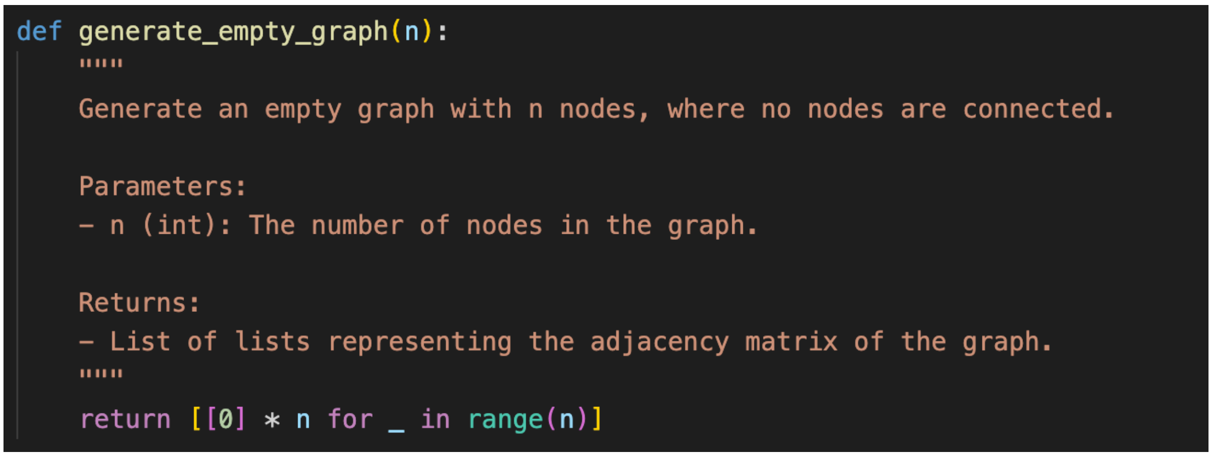
Josep Cubedo, Julian Harder, Teo Arqués, Gerard Figueras

Assignment 3

GRAPHS TEMPLATE 1:
Generate Empty Graph Function:

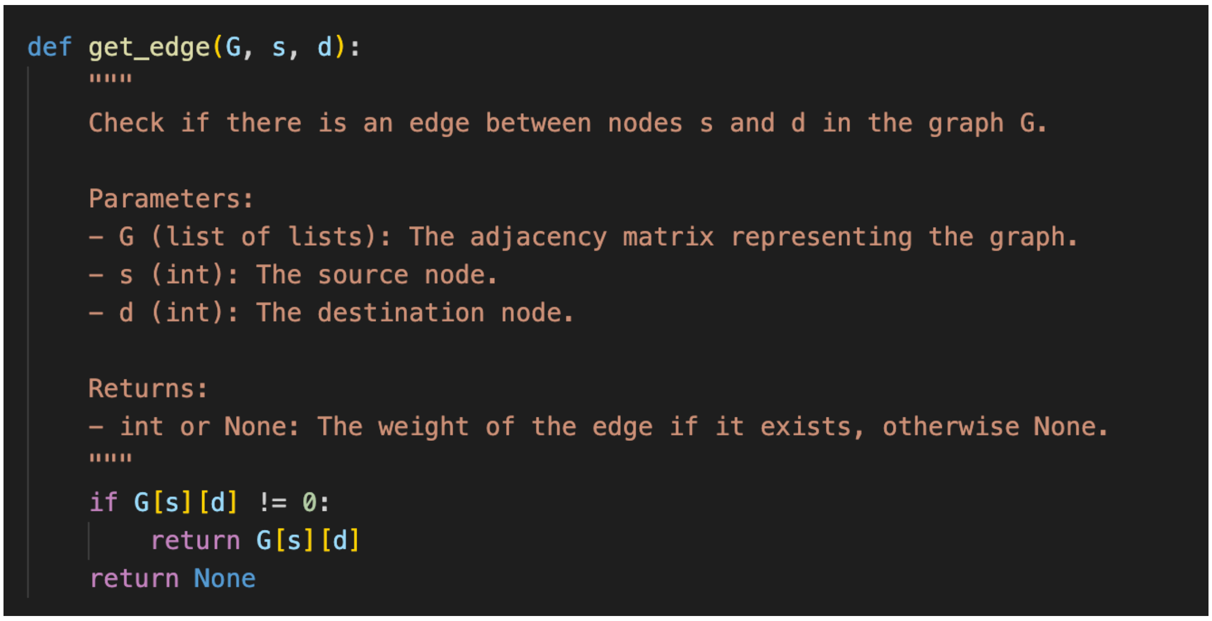


```python
def generate_empty_graph(n):
    """
    Generate an empty graph with n nodes, where no nodes are connected.

    Parameters:
    - n (int): The number of nodes in the graph.

    Returns:
    - List of lists representing the adjacency matrix of the graph.
    """
    return [[0] * n for _ in range(n)]
```

Purpose: This function creates an empty graph with n nodes, represented as an adjacency matrix. In this matrix, each node is initially unconnected to every other node.
How it works:

- The line G = [[0] * n for i in range(n)] creates a 2D list (matrix) with n rows and n columns, where each entry is initialized to 0.
- The 0 values indicate that there are no edges between any pairs of nodes. In other words, all nodes are disconnected initially.
- This matrix is then returned as G, which represents the unconnected graph.

Get Edge Function:



```python
def get_edge(G, s, d):
    """
    Check if there is an edge between nodes s and d in the graph G.

    Parameters:
    - G (list of lists): The adjacency matrix representing the graph.
    - s (int): The source node.
    - d (int): The destination node.

    Returns:
    - int or None: The weight of the edge if it exists, otherwise None.
    """
    if G[s][d] != 0:
        return G[s][d]
    return None
```

Purpose: This function checks if there is an edge between nodes s and d in the graph G. If an edge exists, it returns the weight of that edge; otherwise, it returns None.
How it works:

- G[s][d] accesses the value in the adjacency matrix at row s and column d. This value represents the weight of the edge from node s to node d.
- If G[s][d] is not 0, it indicates there is an edge between s and d with a certain weight, so the function returns G[s][d].
- If G[s][d] is 0, it means there is no edge between s and d, so the function returns None.

Create Edge Function:

```python
def create_edge(G, s, d, w):
    """
    Create an edge between nodes s and d with weight w in the graph G.

    Parameters:
    - G (list of lists): The adjacency matrix representing the graph.
    - s (int): The source node.
    - d (int): The destination node.
    - w (int): The weight of the edge.

    Returns:
    - List of lists: The updated adjacency matrix.
    """
    G[s][d] = w
    return G
```

- **Purpose**: This function adds a weighted edge with weight w between nodes s and d in the graph G.
- **How it works**:
    - G[s][d] = w sets the value at the position [s][d] in the adjacency matrix to w, which represents an edge from node s to node d with weight w.
    - If the graph is undirected, you would likely also set G[d][s] = w to ensure the edge is represented in both directions.

Nodes Selection Function:

```python
def nodes_selection(G, directed=False):
    """
    Select two different nodes at random from the graph G.

    Parameters:
    - G (list of lists): The adjacency matrix representing the graph.
    - directed (bool): Whether the graph is directed (default is False).

    Returns:
    - tuple: A tuple of two node indices (source, destination).
    """
    s, d = random.sample(range(len(G)), 2)
    return s, d
```

Purpose: This function selects two different nodes at random from the graph G to create a new edge.

How it works:

- `random.sample(range(len(G)), k=2)` randomly selects two unique nodes from the range of indices (0 to `len(G) - 1`), ensuring that the two nodes are different and do not refer to the same node.
- `k=2` specifies that two items are to be chosen from the list.
- The function then returns the selected nodes as a tuple, which can be used to create an edge between them.

