FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION
ITMO UNIVERSITY

Report
on the practical task No. 5

Algorithms on graphs: Introduction to graphs and basic algorithms on graphs

Performed by
Daan Rodríguez
Academic group
J4132c
Accepted by
Dr Petr Chunaev

St. Petersburg
2023

Goal

The goal of this task is to introduce the fundamental concepts of graphs and basic graph algorithms.

Firstly, the task requires generating an adjacency matrix for a simple undirected unweighted graph with 100 vertices and 200 edges and then transforming it into an adjacency list. Next, the Depth-First Search (DFS) algorithm is employed to find connected components in the graph, and Breadth-First Search (BFS) is used to determine the shortest path between two random vertices. Furthermore, the task involves analyzing the outcomes of these algorithms, such as identifying connected components in the graph and evaluating the shortest path found by BFS.

Formulation of the problem

This task involves generating a random adjacency matrix for a simple undirected unweighted graph with 100 vertices and 200 edges. The matrix must be symmetric and comprise only 0s and 1s. Subsequently, the matrix is transformed into an adjacency list representation. The generated graph is visualized, with several rows of the adjacency matrix and adjacency list printed for comparison. Depth-First Search (DFS) is utilized to identify connected components, and Breadth-First Search (BFS) is employed to find the shortest path between two random vertices. The results obtained from these algorithms are analyzed. Additionally, the report includes descriptions of the data structures (such as stacks, queues, and arrays) and design techniques (such as recursion for DFS and iteration for BFS) used within the algorithms, providing a comprehensive overview of the graph theory concepts explored.

Brief theoretical part

1. Graph Representations:
   - Adjacency Matrix: An NxN matrix, where N is the number of vertices. If there is an edge between vertices i and j, matrix entry (i, j) (and (j, i)) is 1; otherwise, it's 0. Symmetry is maintained for undirected graphs.
   - Adjacency List: An array of lists, where each index represents a vertex, and the corresponding list contains adjacent vertices.

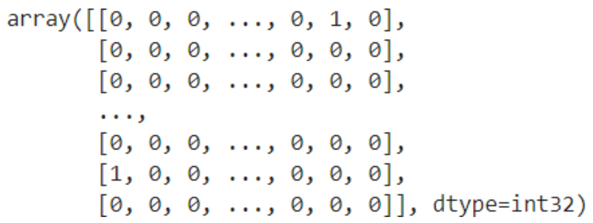
2. Depth-First Search (DFS): DFS explores as far as possible along each branch before backtracking. It uses a stack or recursion to keep track of vertices. DFS is used in this task to identify connected components in the graph.

3. Breadth-First Search (BFS): BFS explores all the vertices at the present depth before moving to vertices at the next depth level. It employs a queue to manage vertices. Here, BFS is utilized to find the shortest path between two vertices.

Result:

I. Question 1

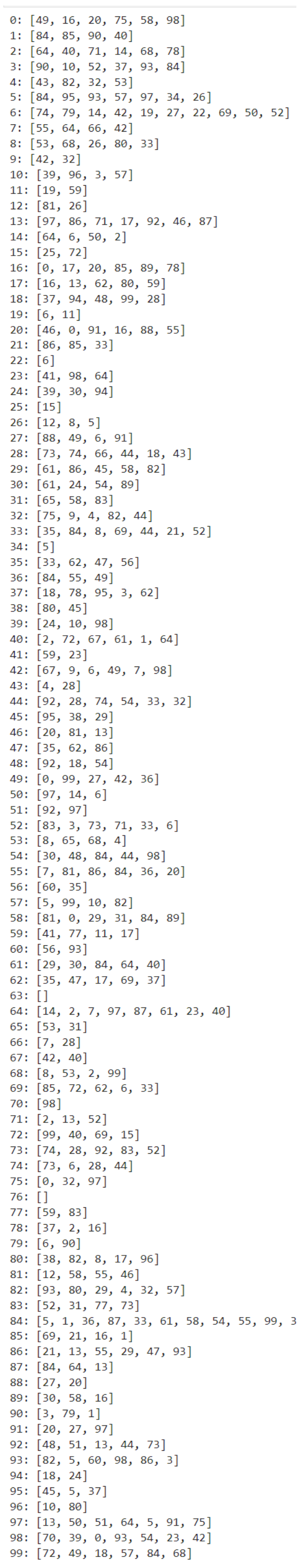
1. Generate a random adjacency matrix



```
array([[0, 0, 0, ..., 0, 1, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=int32)
```

