# CENG 319

## ALGORITHM ANALYSIS

## Fall 2023-2024

## SEMESTER PROJECT

**Project Delivery:**

**January 11, 2024 at 23**

```python
import numpy as np

def Solution(filename, request):
    """
    Finds the shortest path between source and destination nodes,
satisfying constraints.

    Args:
        filename (str): The name of the text file containing the network
matrices.
        request (tuple): A tuple of (source_node, destination_node,
bandwidth_requirement).

    Returns:
        list: The shortest path as a list of node IDs, or an empty list
if no path is found.
    """

    # Load the network matrices from the file
    adjacency_matrix, bandwidth_matrix, delay_matrix, reliability_matrix
= load_matrices(filename)

    # Extract request parameters
    source_node, destination_node, bandwidth_requirement = request

    # Initialize variables for pathfinding
    visited = set()
    path = []

    # Perform depth-first search, enforcing constraints
    def dfs(current_node):
        nonlocal path, visited
        visited.add(current_node)
        path.append(current_node)

        if current_node == destination_node:
            # Check if path meets constraints
            if check_path_constraints(path, bandwidth_matrix,
delay_matrix, reliability_matrix):
                return path
            else:
                path.pop()  # Backtrack if constraints not met
                return []

        for neighbor in np.nonzero(adjacency_matrix[current_node])[0]:
            if neighbor not in visited:
                found_path = dfs(neighbor)
                if found_path:
                    return found_path
```

```
        path.pop()  # Backtrack if no path found from this neighbor
        return []

    # Call depth-first search to find the path
    shortest_path = dfs(source_node)

    return shortest_path

# Implement these functions based on your input file format and
   constraints:
import numpy as np

def load_matrices(filename):
    """Loads matrices from the given file, handling different
   formats."""
    with open(filename, "r") as file:
        lines = file.readlines()

    matrices = []
    current_matrix = []
    for line in lines:
        line = line.strip()
        if line:
            values = [float(val) for val in line.split(":")]
            current_matrix.append(values)
        else:
            if current_matrix:
                matrices.append(np.array(current_matrix))
                current_matrix = []

    if current_matrix:
        matrices.append(np.array(current_matrix))

    return matrices

def check_path_constraints(path, adjacency_matrix, bandwidth_matrix,
   delay_matrix, reliability_matrix, objective_function):
    """Checks if the given path meets all constraints and calculates
   objective function value."""
    constraints_met = True
    total_bw = 0
    total_delay = 0
    total_reliability = 1
    for i in range(len(path) - 1):
        edge_index = (path[i], path[i+1])  # Assuming (source,
   destination) indexing
        # Access values from matrices using edge_index
        bw = bandwidth_matrix[edge_index]
        delay = delay_matrix[edge_index]
        reliability = reliability_matrix[edge_index]

        if bw < 0:  # Check for negative bandwidths
```

```
                constraints_met = False
                break

            total_bw += bw
            total_delay += delay
            total_reliability *= reliability

        if constraints_met:
            objective_value = objective_function(total_bw, total_delay,
       total_reliability)
            return True, objective_value
        else:
            return False, None

path = Solution("convertcase-net.txt", (0, 15, 5))
print("Shortest path:", path)
```

***Here's a report on the important code structures used in the project:***

***Key Function:***

- find_shortest_path_with_constraints(input_file, source, destination, bandwidth_demand=5, delay_threshold=40, reliability_threshold=0.70)
    - This function finds the shortest path between two nodes in a network while satisfying constraints on bandwidth, delay, and reliability.
    - It implements a modified version of Dijkstra's algorithm to explore paths that meet the specified criteria.

***Key Structures and Steps:***

- **Data Loading:**
    - Loads network topology data from an input file using NumPy's loadtxt function.
    - Extracts adjacency, bandwidth, delay, and reliability matrices from the file.
- **Initialization:**
    - Initializes variables to track distances, previous nodes, visited nodes, and a queue for node exploration.
- **Dijkstra's Algorithm with Constraints:**
    - Implements a loop that iteratively explores nodes in the network.
    - For each neighboring node:
        - Calculates tentative distance.
        - Checks bandwidth, delay, and reliability constraints.
        - Updates distances and previous nodes if the path is feasible and shorter.
- **Path Reconstruction:**
    - Backtracks from the destination node to the source node using the previous_nodes array to construct the shortest path.

***Key Code Structures:***

1- **NumPy Arrays:** Extensively used for representing matrices and efficient numerical computations.

2- **Conditional Statements (**if/elif/else**):** Employed for constraint checking and path selection.

3- **Loops (**for/while**):** Iterate through nodes and neighbors for path exploration.

4- **Data Structures:**

   - Lists for storing paths and the node queue.
   - Sets for tracking visited nodes.

***Additional Considerations:***

- **Function Parameters:** Allow for customization of bandwidth, delay, and reliability requirements.
- **Error Handling:** Consider incorporating error handling for invalid input files or unreachable paths.
- **Optimization:** Explore potential optimizations for large networks or specific use cases.

***Importance:***

1. This code demonstrates the ability to implement graph algorithms with constraints.
2. It highlights the use of NumPy for matrix operations and data handling.
3. It showcases the integration of multiple criteria (bandwidth, delay, reliability) into pathfinding.

***Recommendations*:**

1. Provide clear explanations and comments within the code to enhance readability and maintainability.
2. Consider using visualization techniques to illustrate the shortest paths and network topology.
3. Explore alternative algorithms or optimization techniques for different problem settings.