Generate Graph Function:

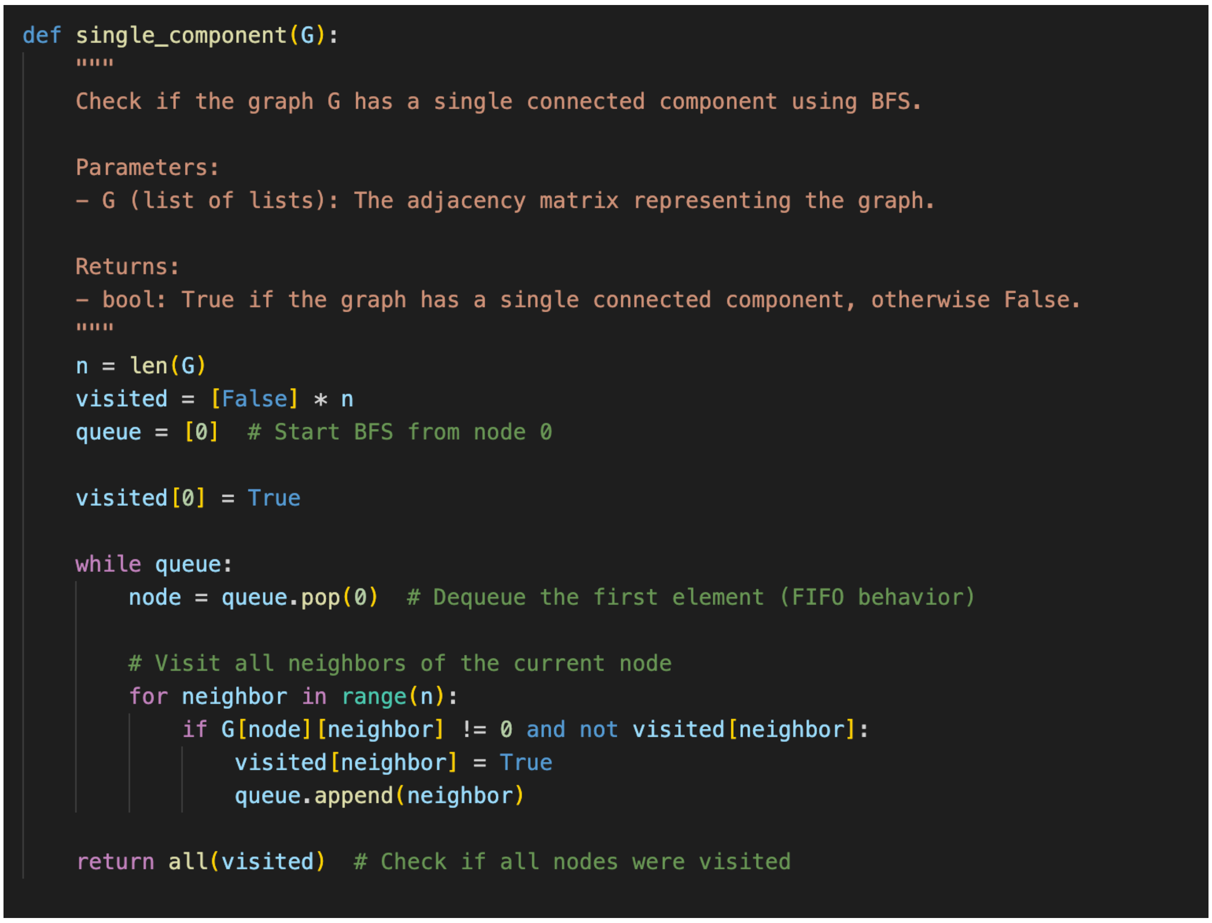
```python
def generate_graph(n=10, e=20, min_weight=1, max_weight=10, directed=True):
    """
    Generate a random graph with n nodes and e edges. The graph can be directed or undirected.

    Parameters:
    - n (int): The number of nodes.
    - e (int): The number of edges.
    - min_weight (int): The minimum edge weight.
    - max_weight (int): The maximum edge weight.
    - directed (bool): Whether the graph is directed (default is True).

    Returns:
    - List of lists: The adjacency matrix representing the graph.
    """
    G = generate_empty_graph(n)  # Start with an empty graph

    edges_set = set()  # To track existing edges and avoid duplicates

    for _ in range(e):
        s, d = nodes_selection(G, directed)

        # Ensure no duplicate edge exists
        while (s, d) in edges_set or (d, s) in edges_set:
            s, d = nodes_selection(G, directed)

        # Assign a random weight to the edge
        w = random.randint(min_weight, max_weight)

        create_edge(G, s, d, w)
        edges_set.add((s, d))  # Add directed edge to the set

        if not directed:
            create_edge(G, d, s, w)  # For undirected graphs, add reverse edge
            edges_set.add((d, s))  # Add reverse directed edge to the set

    return G
```

Purpose: This function generates a graph with n nodes and e edges, where each edge has a random weight between min_weight and max_weight. It can create either a directed or undirected graph based on the directed parameter.
How it works:

1. **Initialize the Graph**: It starts by creating an empty graph with n unconnected nodes using the generate_empty_graph function.

2. **Add Edges**:
   - For each of the e edges to be added:
     - It randomly selects a weight w within the specified range (min_weight to max_weight).
     - It calls nodes_selection to choose two nodes s and d for the new edge.
     - The while loop checks if an edge between s and d already exists (using get_edge). If an edge exists, it selects two new nodes to avoid duplicating edges.
     - After ensuring the nodes are unique, it adds an edge from s to d with weight w by calling create_edge.
     - If the graph is undirected, it also adds the reverse edge from d to s with the same weight.
3. **Return the Graph**: Finally, the function returns the completed graph G with n nodes and e edges.

Single Component Function:



```python
def single_component(G):
    """
    Check if the graph G has a single connected component using BFS.

    Parameters:
    - G (list of lists): The adjacency matrix representing the graph.

    Returns:
    - bool: True if the graph has a single connected component, otherwise False.
    """
    n = len(G)
    visited = [False] * n
    queue = [0]  # Start BFS from node 0

    visited[0] = True

    while queue:
        node = queue.pop(0)  # Dequeue the first element (FIFO behavior)

        # Visit all neighbors of the current node
        for neighbor in range(n):
            if G[node][neighbor] != 0 and not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)

    return all(visited)  # Check if all nodes were visited
```

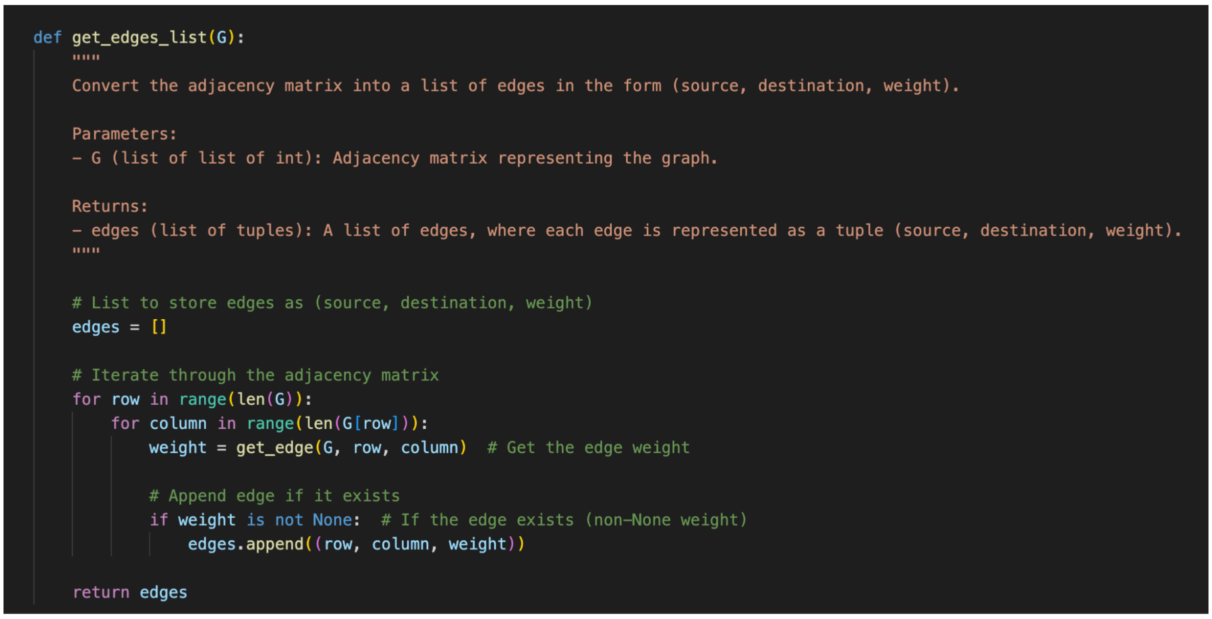
Purpose: This function checks if the graph G is a single connected component using Breadth-First Search (BFS). A single connected component means that all nodes are reachable from any other node in the graph.