Figure 1.1 - adjacency matrix

2. Adjacency list



```
0: [49, 16, 20, 75, 58, 98]
1: [84, 85, 90, 40]
2: [64, 40, 71, 14, 68, 78]
3: [90, 10, 52, 37, 93, 84]
4: [43, 82, 32, 53]
5: [84, 95, 93, 57, 97, 34, 26]
6: [74, 79, 14, 42, 19, 27, 22, 69, 50, 52]
7: [55, 64, 66, 42]
8: [53, 68, 26, 80, 33]
9: [42, 32]
10: [39, 96, 3, 57]
11: [19, 59]
12: [81, 26]
13: [97, 86, 71, 17, 92, 46, 87]
14: [64, 6, 50, 2]
15: [25, 72]
16: [0, 17, 20, 85, 89, 78]
17: [16, 13, 62, 80, 59]
18: [37, 94, 48, 99, 28]
19: [6, 11]
20: [46, 0, 91, 16, 88, 55]
21: [86, 85, 33]
22: [6]
23: [41, 98, 64]
24: [39, 30, 94]
25: [15]
26: [12, 8, 5]
27: [88, 49, 6, 91]
28: [73, 74, 66, 44, 18, 43]
29: [61, 86, 45, 58, 82]
30: [61, 24, 54, 89]
31: [65, 58, 83]
32: [75, 9, 4, 82, 44]
33: [35, 84, 8, 69, 44, 21, 52]
34: [5]
35: [33, 62, 47, 56]
36: [84, 55, 49]
37: [18, 78, 95, 3, 62]
38: [80, 45]
39: [24, 10, 98]
40: [2, 72, 67, 61, 1, 64]
41: [59, 23]
42: [67, 9, 6, 49, 7, 98]
43: [4, 28]
44: [92, 28, 74, 54, 33, 32]
45: [95, 38, 29]
46: [20, 81, 13]
47: [35, 62, 86]
48: [92, 18, 54]
49: [0, 99, 27, 42, 36]
50: [97, 14, 6]
51: [92, 97]
52: [83, 3, 73, 71, 33, 6]
53: [8, 65, 68, 4]
54: [30, 48, 84, 44, 98]
55: [7, 81, 86, 84, 36, 20]
56: [60, 35]
57: [5, 99, 10, 82]
58: [81, 0, 29, 31, 84, 89]
59: [41, 77, 11, 17]
60: [56, 93]
61: [29, 30, 84, 64, 40]
62: [35, 47, 17, 69, 37]
63: []
64: [14, 2, 7, 97, 87, 61, 23, 40]
65: [53, 31]
66: [7, 28]
67: [42, 40]
68: [8, 53, 2, 99]
69: [85, 72, 62, 6, 33]
70: [98]
71: [2, 13, 52]
72: [99, 40, 69, 15]
73: [74, 28, 92, 83, 52]
74: [73, 6, 28, 44]
75: [0, 32, 97]
76: []
77: [59, 83]
78: [37, 2, 16]
79: [6, 90]
80: [38, 82, 8, 17, 96]
81: [12, 58, 55, 46]
82: [93, 80, 29, 4, 32, 57]
83: [52, 31, 77, 73]
84: [5, 1, 36, 87, 33, 61, 58, 54, 55, 99, 3
85: [69, 21, 16, 1]
86: [21, 13, 55, 29, 47, 93]
87: [84, 64, 13]
88: [27, 20]
89: [30, 58, 16]
90: [3, 79, 1]
91: [20, 27, 97]
92: [48, 51, 13, 44, 73]
93: [82, 5, 60, 98, 86, 3]
94: [18, 24]
95: [45, 5, 37]
96: [10, 80]
97: [13, 50, 51, 64, 5, 91, 75]
98: [70, 39, 0, 93, 54, 23, 42]
99: [72, 49, 18, 57, 84, 68]
```

