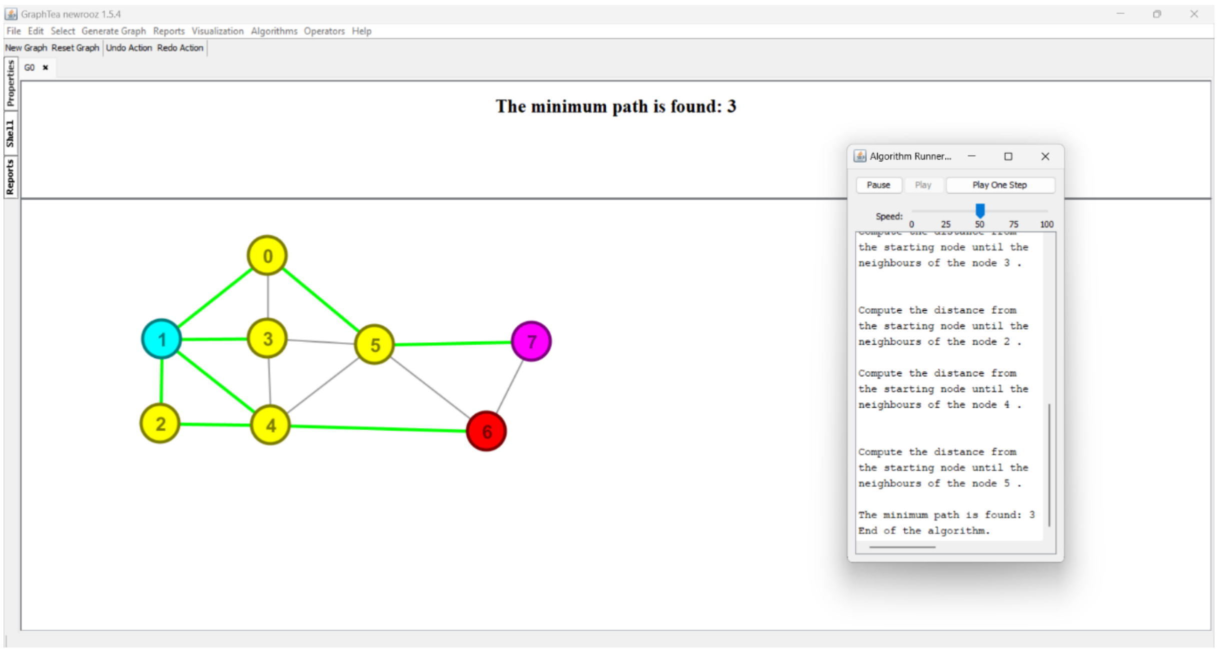
- The mention of computing distances and finding the minimum path aligns with path-finding algorithms. This could be indicative of algorithms such as Dijkstra's algorithm or A*.

3. Visualization

- The use of the red vertex is likely part of a visualization mechanism to highlight the current vertex during the algorithm's execution.

This output suggests that the algorithm is likely a path-finding algorithm, and the output demonstrates its step-by-step process of exploring neighbors and finding the minimum path.

GraphTea Discussion (picture 2)


GraphTea newrooz 1.5.4
File Edit Select Generate Graph Reports Visualization Algorithms Operators Help
New Graph Reset Graph Undo Action Redo Action
Properties
Shell
Reports
G0
Algorithm Runner...
Pause
Play
Play One Step
Speed:
0 25 50 75 100
0
1
2
3
4
5
6
7

The minimum path is found: 3

```
the starting node until the
neighbours of the node 3 .


Compute the distance from
the starting node until the
neighbours of the node 2 .

Compute the distance from
the starting node until the
neighbours of the node 4 .


Compute the distance from
the starting node until the
neighbours of the node 5 .

The minimum path is found: 3
End of the algorithm.
```

Algorithm outpout explanation

1. Start of the Algorithm

- The algorithm begins its execution.
- The user is prompted to select a starting vertex.
- The user is prompted to select a target vertex.

2. Searching for the Minimum Path

- The algorithm is now searching for the minimum path from the starting vertex to the target vertex.
- The red vertex is used to indicate the current vertex under consideration.

3. Compute Distance for Neighbors of Node 1

- The algorithm computes the distance from the starting node to the neighbors of Node 1.
- This involves evaluating the weights or distances associated with edges connected to Node 1.

4. Compute Distance for Neighbors of Node 0

- Similarly, the algorithm computes the distance from the starting node to the neighbors of Node 0.
- This process is repeated for each neighboring node.

5. Compute Distance for Neighbors of Node 3

- The algorithm continues to compute distances for the neighbors of Node 3.

6. Compute Distance for Neighbors of Node 2

- The distances from the starting node to the neighbors of Node 2 are calculated.

7. Compute Distance for Neighbors of Node 4

- Similar computations are performed for the neighbors of Node 4.

8. Compute Distance for Neighbors of Node 5

- The algorithm computes distances for the neighbors of Node 5.

