Python Implementation of Graph-Theoretic Concepts: Closed Neighbourhood and Distance Between Vertices

[1]S. Vimalajenifer, [2]P. Sathiya, [3]C. Abirami

[1,2,3]Department of Data Science, Ayya Nadar Janaki Ammal College, Sivakasi, India

Abstract

This paper presents a Python implementation of two fundamental graph-theoretic concepts: the closed neighbourhood of a vertex and the distance between any two vertices in a simple connected graph. Using adjacency list representation, efficient algorithms are developed to compute the closed neighbourhood $N[v]$and the shortest path distance $d(u,v)$for any vertices u and v. The proposed implementations are simple, readable, and suitable for educational purposes as well as for integration into larger graph-based applications. By bridging pure graph theory with practical programming, this work demonstrates how core mathematical concepts can be effectively translated into computational tools using Python. The paper also discusses the theoretical background, provides detailed code explanations, and includes illustrative examples and figures to enhance understanding. This serves as a starting point for incorporating additional graph theory concepts into programming languages and highlights the growing importance of computational approaches in discrete mathematics.

AMS Subject Classification (2020): 05C12, 05C85, 68R10, 68W05

Keywords: Graph theory, Closed neighbourhood, shortest path distance, Python implementation, Adjacency list, Computational graph theory

1. Introduction

Let $G = (V(G), E(G))$ be a simple connected graph. The distance between two vertices u and v is the length of the shortest path between u and v and is denoted by $d(u,v)$. Two vertices u and v are adjacent if the distance between them is one. The set of all vertices which are adjacent to the vertex v is called as *open neighbourhood* of v and is denoted by $N(v)$. The set $N[u] = N(u) \cup \{u\}$ is called as *closed neighbourhood* of u.

The eccentricity of a vertex v is the distance between the vertex v and the vertex farthest from v. The radius of a graph is the minimum eccentricity among all vertices and is denoted by rad(G). The diameter of a graph G is the maximum eccentricity among all vertices and is denoted by diam(G).

In today's AI-driven world, almost everything is being computerized. This naturally raises an important question: Why should basic graph theory concepts remain only as theoretical ideas? Even simple notions such as the closed neighbourhood of a vertex, distance, radius, and diameter of a graph can be easily implemented using Python.

There are several popular Python libraries available for working with graphs. The most commonly used ones are **NetworkX**, **python-igraph**, and **graph-tool**. Each of these libraries has its own strengths and serves different needs depending on the size of the graph and the purpose of the user.

This paper presents simple, beginner-friendly algorithms and Python programs for these fundamental graph theory concepts. These implementations are useful for students, researchers, and teachers to understand, verify, and apply graph theory ideas in practice. They also help in building a strong foundation for developing more advanced graph-based applications.

For basic graph theory concepts, one may refer to [1–5]. For Python programming, one may refer to [6, 7]. S. Vimalajenifer et al. [8, 9] have already worked on the implementation of finding the radial radio numbers of complete multipartite graphs and cycle graphs.

2. Closed neighbours of a vertex

The set of all vertices which are adjacent to the vertex v is called as *open neighbourhood* of v and is denoted by $N(v)$. The set $N[u] = N(u) \cup \{u\}$ is called as *closed neighbourhood* of u. Figure 2.1 shows an example of computing the closed neighbourhood of a vertex. Consider the graph G, shown in Figure 2.1

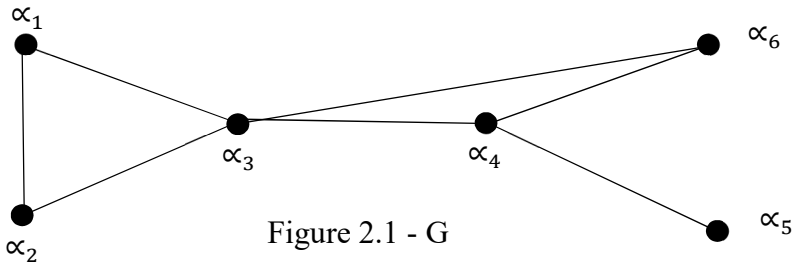

Figure 2.1 - G

Here, $N[\propto_1] = \{\propto_1, \propto_2, \propto_3\}$; $N[\propto_2] = \{\propto_1, \propto_2, \propto_3\}$; $N[\propto_3] = \{\propto_1, \propto_2, \propto_3, \propto_4, \propto_6\}$; $N[\propto_4] = \{\propto_3, \propto_6, \propto_5\}$; $N[\propto_5] = \{\propto_4\}$; $N[\propto_6] = \{\propto_4, \propto_5\}$.