Figure 1.2 – print some rows of adjacency list

3. Visualize the graph

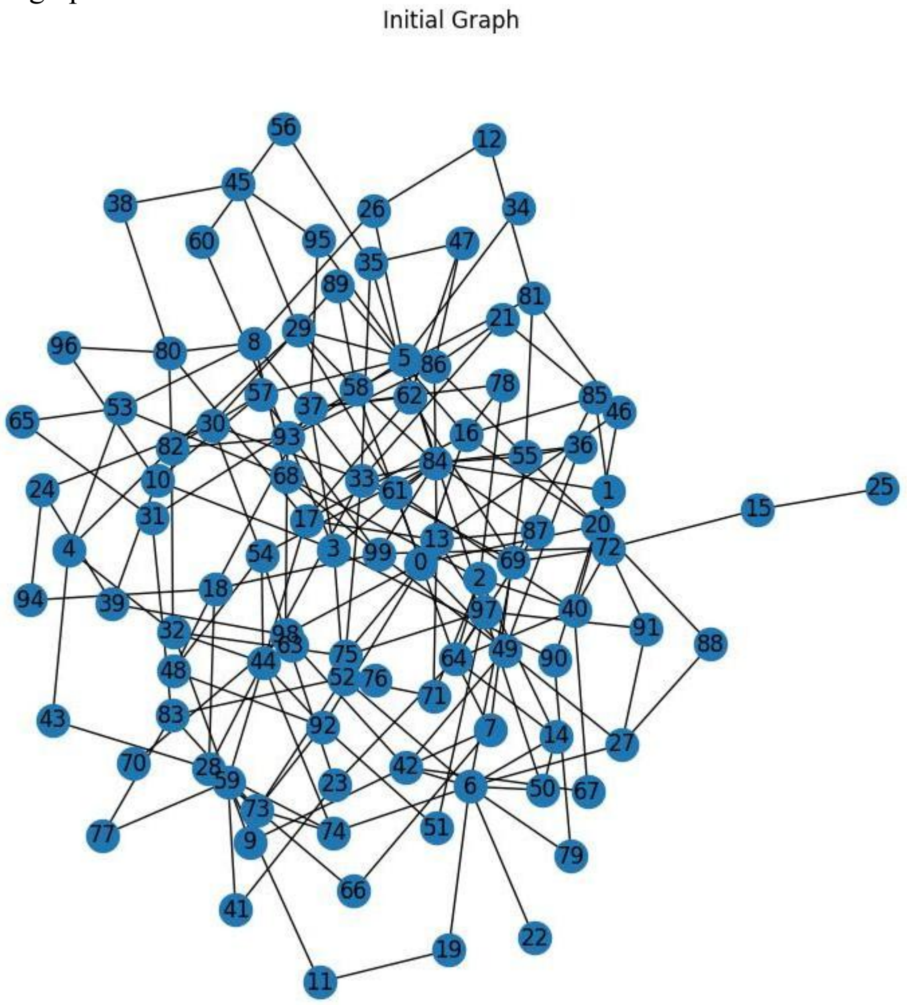


Figure 1.3 – the visualization of graph

Adjacency Matrix:

- Convenient for Checking Edge Existence: Ideal for quickly checking if there is an edge between specific vertices (constant time complexity, O (1)).
- Space Efficiency for Dense Graphs: Suitable for dense graphs where most vertices are connected (uses less space when the graph is densely populated with edges).

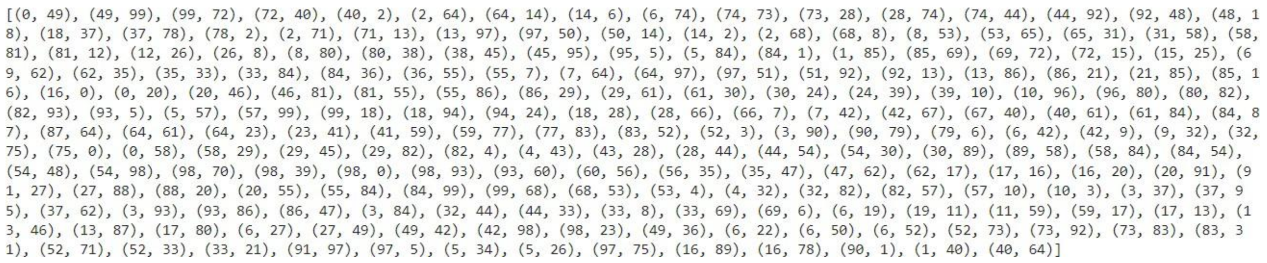
Adjacency List:

- Convenient for Sparse Graphs: Efficient for sparse graphs where there are relatively few edges, as it doesn't waste space storing non-existing edges.
- Efficient Iteration Through Neighbors: Suitable for operations requiring iteration through neighbors of a vertex, making it efficient for algorithms like DFS and BFS (linear time complexity, O (V + E), where V is the number of vertices and E is the number of edges).
- Memory Efficiency: Consumes less memory for graphs with fewer edges, saving space compared to a dense adjacency matrix.

In summary, it is more convenient to use an adjacency matrix when fast edge presence checks are essential or when dealing with dense graphs. When representing sparse graphs, it is more convenient to use adjacency lists, especially when the algorithm involves efficient traversal and memory usage is an issue. The choice depends on the specific operation and the characteristics of the graph being processed.

II. Question 2

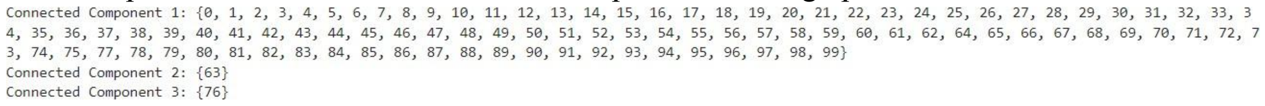
1. Print all the vertices



```
[(0, 49), (49, 99), (99, 72), (72, 40), (40, 2), (2, 64), (64, 14), (14, 6), (6, 74), (74, 73), (73, 28), (28, 74), (74, 44), (44, 92), (92, 48), (48, 1
8), (18, 37), (37, 78), (78, 2), (2, 71), (71, 13), (13, 97), (97, 50), (50, 14), (14, 2), (2, 68), (68, 8), (8, 53), (53, 65), (65, 31), (31, 58), (58,
81), (81, 12), (12, 26), (26, 8), (8, 80), (80, 38), (38, 45), (45, 95), (95, 5), (5, 84), (84, 1), (1, 85), (85, 69), (69, 72), (72, 15), (15, 25), (6
9, 62), (62, 35), (35, 33), (33, 84), (84, 36), (36, 55), (55, 7), (7, 64), (64, 97), (97, 51), (51, 92), (92, 13), (13, 86), (86, 21), (21, 85), (85, 1
6), (16, 0), (0, 20), (20, 46), (46, 81), (81, 55), (55, 86), (86, 29), (29, 61), (61, 30), (30, 24), (24, 39), (39, 10), (10, 96), (96, 80), (80, 82),
(82, 93), (93, 5), (5, 57), (57, 99), (99, 18), (18, 94), (94, 24), (18, 28), (28, 66), (66, 7), (7, 42), (42, 67), (67, 40), (40, 61), (61, 84), (84, 8
7), (87, 64), (64, 61), (64, 23), (23, 41), (41, 59), (59, 77), (77, 83), (83, 52), (52, 3), (3, 90), (90, 79), (79, 6), (6, 42), (42, 9), (9, 32), (32,
75), (75, 0), (0, 58), (58, 29), (29, 45), (29, 82), (82, 4), (4, 43), (43, 28), (28, 44), (44, 54), (54, 30), (30, 89), (89, 58), (58, 84), (84, 54),
(54, 48), (54, 98), (98, 70), (98, 39), (98, 0), (98, 93), (93, 60), (60, 56), (56, 35), (35, 47), (47, 62), (62, 17), (17, 16), (16, 20), (20, 91), (9
1, 27), (27, 88), (88, 20), (20, 55), (55, 84), (84, 99), (99, 68), (68, 53), (53, 4), (4, 32), (32, 82), (82, 57), (57, 10), (10, 3), (3, 37), (37, 9
5), (37, 62), (3, 93), (93, 86), (86, 47), (3, 84), (32, 44), (44, 33), (33, 8), (33, 69), (69, 6), (6, 19), (19, 11), (11, 59), (59, 17), (17, 13), (1
3, 46), (13, 87), (17, 80), (6, 27), (27, 49), (49, 42), (42, 98), (98, 23), (49, 36), (6, 22), (6, 50), (6, 52), (52, 73), (73, 92), (73, 83), (83, 3
1), (52, 71), (52, 33), (33, 21), (91, 97), (97, 5), (5, 34), (5, 26), (97, 75), (16, 89), (16, 78), (90, 1), (1, 40), (40, 64)]
```