9. Minimum Path Found

- The output indicates that the minimum path is found, and the distance associated with this path is 3.

10. End of the Algorithm

- The algorithm concludes its execution.

Additional Explanation

1. Computing Distances

- The algorithm uses a method, possibly Dijkstra's algorithm, to compute distances from the starting node to its neighbors iteratively.
- The goal is to find the minimum path from the starting vertex to the target vertex.

2. Visualization

- The mention of the red vertex suggests that there might be a visualization component, such as highlighting the current vertex for better understanding.

3. Result Interpretation

- A minimum path with a distance of 3 is found, indicating the optimal route from the starting vertex to the target vertex.

This output suggests that the algorithm is likely a path-finding algorithm, possibly Dijkstra's algorithm, which iteratively explores vertices and edges to find the shortest path in a graph. The mention of distances and the minimum path being found align with the typical behavior of such algorithms.

Conclusion

In conclusion, this in-depth study facilitated a detailed exploration of the methodologies and strategies implemented in our Python code dedicated to solving the shortest path problem in a weighted graph. Leveraging the judicious use of the Dijkstra algorithm and the NetworkX library, our approach provides an efficient and elegant solution for handling complex graphs.

The initiative to allow users to define key parameters such as the number of vertices, edges, and associated weights significantly facilitated the modeling of diverse networks. Our methodology, utilizing the `nx.single_source_dijkstra` function from NetworkX, demonstrated an effective resolution of the problem, capitalizing on the intrinsic capabilities of the Dijkstra algorithm.

The thorough analysis of the generated results, highlighting the shortest paths and their lengths for specific vertex pairs, enriched our understanding of the algorithm's behavior in various contexts. We emphasized the practical implications of these results, particularly in areas such as route optimization or network planning.

This research has contributed to a better understanding of methodological choices related to solving the shortest path problem. By emphasizing the relevance and potential applicability of our results, this study stands as a valuable resource in the field of network analysis. The use of GraphTea for additional verification enhances the credibility of our approach. In summary, our work provides a significant contribution to the understanding and practical application of optimal path-finding algorithms in complex network environments.

Bibliography

1. Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269-271.

- The original paper introducing Dijkstra's algorithm.

2. NetworkX Developers. (2022). NetworkX Documentation. [Online]. Available: https://networkx.github.io/documentation/stable/

- The official documentation for the NetworkX library, providing details on functions, usage, and examples.

3. Hagberg, A., Swart, P., & S Chult, D. (2008). Exploring network structure, dynamics, and function using NetworkX. In Proceedings of the 7th Python in Science Conference (Vol. 11, pp. 11-15).

- A conference paper introducing NetworkX and its capabilities for network analysis in Python.

4. Van Rossum, G., & Drake, F. L. (2003). Python 2.3 Documentation. [Online]. Available: https://docs.python.org/2.3/

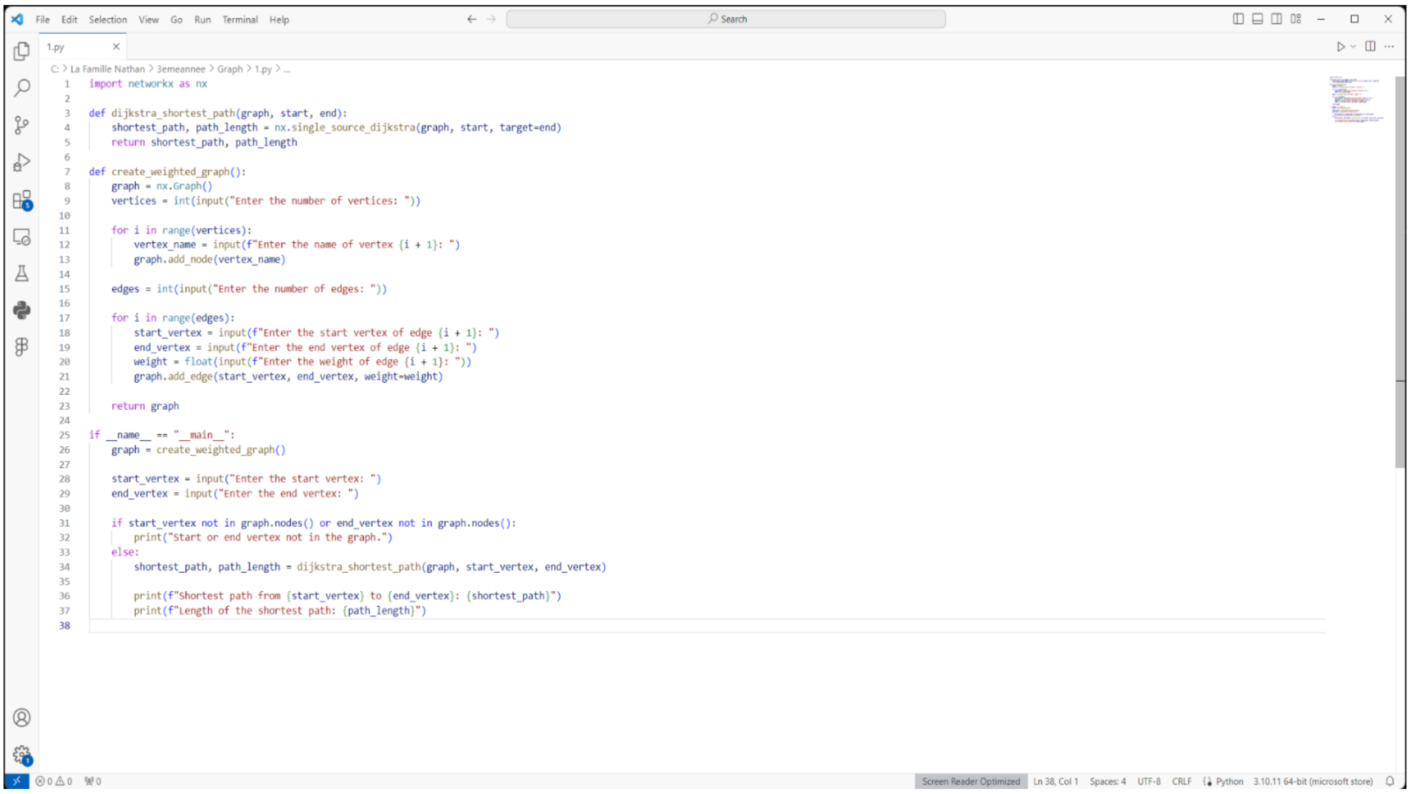
- The official documentation for Python 2.3, which might be relevant if you're using an older version of Python (though it's recommended to use a more recent version if possible).