How it works:

1. **Initialize Variables**:
    - n is set to the number of nodes in G.
    - visited is a list initialized to False for each node, indicating that no nodes have been visited initially.
    - queue is initialized with [0], meaning the BFS traversal will start from node 0.
2. **BFS Traversal**:
    - The function sets visited[0] = True to mark the starting node as visited.
    - While there are nodes in the queue, it:
        - Dequeues the first node (node = queue.pop(0)).
        - Iterates over all potential neighbors of the current node. For each neighbor:
            - If there is an edge between node and neighbor (G[node][neighbor] != 0) and neighbor has not been visited, it marks neighbor as visited and adds it to the queue.
3. **Final Check**:
    - After the BFS traversal, all(visited) checks if every node in the graph was visited.
    - If all nodes were visited, the function returns True, indicating the graph is a single connected component.
    - If any node was not visited, it returns False, meaning the graph is not fully connected.

GRAPHS TEMPLATE 2

Get Edges List Function:

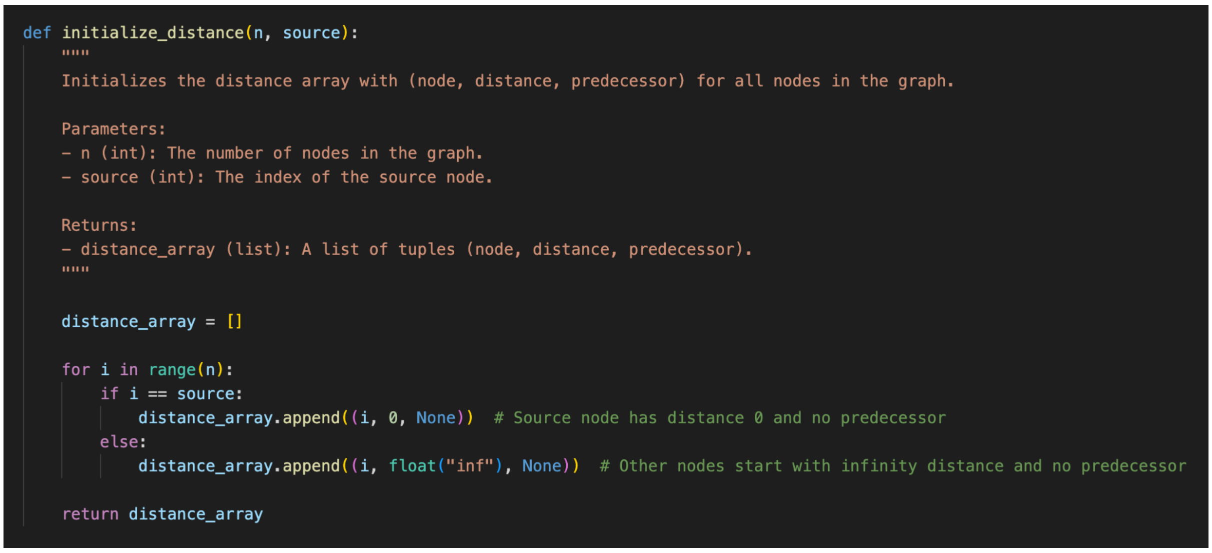


```python
def get_edges_list(G):
    """
    Convert the adjacency matrix into a list of edges in the form (source, destination, weight).

    Parameters:
    - G (list of list of int): Adjacency matrix representing the graph.

    Returns:
    - edges (list of tuples): A list of edges, where each edge is represented as a tuple (source, destination, weight).
    """

    # List to store edges as (source, destination, weight)
    edges = []

    # Iterate through the adjacency matrix
    for row in range(len(G)):
        for column in range(len(G[row])):
            weight = get_edge(G, row, column)  # Get the edge weight

            # Append edge if it exists
            if weight is not None:  # If the edge exists (non-None weight)
                edges.append((row, column, weight))

    return edges
```

Purpose: This function generates a list of all edges in the graph G, where each edge is represented as a tuple (source, destination, weight).

- **How it works**:
    - It initializes an empty list edges to store the edges of the graph.
    - The function uses nested loops to iterate through each pair of nodes (row, column) in the adjacency matrix G.
    - get_edge(G, row, column) is called to retrieve the weight of the edge between row and column.
    - If there is an edge (i.e., weight is not None), it appends a tuple (row, column, weight) to the edges list.
    - After all edges are processed, it returns the edges list.

Initialize Distance Function:

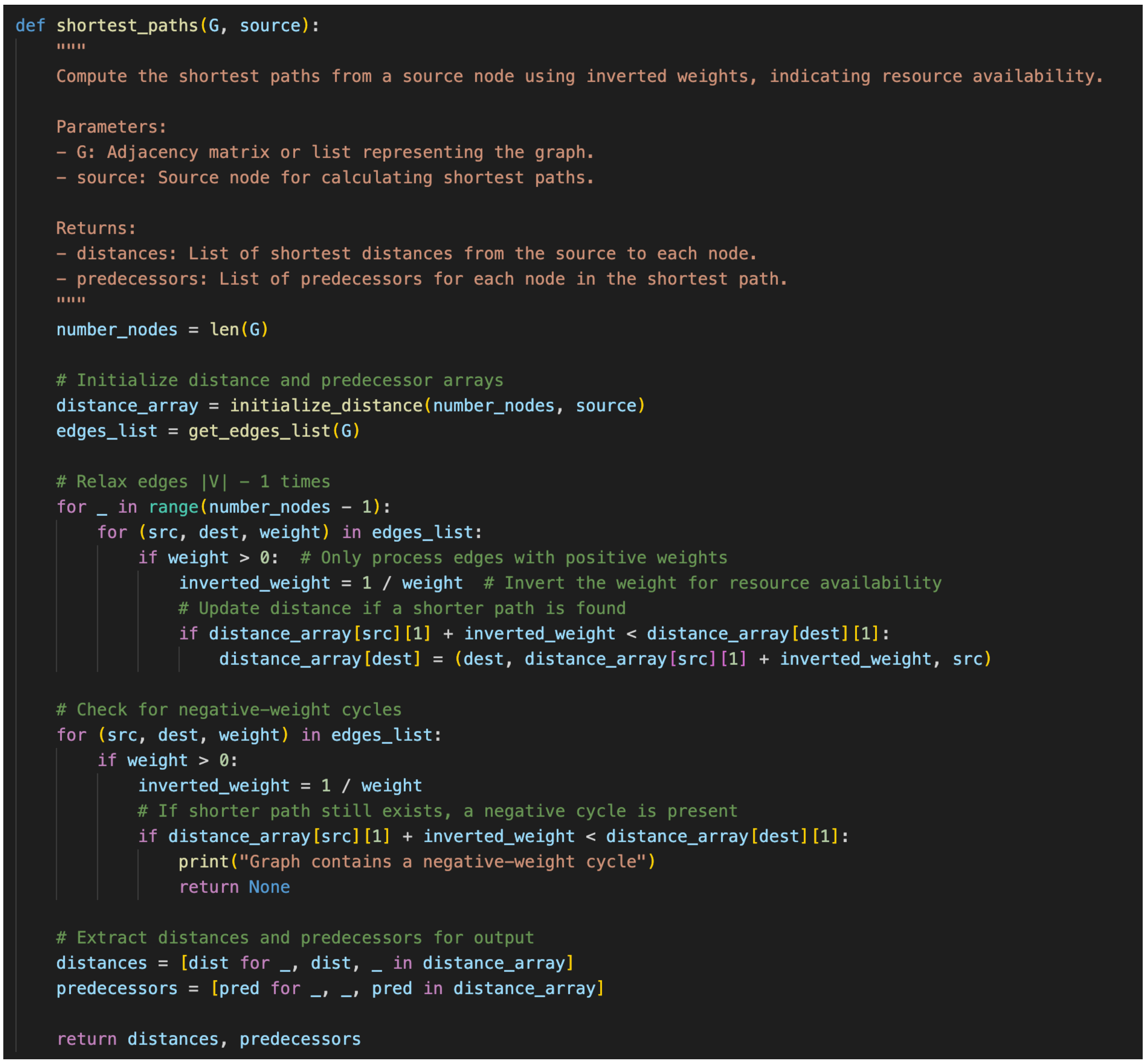


```python
def initialize_distance(n, source):
    """
    Initializes the distance array with (node, distance, predecessor) for all nodes in the graph.

    Parameters:
    - n (int): The number of nodes in the graph.
    - source (int): The index of the source node.

    Returns:
    - distance_array (list): A list of tuples (node, distance, predecessor).
    """

    distance_array = []

    for i in range(n):
        if i == source:
            distance_array.append((i, 0, None))  # Source node has distance 0 and no predecessor
        else:
            distance_array.append((i, float("inf"), None))  # Other nodes start with infinity distance and no predecessor

    return distance_array
```

Purpose: This function initializes an array to store the distance and predecessor information for each node in the graph. It sets the distance to the source node as 0 and all other nodes as infinity (inf), indicating they are initially unreachable from the source.
How it works:

- distance_array is initialized as an empty list.
- A loop iterates over each node i in the graph (from 0 to n-1).
    - If i is the source node, it appends (i, 0, "-") to distance_array, indicating that the distance to the source itself is 0 and there is no predecessor (indicated by "-").
    - For all other nodes, it appends (i, float("inf"), "-"), meaning they start with an infinite distance and no predecessor.
- Finally, it returns the distance_array, which contains the initial distance and **predecessor values for each node.**

Shortest Paths Function:



```python
def shortest_paths(G, source):
    """
    Compute the shortest paths from a source node using inverted weights, indicating resource availability.

    Parameters:
    - G: Adjacency matrix or list representing the graph.
    - source: Source node for calculating shortest paths.

    Returns:
    - distances: List of shortest distances from the source to each node.
    - predecessors: List of predecessors for each node in the shortest path.
    """
    number_nodes = len(G)

    # Initialize distance and predecessor arrays
    distance_array = initialize_distance(number_nodes, source)
    edges_list = get_edges_list(G)

    # Relax edges |V| - 1 times
    for _ in range(number_nodes - 1):
        for (src, dest, weight) in edges_list:
            if weight > 0:  # Only process edges with positive weights
                inverted_weight = 1 / weight  # Invert the weight for resource availability
                # Update distance if a shorter path is found
                if distance_array[src][1] + inverted_weight < distance_array[dest][1]:
                    distance_array[dest] = (dest, distance_array[src][1] + inverted_weight, src)

    # Check for negative-weight cycles
    for (src, dest, weight) in edges_list:
        if weight > 0:
            inverted_weight = 1 / weight
            # If shorter path still exists, a negative cycle is present
            if distance_array[src][1] + inverted_weight < distance_array[dest][1]:
                print("Graph contains a negative-weight cycle")
                return None

    # Extract distances and predecessors for output
    distances = [dist for _, dist, _ in distance_array]
    predecessors = [pred for _, _, pred in distance_array]

    return distances, predecessors
```

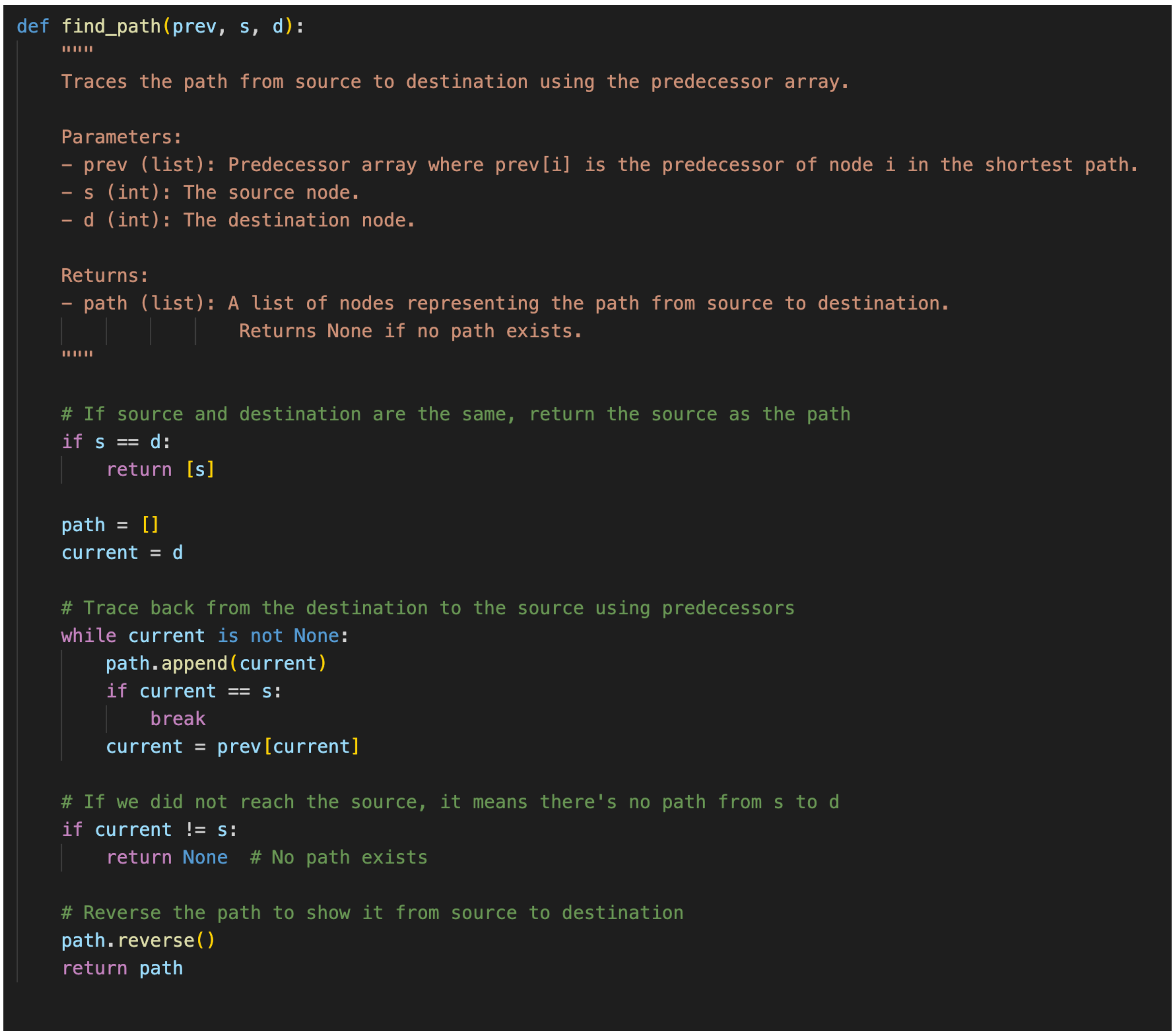
Purpose: This function calculates the shortest paths from a specified source node to all other nodes in the graph Gusing the Bellman-Ford algorithm. It returns the minimum distances and predecessors for each node.