Figure 2.1 – print all vertices

2. Use Depth-first search to find connected components of the graph



```
Connected Component 1: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 3
4, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 64, 65, 66, 67, 68, 69, 70, 71, 72, 7
3, 74, 75, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99}
Connected Component 2: {63}
Connected Component 3: {76}
```

Figure 2.2 – depth first search

We find three connected components in the graph, the result shown in the figure 2.2.

3. Use Breadth-first search to find a shortest path

```
Shortest Path from Node 18 to Node 22: [18, 28, 74, 6, 22]
```

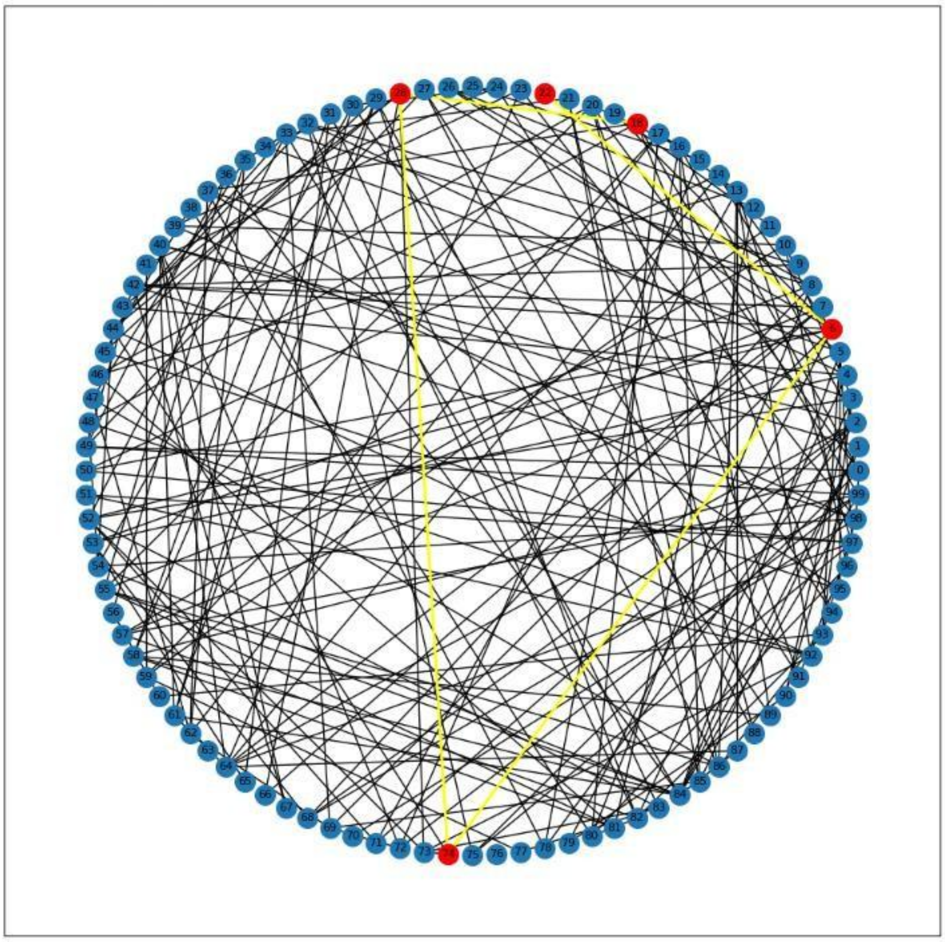


Figure 2.3 - breadth-first search

We find the shortest path is from node 18 to node 22 {18,28,74,6,22}, the result shown in the figure 2.3.