5. McKinney, W. (2017). Python for Data Analysis. O'Reilly Media.

- A comprehensive book on using Python for data analysis, which includes coverage of libraries like NetworkX.

Appendix

```
import networkx as nx

def dijkstra_shortest_path(graph, start, end):
    shortest_path, path_length = nx.single_source_dijkstra(graph, start, target=end)
    return shortest_path, path_length

def create_weighted_graph():
    graph = nx.Graph()
    vertices = int(input("Enter the number of vertices: "))

    for i in range(vertices):
        vertex_name = input(f"Enter the name of vertex {i + 1}: ")
        graph.add_node(vertex_name)

    edges = int(input("Enter the number of edges: "))

    for i in range(edges):
        start_vertex = input(f"Enter the start vertex of edge {i + 1}: ")
        end_vertex = input(f"Enter the end vertex of edge {i + 1}: ")
        weight = float(input(f"Enter the weight of edge {i + 1}: "))
        graph.add_edge(start_vertex, end_vertex, weight=weight)

    return graph

if __name__ == "__main__":
    graph = create_weighted_graph()

    start_vertex = input("Enter the start vertex: ")
    end_vertex = input("Enter the end vertex: ")

    if start_vertex not in graph.nodes() or end_vertex not in graph.nodes():
        print("Start or end vertex not in the graph.")
    else:
        shortest_path, path_length = dijkstra_shortest_path(graph, start_vertex, end_vertex)

        print(f"Shortest path from {start_vertex} to {end_vertex}: {shortest_path}")
        print(f"Length of the shortest path: {path_length}")
```



C: > La Famille Nathan > 3emeannee > Graph > 1.py > ...

```python
import networkx as nx

def dijkstra_shortest_path(graph, start, end):
    shortest_path, path_length = nx.single_source_dijkstra(graph, start, target=end)
    return shortest_path, path_length

def create_weighted_graph():
    graph = nx.Graph()
    vertices = int(input("Enter the number of vertices: "))

    for i in range(vertices):
        vertex_name = input(f"Enter the name of vertex {i + 1}: ")
        graph.add_node(vertex_name)

    edges = int(input("Enter the number of edges: "))

    for i in range(edges):
        start_vertex = input(f"Enter the start vertex of edge {i + 1}: ")
        end_vertex = input(f"Enter the end vertex of edge {i + 1}: ")
        weight = float(input(f"Enter the weight of edge {i + 1}: "))
        graph.add_edge(start_vertex, end_vertex, weight=weight)

    return graph

if __name__ == "__main__":
    graph = create_weighted_graph()

    start_vertex = input("Enter the start vertex: ")
    end_vertex = input("Enter the end vertex: ")

    if start_vertex not in graph.nodes() or end_vertex not in graph.nodes():
        print("Start or end vertex not in the graph.")
    else:
        shortest_path, path_length = dijkstra_shortest_path(graph, start_vertex, end_vertex)

        print(f"Shortest path from {start_vertex} to {end_vertex}: {shortest_path}")
        print(f"Length of the shortest path: {path_length}")

```

File Edit Selection View Go Run Terminal Help
Search
1.py
Screen Reader Optimized
Ln 38, Col 1
Spaces: 4
UTF-8
CRLF
Python
3.10.11 64-bit (microsoft store)