PES UNIVERSITY

EC Campus, Bengaluru

Department of Computer Science & Engineering

COMPUTER NETWORKS - UE21CS252B

5th Semester – E Section

ASSIGNMENT – 1

Energy Efficient Multi Path Routing Algorithm

Submitted to:

Dr. Geetha D
Associate Professor

Submitted By:

Name : Maryam Khan SRN: PES2UG21CS283
Name : Melvin A Gomez SRN: PES2UG21CS293
Name : Manasvi Varma SRN: PES2UG21CS305

Table of Contents

1. Abstract and Scope of the Project

Abstract:

This project presents an implementation of the Energy Efficient Multi-Path Routing Algorithm, which finds up to a specified number of energy-efficient paths in a network topology represented as a graph. The algorithm takes the source and destination nodes, battery capacity of each node, threshold value for the remaining battery capacity below which a node is considered dead, and the maximum number of paths to return as inputs. The output is a list of up to num_paths paths from the source to the destination that avoid nodes with low battery capacity.

Scope:

The python code imports the networkx module, which is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. The algorithm defined in the code creates a copy of the original graph and adds reversed edges to it. It then finds the shortest path in the original graph using Dijkstra's algorithm and initializes a priority queue to store the paths to explore in the reversed graph. The code updates the remaining battery capacity of each node in the original graph and adds the path to the list of valid paths. Finally, it removes the edge from the reversed graph and returns the valid paths found so far. The algorithm is tested on a sample network topology with predefined battery capacities and threshold values. The code uses the print function to output the list of valid paths found by the algorithm.

2. Project Description

(Detailed explanation about project implementation and details of functions defined)

This project implements an Energy Efficient Multi-Path Routing Algorithm, which is used to find up to num_paths energy-efficient paths from a source node to a destination node in a given network topology represented as a graph G. The algorithm takes into consideration the battery capacity of each node, as well as a threshold value threshold for the remaining battery capacity below which a node is considered dead.

The algorithm first creates a copy of the original graph G and adds reversed edges to it. It then initializes a dictionary to store the remaining battery capacity of each node, finds the shortest path in the original graph G using Dijkstra's algorithm, and initializes a priority queue to store the paths to explore in the reversed graph.

The algorithm then enters a loop, in which it pops the path with the highest priority from the queue, checks if the path has already been explored, and if not, checks if the path avoids nodes with low battery capacity. If the path is valid, the remaining battery capacity of each node in the original graph is updated, the path is added to the list of valid paths, and the priority queue is updated with new paths to explore.

The algorithm returns the valid paths.

The Energy_Efficient_Multipath_Routing Function Takes In The Following Parameters:

G: a network topology represented as a graph
source: the source node
dest: the destination node
battery_capacity: a dictionary mapping each node to its battery capacity
threshold: the threshold value for the remaining battery capacity below which a node is considered dead
num_paths: the maximum number of paths to return

Details Of Functions:

networkx: This is a Python library used to work with graphs and networks, and is used in the code to create, manipulate and traverse the graph.

heapq: This is a Python library used to implement heap-based priority queues, and is used in the code to implement the priority queue to explore paths in the reversed graph.

copy: This is a Python library used to create copies of objects, and is used in the code to create a deep copy of the battery capacity dictionary.

energy_efficient_multipath_routing: This is a function that implements an energy-efficient multi-path routing algorithm. It takes as input a graph G, the source and destination nodes source and dest, a dictionary battery_capacity mapping each node to its battery capacity, a threshold value threshold for the remaining battery capacity below which a node is considered dead, and an optional parameter num_paths specifying the maximum number of paths to return. It returns a list of up to num_paths paths from the source to the destination that avoid nodes with low battery capacity.

G.to_directed().reverse(copy=True): This creates a copy of the original graph G, converts it to a directed graph and reverses the direction of all edges, to create a reversed graph H used in the algorithm.

nx.dijkstra_path: This is a function from the networkx library used to find the shortest path in a graph between two nodes, based on Dijkstra's algorithm. It is used in the algorithm to find the shortest path in the original graph.

heapq.heappush: This is a function from the heapq library used to add a new element to the priority queue paths_to_explore with a given priority.

set: This is a Python data type used to store an unordered collection of unique elements, and is used in the algorithm to store the explored paths in the reversed graph.

valid_paths: This is a list used in the algorithm to store the valid paths found so far.

remaining_battery_capacity: This is a dictionary used in the algorithm to store the remaining battery capacity of each node in the original graph.

H.add_edges_from(): This is a function from the networkx library used to add a list of edges to the reversed graph H.

H.remove_edge(): This is a function from the networkx library used to remove an edge from the reversed graph H.

3. Software Requirements

(Description about Programming Languages, API's, Methods, etc.,)

Python is a popular, high-level programming language that is widely used in scientific computing, data analysis, and machine learning. It has a large and active community that provides many third-party libraries and tools for various domains.

NetworkX is a Python library for creating, manipulating, and studying complex networks or graphs. It provides data structures for graphs, algorithms for graph manipulation and analysis, and visualization tools. The library is used for a wide range of applications, including social network analysis, bioinformatics, and transportation network modeling.

The code defines a function energy_efficient_multipath_routing that takes as input a graph object created using NetworkX, along with other parameters such as the source and destination nodes, battery capacities, and threshold value. The function implements an energy-efficient multi-path routing algorithm that finds up to num_paths valid paths from the source to the destination while avoiding nodes with low battery capacity.

The algorithm uses **Dijkstra's shortest path algorithm** from the source to the destination node to find the initial shortest path in the original graph. It then explores the reversed graph using a priority queue to find alternative paths that avoid nodes with low battery capacity. The algorithm updates the remaining battery capacity of each node and checks if the path is valid before adding it to the list of valid paths.