1) DFS Analysis for Connected Components:
   - Main Connected Component: DFS identified a major connected component (Connected Component 1) containing most of the nodes. This represents the core functional part of the network, possibly indicating a main user community in a social network or a central functional unit in a system.
   - Isolated Nodes: Connected Component 2 and Connected Component 3 contain isolated nodes (Nodes 63 and 76). These isolated nodes might represent special individuals or outliers within the network.

2) BFS Shortest Path Analysis:
   - Shortest Path: BFS discovered the shortest path from Node 18 to Node 22: 18,28,74,6,2218,28,74,6,22. This path represents the most efficient communication route between these nodes, critical for tasks like data transmission or network optimization.
   - Path Node Analysis: The chosen nodes along the path (Nodes 18, 28, 74, 6, and 22) likely signify crucial junctions or specific functional areas within the network.

III. Describe the data structures and design techniques used within the algorithms.

1) Data Structures:
   - Adjacency Matrix: An adjacency matrix is a 2D array (V x V, where V is the number of vertices) where each cell represents the presence or absence of an edge between two vertices. It is an efficient way to check if there is an edge between two specific vertices (constant time complexity, O (1)), but it consumes more space, especially for sparse graphs.
   - Adjacency List: An adjacency list uses an array of lists. The size of the array is equal to the number of vertices. Each element of the array is a list that contains the neighbors of a specific vertex. It is efficient for sparse graphs as it consumes less memory and allows for quick iteration through neighbors (linear time complexity, O (V + E), where V is the number of vertices and E is the number of edges).
2) Design Techniques:
   a) Depth-First Search (DFS):
      - Recursion: DFS uses recursion to explore as far as possible along each branch before backtracking. This technique leverages the call stack implicitly, allowing for easy backtracking and exploration of adjacent vertices in a systematic manner.
      - Visited Array: To prevent revisiting already explored vertices, a boolean array (or set) is often used to mark visited vertices. This ensures that each vertex is visited exactly once.
   b) Breadth-First Search (BFS):
      - Queue: BFS uses a queue data structure to maintain the order in which vertices are explored. This ensures that vertices are explored level by level, allowing BFS to find the shortest path.
      - Parent Array: To reconstruct the shortest path in BFS, a parent array is often used. Each element in the parent array stores the previous vertex on the shortest path from the source vertex. This array is populated during the BFS traversal.
   c) Path Reconstruction:
      - Parent Pointers: In algorithms like BFS, parent pointers (or parent arrays) are used to reconstruct the shortest path from the source to any other vertex. By backtracking from the destination vertex using parent pointers, the actual path is obtained.
      - Path Representation: The reconstructed path can be represented as a list or an array, allowing for easy visualization and analysis.

These data structures and design techniques form the foundation of graph algorithms. By employing these structures and techniques effectively, various graph-related problems, including connectivity analysis, shortest path finding, and component detection, can be efficiently solved.

Conclusions

In this task, we generated a random adjacency matrix for a simple undirected unweighted graph with 100 vertices and 200 edges. We then converted this matrix into an adjacency list representation. The adjacency matrix is convenient for checking the presence of edges between specific vertices (in constant time), while the adjacency list is useful for iterating through neighbors of a vertex efficiently (in time proportional to the degree of the vertex).

We used Depth-First Search (DFS) to find connected components in the graph. DFS is excellent for exploring all vertices in a connected component efficiently. Connected components are crucial in various applications, such as identifying communities in social networks or analyzing the structure of complex systems.

Additionally, we utilized Breadth-First Search (BFS) to find the shortest path between two randomly chosen vertices. BFS guarantees the shortest path in an unweighted graph, making it valuable for tasks like network routing or social network analysis where finding the shortest path is essential.

In summary, this task provided a hands-on experience with fundamental graph algorithms and illustrated the importance of choosing the right representation and algorithm for specific graph-related tasks. Understanding these algorithms and representations is crucial for solving a wide range of real-world problems in fields such as computer networking, social network analysis, and optimization.