How it works:

1. **Initialize Variables**:
    - number_nodes is set to the number of nodes in G.
    - distance_array is initialized with infinite distances (except for the source node, which is set to 0).
    - edges_list is created using get_edges_list(G), which provides a list of all edges.
2. **Relax Edges**:
    - The outer loop runs |V| - 1 times, where |V| is the number of nodes. This is a requirement of the Bellman-Ford algorithm to ensure that all shortest paths are found.

   - For each edge (src, dest, weight) in edges_list, it checks if going through src provides a shorter path to dest. If so, it updates distance_array[dest] with the new shorter distance and sets src as the predecessor.
3. **Check for Negative-Weight Cycles**:
   - After the relaxation process, it iterates over each edge one more time to detect any negative-weight cycles. If a shorter path is found during this check, it means there is a negative cycle, and the function returns None.
4. **Extract Distances and Predecessors**:
   - Finally, it extracts the distances and predecessors from distance_array into separate lists, distances and predecessors, for easier access and returns them.

Find Path Function:



```python
def find_path(prev, s, d):
    """
    Traces the path from source to destination using the predecessor array.

    Parameters:
    - prev (list): Predecessor array where prev[i] is the predecessor of node i in the shortest path.
    - s (int): The source node.
    - d (int): The destination node.

    Returns:
    - path (list): A list of nodes representing the path from source to destination.
                Returns None if no path exists.
    """

    # If source and destination are the same, return the source as the path
    if s == d:
        return [s]

    path = []
    current = d

    # Trace back from the destination to the source using predecessors
    while current is not None:
        path.append(current)
        if current == s:
            break
        current = prev[current]

    # If we did not reach the source, it means there's no path from s to d
    if current != s:
        return None  # No path exists

    # Reverse the path to show it from source to destination
    path.reverse()
    return path
```

Purpose: This function recovers the shortest path from a source node s to a destination node d using the predecessor array prev, which is generated by a shortest-path algorithm (such as Bellman-Ford). It returns the path as a list of nodes from s to d.

How it works:

1. **Initialize Variables**:
   - path is initialized as an empty list to store the path from d back to s.
   - current is set to d (the destination node), and we'll trace back from d to s using the predecessor array.
2. **Trace the Path**:
   - While current is not None, the loop adds current to the path list and then updates current to its predecessor (prev[current]).
   - If current reaches the source node s, it breaks out of the loop, indicating that the full path has been traced back successfully.
3. **Check for No Path**:
   - After the loop, if current is not equal to s, it means there was no valid path from s to d, and the function returns None.
4. **Reverse the Path**:
   - If a path is found, path.reverse() is called to reverse the list, displaying the path from s to d.
5. **Return the Path**:
   - The function returns the path list, which represents the shortest path from s to d.

GRAPHS 3 TEMPLATE:

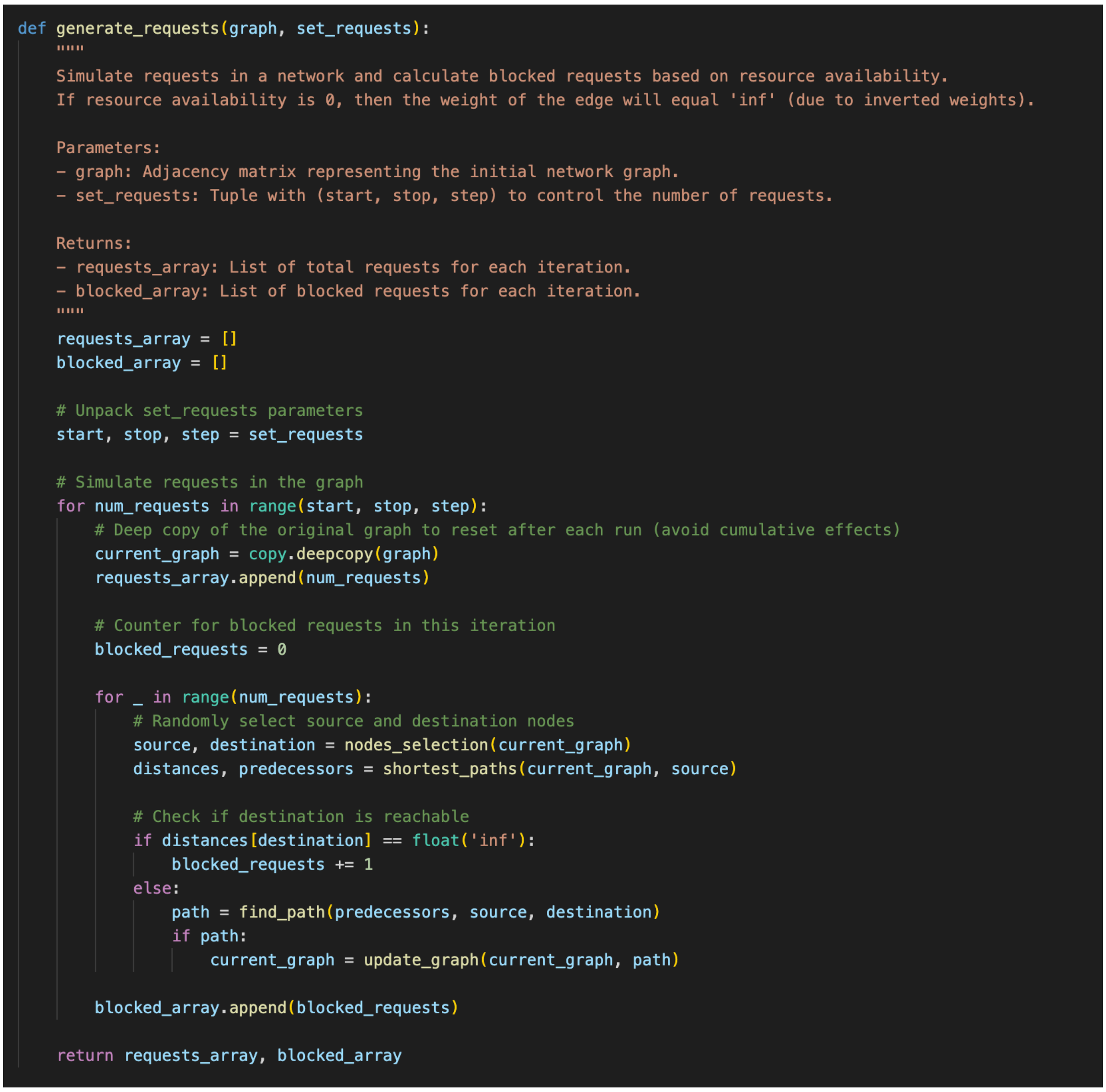
Convert Adjacency List to Adjacency Matrix Function:

```python
def convert_adjacency_list_to_adjacency_matrix(graphL):
    """
    Converts an adjacency list representation of a graph to an adjacency matrix.

    Parameters:
    - graphL (list of list of tuples): The adjacency list where each element at index `i`
      contains a list of tuples `(dest, weight)` representing edges from node `i`
      to node `dest` with the given `weight`.

    Returns:
    - adj_matrix (list of list of int): The adjacency matrix where `adj_matrix[i][j]`
      holds the weight of the edge from node `i` to node `j`, or 0 if no edge exists.
    """

    # Initialize an adjacency matrix with zeros
    num_nodes = len(graphL)
    adj_matrix = generate_empty_graph(num_nodes)

    # Populate the adjacency matrix from the adjacency list
    for node, edges in enumerate(graphL):
        for edge in edges:
            try:
                dest, weight = edge
                adj_matrix[node][dest - 1] = weight  # Adjust for 0-based indexing if necessary
            except IndexError:
                print(f"Error: Node {dest} in adjacency list is out of range for graph size {num_nodes}")
                raise

    return adj_matrix
```