2.1 Node Creation and Find closed neighbours of a vertex

Algorithm: Computation of Closed Neighbourhood

Input: A connected graph represented by an adjacency list and a vertex v.

Output: The closed neighbourhood $N[v]$of vertex v.

```
# Function to create graph automatically
def create_graph(n):
    graph = {}

    for i in range(n):
        neighbours = list(map(int, input(f"Enter neighbours of {i}:
").split()))
        graph[i] = neighbours
    return graph
# Function to find closed neighbours
def closed_neighbours(graph, vertex):
    closed = [vertex]
    for neighbour in graph[vertex]:
        closed.append(neighbour)
    return closed
# Get number of vertices
n = int(input("Enter number of vertices: "))
# Create graph vertices automatically from 0 to n-1
graph = create_graph(n)
# Display graph
print("Graph =", graph)
```

```
# Get vertex to find closed neighbours
vertex = int(input("Enter the vertex to find closed neighbours: "))
# Find result
result = closed_neighbours(graph, vertex)
# Output
print("Closed neighbours of", vertex, "=", result)
```

2.2 Explanation:

The above Python program is designed to find the closed neighbours of a selected vertex in a graph using an adjacency list representation. Initially, the program reads the number of vertices from the user. Based on this value, the vertices are automatically numbered from 0 to n−1. The function create_graph(n) is used to construct the graph dynamically. Inside this function, an empty dictionary is created to store the graph, where each key represents a vertex and its value represents the list of neighbouring vertices. A loop runs from 0 to n-1, and for each vertex, the user enters its adjacent vertices. These neighbours are converted into integer values and stored in the dictionary.

After creating the graph, the program displays the complete graph structure. Next, the user is asked to choose a specific vertex for which the closed neighbourhood has to be determined. The function closed_neighbours(graph, vertex) is then called. This function first creates a list containing the selected vertex itself, because in graph theory the closed neighbourhood always includes the vertex. Then, it traverses all adjacent vertices of the selected node and appends them to the list. Finally, the function returns the complete closed neighbourhood.

The mathematical representation used in this program is:

$$N[v]=\{v\}\cup N(v),$$

where v is the selected vertex, N(v) represents its neighbouring vertices, and N[v] denotes the closed neighbourhood. For example, if the selected vertex is 0 and its neighbours are 1 and 2, the closed neighbourhood will be $\{0,1,2\}$. This program is simple, efficient, and suitable for graph theory applications such as network analysis, social connectivity, and shortest path studies.

2.3 Output:

```
Enter number of vertices: 4
Enter neighbours of 0: 1 3
Enter neighbours of 1: 0 2
Enter neighbours of 2: 1 3
Enter neighbours of 3: 0 2
Graph = {0: [1, 3], 1: [0, 2], 2: [1, 3], 3: [0, 2]}
Enter the vertex to find closed neighbours: 3
Closed neighbours of 3 = [3, 0, 2]
```

3. Distance between two vertices

The distance between any two vertices u and v is the length of the shortest path between them and is denotes by $d(u,v)$. Figure 3.1 illustrates the method of finding the distance between any two vertices in G.

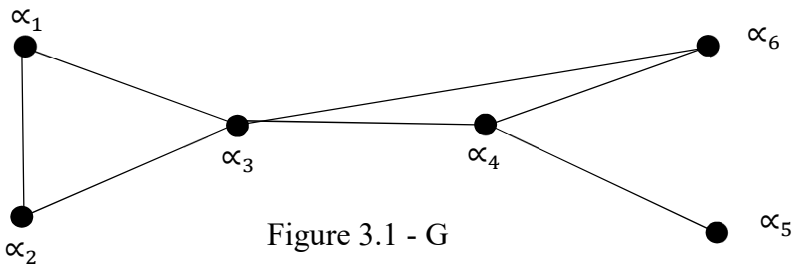

Figure 3.1 - G

We note that, $d(\propto_1, \propto_2) = 1$; $d(\propto_1, \propto_3) = 1$; $d(\propto_1, \propto_4) = 2$; $d(\propto_1, \propto_5) = 3$; $d(\propto_1, \propto_6) = 2$. Similarly, we found the distance between all possible pairs of vertices.