Overall, the project requires Python 3.x and the NetworkX library to be installed on the system.

4. Source Code

```
import networkx as nx
import heapq
import copy

def energy_efficient_multipath_routing(G, source, dest, battery_capacity, threshold,
num_paths=3):

    H = G.to_directed().reverse(copy=True)

    H.add_edges_from((v, u, H[u][v]) for u, v in H.edges())

    remaining_battery_capacity = copy.deepcopy(battery_capacity)

    shortest_path = nx.dijkstra_path(G, source, dest, weight='weight')


    paths_to_explore = []
    heapq.heappush(paths_to_explore, (-len(shortest_path), shortest_path))

    explored_paths = set()


    valid_paths = []

    while paths_to_explore and len(valid_paths) < num_paths:


        _, path = heapq.heappop(paths_to_explore)


        if tuple(path) in explored_paths:
            continue


        explored_paths.add(tuple(path))


        valid = True
        for node in path:
            if remaining_battery_capacity[node] < threshold:
                valid = False
                break

        if valid:
```

```
            for node in G.nodes:
                remaining_battery_capacity[node] -= threshold


            valid_paths.append(path)


            for i in range(len(path) - 1):
                u = path[i]
                v = path[i+1]
                if remaining_battery_capacity[v] >= threshold:

                    H.add_edge(v, u, **G[u][v])

                    shortest_path_v_u = nx.dijkstra_path(H, v, u, weight='weight')

                    heapq.heappush(paths_to_explore, (-len(shortest_path_v_u), shortest_path_v_u))

                    H.remove_edge(v, u)

    return valid_paths


G = nx.Graph()
G.add_edges_from([(1, 2, {'weight': 10}), (1, 3, {'weight': 5}), (2, 3, {'weight': 3}),
                  (2, 4, {'weight': 2}), (3, 4, {'weight': 7}), (3, 5, {'weight': 5}),
                  (4, 5, {'weight': 10}), (4, 6, {'weight': 1}), (5, 6, {'weight': 2})])


battery_capacity = {1: 100, 2: 80, 3: 70, 4: 50, 5: 30, 6: 10}

threshold = 20

source = 1
dest = 4

paths = energy_efficient_multipath_routing(G, source, dest, battery_capacity, threshold,
num_paths=3)

print(paths)
```

5. Sample Output

(Include necessary output screenshots followed by brief description)

We have considered a graph having edges as shown in the below screenshot.
We consider the source = 1 and destination = 4

```python
G = nx.Graph()
G.add_edges_from([(1, 2, {'weight': 10}), (1, 3, {'weight': 5}), (2, 3, {'weight': 3}),
                  (2, 4, {'weight': 2}), (3, 4, {'weight': 7}), (3, 5, {'weight': 5}),
                  (4, 5, {'weight': 10}), (4, 6, {'weight': 1}), (5, 6, {'weight': 2})])

# Define the battery capacity of each node
battery_capacity = {1: 100, 2: 80, 3: 70, 4: 50, 5: 30, 6: 10}

# Set the threshold value for the remaining battery capacity
threshold = 20

# Define the source and destination nodes
source = 1
dest = 4

# Run the Energy Efficient Multi-Path Routing Algorithm
paths = energy_efficient_multipath_routing(G, source, dest, battery_capacity, threshold, num_paths=3)

# Print the paths
print(paths)
```

```
PROBLEMS 12  OUTPUT  DEBUG CONSOLE  TERMINAL

(base) maryamkhan@Maryams-MacBook-Air Assignmnent 2 % python energy.py
[[1, 3, 2, 4], [2, 3], [3, 1]]
(base) maryamkhan@Maryams-MacBook-Air Assignmnent 2 %
```

Taking the source = 1 and destination = 5

```python
G = nx.Graph()
G.add_edges_from([(1, 2, {'weight': 10}), (1, 3, {'weight': 5}), (2, 3, {'weight': 3}),
                  (2, 4, {'weight': 2}), (3, 4, {'weight': 7}), (3, 5, {'weight': 5}),
                  (4, 5, {'weight': 10}), (4, 6, {'weight': 1}), (5, 6, {'weight': 2})])

# Define the battery capacity of each node
battery_capacity = {1: 100, 2: 80, 3: 70, 4: 50, 5: 30, 6: 10}

# Set the threshold value for the remaining battery capacity
threshold = 20

# Define the source and destination nodes
source = 1
dest = 5

# Run the Energy Efficient Multi-Path Routing Algorithm
paths = energy_efficient_multipath_routing(G, source, dest, battery_capacity, threshold, num_paths=3)

# Print the paths
print(paths)
```

```
PROBLEMS 12  OUTPUT  DEBUG CONSOLE  TERMINAL

(base) maryamkhan@Maryams-MacBook-Air Assignmnent 2 % python energy.py
[[1, 3, 5], [3, 1], [1, 3]]
(base) maryamkhan@Maryams-MacBook-Air Assignmnent 2 %
```

6. Conclusion

In conclusion, this project implements an Energy Efficient Multi-Path Routing Algorithm for a network topology represented as a graph using the NetworkX library in Python. The algorithm finds up to a specified number of paths from a source node to a destination node that avoid nodes with low battery capacity, with the aim of prolonging the network lifetime. The implementation involves creating a copy of the original graph and adding reversed edges to it, using Dijkstra's algorithm to find the shortest path in the original graph, and exploring paths in the reversed graph to find valid paths.

7. References

- https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
- https://www.geeksforgeeks.org/networkx-python-software-package-study-complex-networks/
- https://docs.python.org/3/library/heapq.html