Purpose: This function converts a graph from an adjacency list representation to an adjacency matrix representation.
How it works:

1. **Initialize the Matrix**:
    - adj_matrix is initialized as a square matrix of size n x n, where n is the number of nodes in the graph (len(graphL)). It's initially filled with zeros, meaning there are no edges between any nodes.
2. **Populate the Matrix**:
    - The function iterates over each node and its associated edges in the adjacency list graphL.
    - For each edge, it extracts dest (the destination node) and weight (the edge's weight).
    - It then sets adj_matrix[node][dest - 1] = weight to add the edge weight to the matrix. The - 1adjustment ensures correct indexing if nodes are 1-based in the adjacency list, converting them to 0-based indexing for the matrix.
3. **Return the Matrix**:
    - After populating all edges, the function returns adj_matrix, which represents the graph in matrix form.

Generate Requests Function:



```python
def generate_requests(graph, set_requests):
    """
    Simulate requests in a network and calculate blocked requests based on resource availability.
    If resource availability is 0, then the weight of the edge will equal 'inf' (due to inverted weights).

    Parameters:
    - graph: Adjacency matrix representing the initial network graph.
    - set_requests: Tuple with (start, stop, step) to control the number of requests.

    Returns:
    - requests_array: List of total requests for each iteration.
    - blocked_array: List of blocked requests for each iteration.
    """
    requests_array = []
    blocked_array = []

    # Unpack set_requests parameters
    start, stop, step = set_requests

    # Simulate requests in the graph
    for num_requests in range(start, stop, step):
        # Deep copy of the original graph to reset after each run (avoid cumulative effects)
        current_graph = copy.deepcopy(graph)
        requests_array.append(num_requests)

        # Counter for blocked requests in this iteration
        blocked_requests = 0

        for _ in range(num_requests):
            # Randomly select source and destination nodes
            source, destination = nodes_selection(current_graph)
            distances, predecessors = shortest_paths(current_graph, source)

            # Check if destination is reachable
            if distances[destination] == float('inf'):
                blocked_requests += 1
            else:
                path = find_path(predecessors, source, destination)
                if path:
                    current_graph = update_graph(current_graph, path)

        blocked_array.append(blocked_requests)

    return requests_array, blocked_array
```

Purpose: This function simulates multiple connection requests in a communication network, attempting to allocate paths between randomly chosen nodes. It tracks how many requests are blocked due to insufficient resources.

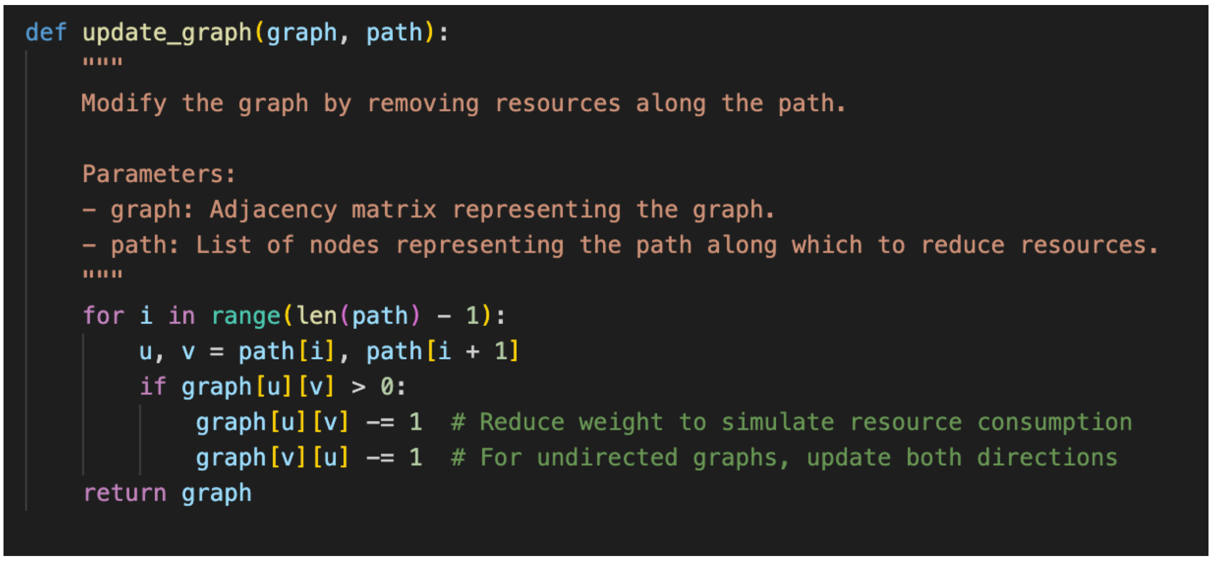
How it works:

1. **Initialize Arrays**:
    - requests_array keeps track of the total number of requests in each simulation run.
    - blocked_array records the count of blocked requests for each run.
    - start_graph is a deep copy of the initial graph, preserving its state for resetting after each run.
2. **Simulate Requests**:

- For each run (with an increasing number of requests as defined by set_requests):
  - num_requests is appended to requests_array.
  - blocked_requests is initialized to 0 to count blocked requests for the current run.
  - A loop iterates over num_requests, simulating each connection request:
    - Two nodes (source and destination) are selected randomly as the endpoints.
    - shortest_paths(graph, source) calculates the shortest paths from the source node.
    - If distances[destination] is inf, it means no path is available, so blocked_requests is incremented.
    - If a path exists, find_path(predecessors, source, destination) retrieves it, and update_graph(graph, path) updates the graph to reflect reduced resources along this path.

3. **Track Blocked Requests**:
   - After each run, the blocked request count for that run is appended to blocked_array.
   - The graph is reset to its initial state (start_graph) for the next simulation run.
4. **Return Results**:
   - The function returns requests_array and blocked_array, which store the total and blocked requests for each run.