3.1 Node Creation and Find Distance Between Vertices

```
# Function to create graph automatically
def create_graph(n):
    graph = {}
    for i in range(n):
        neighbours = list(map(int, input(f"Enter neighbours of {i}:
").split()))
        graph[i] = neighbours
    return graph
def find_distance(graph, start, end):
    queue = [(start, 0)]
    visited = []
    while queue:
        current, dist = queue.pop(0)
        if current == end:
            return dist
        if current not in visited:
            visited.append(current)
            for neighbour in graph[current]:
                queue.append((neighbour, dist + 1))
    return -1
# Get number of vertices
n = int(input("Enter number of vertices: "))
# Create graph vertices automatically from 0 to n-1
graph = create_graph(n)
# Display graph
print("Graph =", graph)
start = int(input("Enter start vertex: "))
end = int(input("Enter end vertex: "))
print("Distance =", find_distance(graph, start, end))
```

3.2 Explanation:

The above Python program is used to find the shortest distance between two vertices in a graph. The graph is represented using an adjacency list, where each vertex is stored as a key

in a dictionary and its neighbouring vertices are stored as a list of values. First, the program asks the user to enter the number of vertices, and based on this input, the vertices are automatically numbered from 0 to n−1. The function create_graph(n) is responsible for constructing the graph dynamically. Inside this function, an empty dictionary is initialized, and for every vertex from 0 to n-1, the user enters its adjacent vertices. These neighbours are then stored in the dictionary format, creating the complete graph structure.

After the graph is created, the user enters the starting vertex and the ending vertex between which the distance has to be found. The function find_distance(graph, start, end) is then used to calculate the shortest distance. This function applies the Breadth First Search (BFS) traversal method, which is widely used in graph theory to find the shortest path in an unweighted graph. A queue is initialized with the starting vertex and distance 0. The algorithm repeatedly removes the first element from the queue and checks whether the current vertex is the destination vertex. If the destination is reached, the current distance value is returned as the shortest distance.

If the vertex is not yet visited, it is marked as visited, and all its neighbouring vertices are added to the queue with their distance increased by 1. This process continues level by level until the destination vertex is found. If there is no path between the selected vertices, the function returns -1, indicating that the destination is unreachable. For example, if the path from vertex 0 to vertex 3 is $0 \rightarrow 1 \rightarrow 3$, then the distance is 2 because two edges are crossed.

The mathematical concept used here is the shortest path distance, which can be represented as:

$$d(u,v)=\text{minimum number of edges from } u \text{ to } v$$

This program is highly useful in applications such as network routing, social network analysis, map navigation, and graph traversal problems, where finding the minimum distance between two nodes is important.

3.3 Output

```
Enter number of vertices: 4
Enter neighbours of 0: 1 3
Enter neighbours of 1: 0 2
Enter neighbours of 2: 1 3
Enter neighbours of 3: 0 2
Graph = {0: [1, 3], 1: [0, 2], 2: [1, 3], 3: [0, 2]}
Enter start vertex: 2
Enter end vertex: 3
Distance = 1
```

Conclusion

In this paper, we implement fundamental graph theoretical concepts such as finding the closed neighbourhood of a vertex and computing the distance between any two vertices in a

simple connected graph. This work aims to bridge the gap between pure mathematics and computer science by serving as a starting point for integrating graph theory concepts into programming. In the future, many other graph-theoretic ideas can similarly be incorporated into programming languages.

Acknowledgement

The authors sincerely thank the Management of Ayya Nadar Janaki Ammal College (Autonomous), Sivakasi, Tamil Nadu, India, for their constant support and encouragement. The authors also gratefully acknowledge the Postgraduate Department of Data Science, Computer Science Lab and the Science Instrumentation Centre of Ayya Nadar Janaki Ammal College for providing the necessary facilities and support to carry out this research work.

References

1. R. Balakrishnan and K. Ranganathan, *A Textbook of Graph Theory*, Second Edition, Springer, New York, 2012.
2. G. Chartrand and P. Zhang, *Chromatic Graph Theory*, Second Edition, CRC Press, Taylor and Francis Group, New York, 2019.
3. R. Diestel, *Graph Theory*, Sixth Edition, Springer, Heidelberg, 2025.
4. A. Benjamin, G. Chartrand and P. Zhang, *The Fascinating World of Graph Theory*, Princeton University Press, 2015.
5. C. M. Farrelly, *Modern Graph Theory Algorithms with Python: Harness the power of graph algorithms and real-world network applications using Python*, Packt Publishing, 2024.
6. E. Matthes, *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*, Third Edition, No Starch Press, 2023.
7. A. Sweigart, *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*, Second Edition, No Starch Press, 2019.
8. S. Vimalajenifer, C. Abirami, P. Sathiya, “Exact radial radio number of complete multipartite graphs: proof, optimal algorithm and python implementation”, Journal of Electrical Engineering, Vol 10, Issue 12, 2025.
9. S. Vimalajenifer, “A constructive algorithm for the radial radio labeling of cycle graph”, JZU Natural Sciences, Vol 56, Issue 12, 2025.