Update Graph Function:



```python
def update_graph(graph, path):
    """
    Modify the graph by removing resources along the path.

    Parameters:
    - graph: Adjacency matrix representing the graph.
    - path: List of nodes representing the path along which to reduce resources.
    """
    for i in range(len(path) - 1):
        u, v = path[i], path[i + 1]
        if graph[u][v] > 0:
            graph[u][v] -= 1  # Reduce weight to simulate resource consumption
            graph[v][u] -= 1  # For undirected graphs, update both directions
    return graph
```

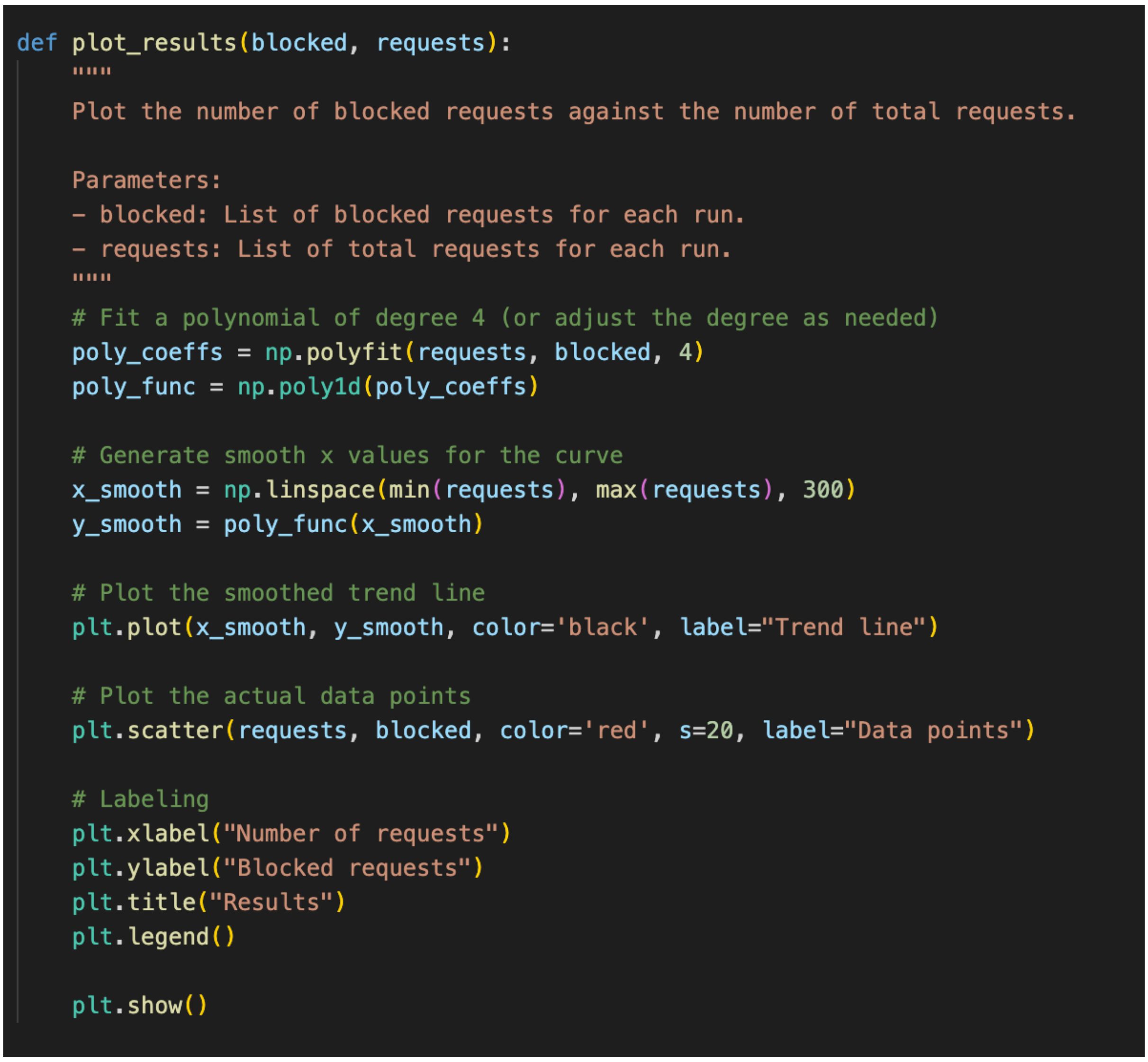
Purpose: This function updates the graph to reflect the consumption of resources along a given path by decreasing the weights of the edges in the path. It simulates a reduction in available resources on each link used.
How it works:

1. **Loop Through the Path**:
   - It iterates over each consecutive pair of nodes (u, v) in the path.
   - u is the current node, and v is the next node in the path.
2. **Reduce Resources**:

- ○ If graph[u][v] > 0, it decreases the weight of the edge from u to v by 1, simulating the use of one unit of resource.
- ○ For undirected graphs, it also decreases graph[v][u] by 1 to ensure symmetry (both directions are updated).

3. **Return the Updated Graph**:
    - ○ After updating the edges along the entire path, the function returns the modified graph.

Plot Results Function:



```python
def plot_results(blocked, requests):
    """
    Plot the number of blocked requests against the number of total requests.

    Parameters:
    - blocked: List of blocked requests for each run.
    - requests: List of total requests for each run.
    """
    # Fit a polynomial of degree 4 (or adjust the degree as needed)
    poly_coeffs = np.polyfit(requests, blocked, 4)
    poly_func = np.poly1d(poly_coeffs)

    # Generate smooth x values for the curve
    x_smooth = np.linspace(min(requests), max(requests), 300)
    y_smooth = poly_func(x_smooth)

    # Plot the smoothed trend line
    plt.plot(x_smooth, y_smooth, color='black', label="Trend line")

    # Plot the actual data points
    plt.scatter(requests, blocked, color='red', s=20, label="Data points")

    # Labeling
    plt.xlabel("Number of requests")
    plt.ylabel("Blocked requests")
    plt.title("Results")
    plt.legend()

    plt.show()
```

Purpose: This function generates a plot to visualize the relationship between the total number of requests and the number of blocked requests across multiple simulation runs. It includes both actual data points and a trend line for clearer analysis.

How it works:

1. **Fit a Polynomial Trend Line**:
    - np.polyfit(requests, blocked, deg=4) calculates the coefficients for a polynomial of degree 4 that fits the blocked data points against requests. This polynomial provides a smoothed trend line.
    - poly_func = np.poly1d(poly_coeffs) creates a polynomial function using the calculated coefficients, which can be used to generate y-values for the trend line.
2. **Generate Smooth Data for the Trend Line**:
    - x_smooth is generated using np.linspace, providing a smooth set of x-values from the minimum to the maximum number of requests.
    - y_smooth is computed by applying poly_func to x_smooth, generating the y-values for the trend line.
3. **Plot the Trend Line and Data Points**:
    - plt.plot(x_smooth, y_smooth, color='black', label="Trend line") plots the smoothed polynomial trend line in black.
    - plt.scatter(requests, blocked, color='red', s=20, label="Data points") plots the actual data points in red to show the raw blocked request counts for each run.
4. **Labeling and Display**:
    - plt.xlabel and plt.ylabel set the axis labels.
    - plt.title sets the plot title.
    - plt.legend() adds a legend for clarity.
    - plt.show() displays the final plot.

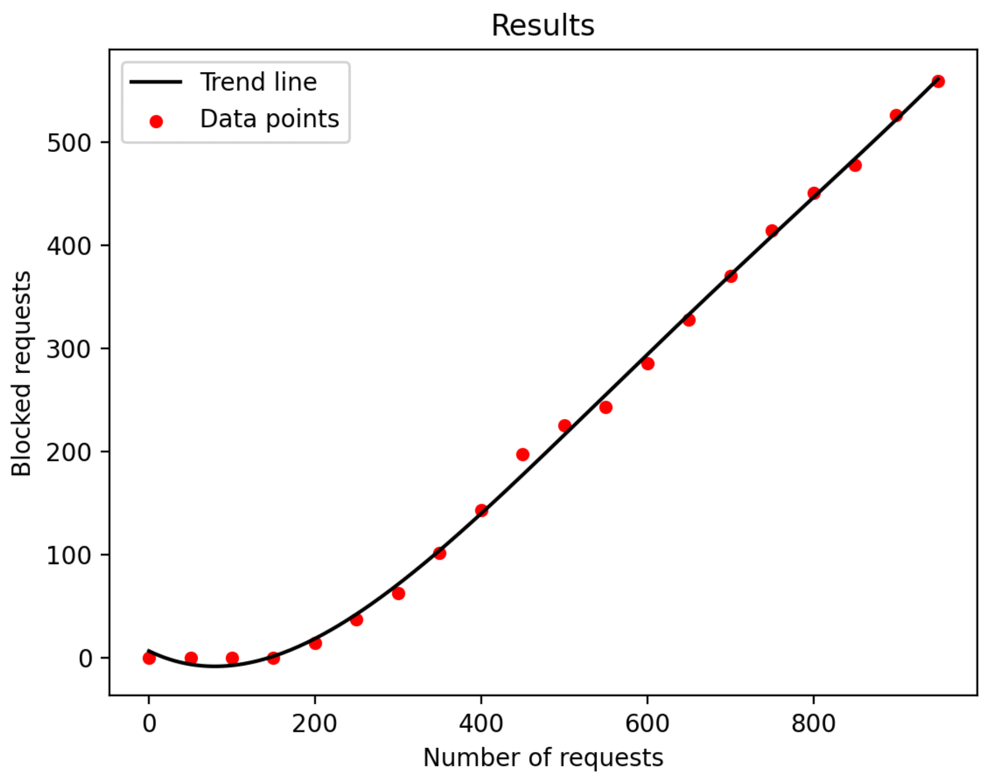
Graph

This graph illustrates the relationship between the number of requests and the blocked requests, showing how the network performs as demand increases.

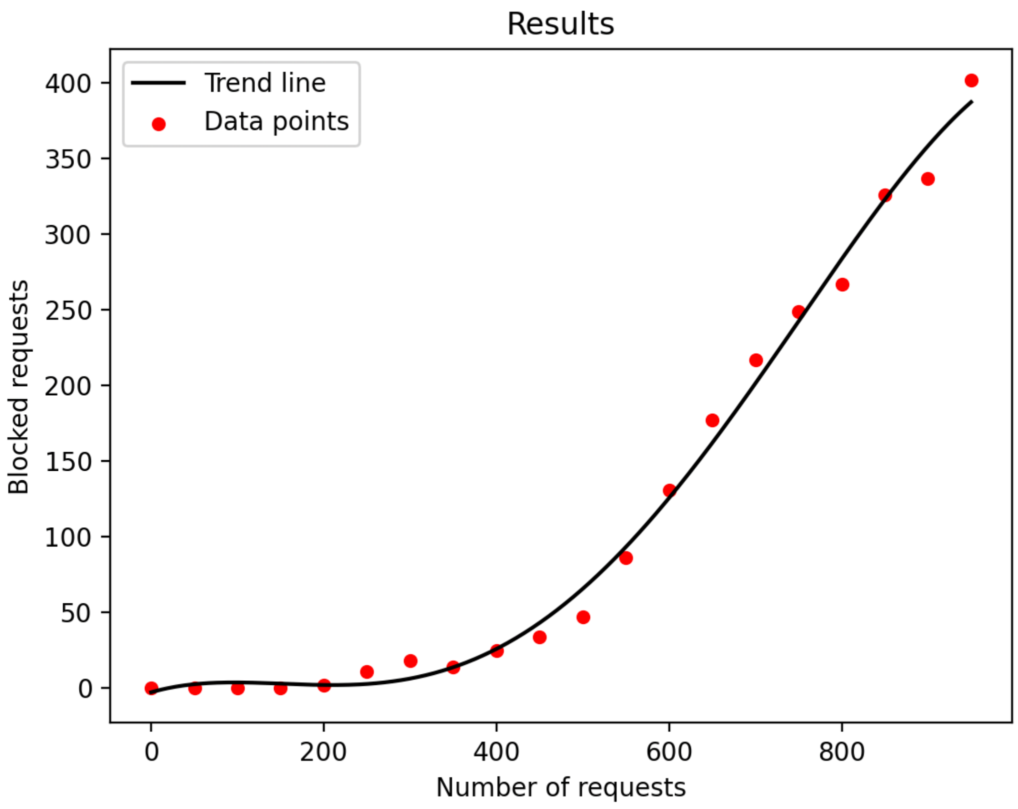
- **Initial Stability**: At low request levels, blocked requests are minimal, indicating that the network can handle light loads efficiently.
- **Gradual Increase**: As demand grows, blocked requests start to rise gradually. This suggests that the network is beginning to reach its capacity, with resources becoming less readily available.
- **Sharp Increase**: With further increases in demand, blocked requests rise more sharply. This upward trend shows the network approaching its limits, where additional requests quickly lead to resource exhaustion.

Insights

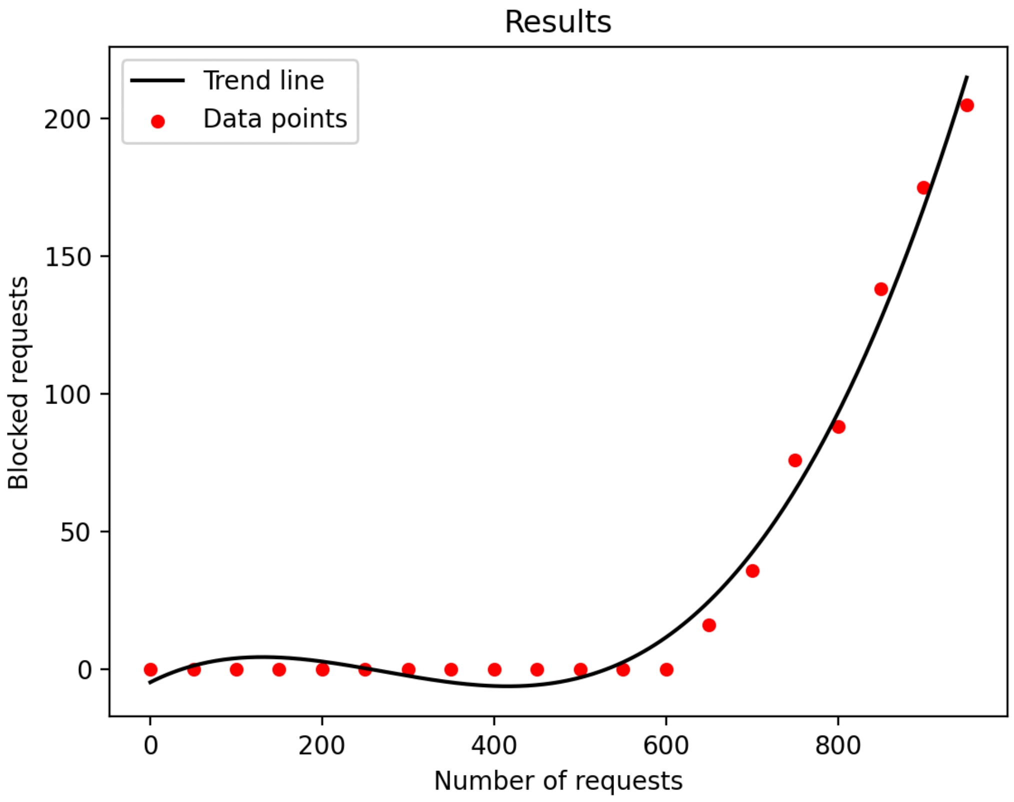
- **Capacity Threshold**: The rapid increase in blocked requests at higher demand levels indicates a capacity threshold. To handle heavier loads, expanding resources or implementing load-balancing solutions would be beneficial.
- **System Efficiency**: The curve highlights a tipping point where the network shifts from effectively handling requests to rapid saturation. This suggests the network is efficient under moderate demand but faces challenges as load intensifies.



INT9



NSFnet



Telefonica

Self-Assessment

o Josep Cubedo: 10/10
o Julian Harder: 10/10
o Teo Arqués: 10/10
o Gerard Figueras: 10/10

We all worked together on the W1, W2, and W3 templates, with everyone contributing equally to the coding and testing. We also collaborated on the documentation, making sure everything was clear and accurate as a team.