DESIGN AND ANALYSIS OF ALGORITHM PROJECT

NETWORK FLOW OPTIMIZATION – ROUTING INTERNET TRAFFIC USING DIJKSTRA ALOGRITHM (GREEDY METHOD)

CSE SWE- X2

Team Members

Vansh Jain
RA2111033010028

Natasha Kumari
RA2111033010065

CONTENTS

CONTRIBUTION TABLE		
S.no	**Name**	**Contribution**
1.	Natasha & Vansh	Problem Definition
2.	Natasha	Dijkstra problem explanation
3.	Vansh	Design Technique Used: Greedy Method
4.	Natasha	Explanation of algorithm with example
5.	Natasha & Vansh	C++ code for the problem
6.	Vansh	Complexity Analysis
7.	Natasha	Documentation

PROBLEM DEFINITION

Given a network topology, with nodes representing routers and links representing physical connections between routers, and a set of traffic demands between nodes, find the best routing paths for the traffic flows to minimize network congestion and delay. The objective is to maximize network throughput and minimize packet loss and delay.

Dijkstra Application

- Digital Mapping Services in Google Maps: Many times, we have tried to find the distance in G-Maps, from one city to another, or from your location to the nearest desired location.
- Social Networking Applications: The standard Dijkstra algorithm can be applied using the shortest path between users measured through handshakes or connections among them.
- Designate file server: To designate a file server in a LAN (local area network), Dijkstra's algorithm can be used. Consider that an infinite amount of time is required for transmitting files from one computer to another computer.

DIJKSTRA PROBLEM EXPLANATION

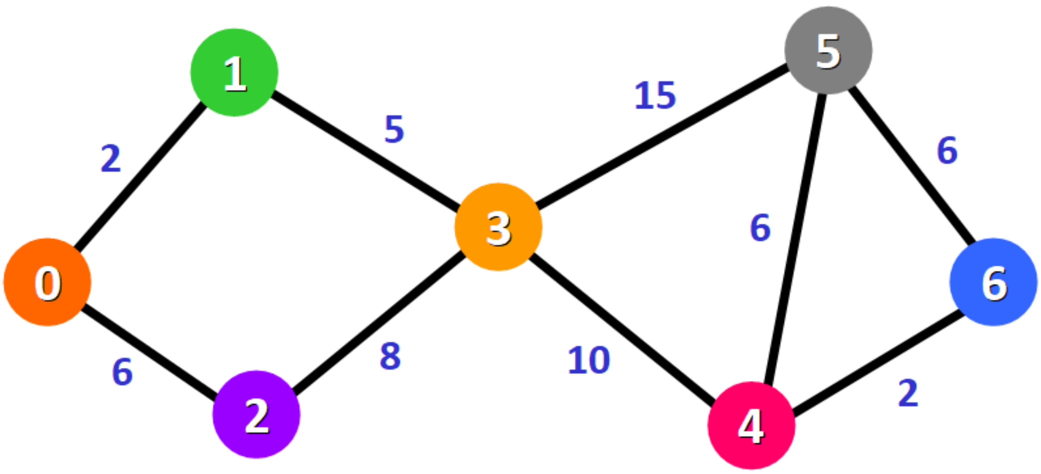
It starts with the source node and finds the rest of the distances from the source node. Dijkstra's algorithm keeps track of the currently known distance from the source node to the rest of the nodes and dynamically updates these values if a shorter path is found.

A node is then marked as **visited** and added to the path if the distance between it and the source node is the shortest. This continues until all the nodes have been added to the path, and finally, we get the shortest path from the source node to all other nodes, which packets in a network can follow to their destination.

- We need **positive** weights because they have to be added to the computations to achieve our goal. Negative weights would make the algorithm not give the desired results.

Here's an explanation of the routing internet traffic problem with a diagram and example:

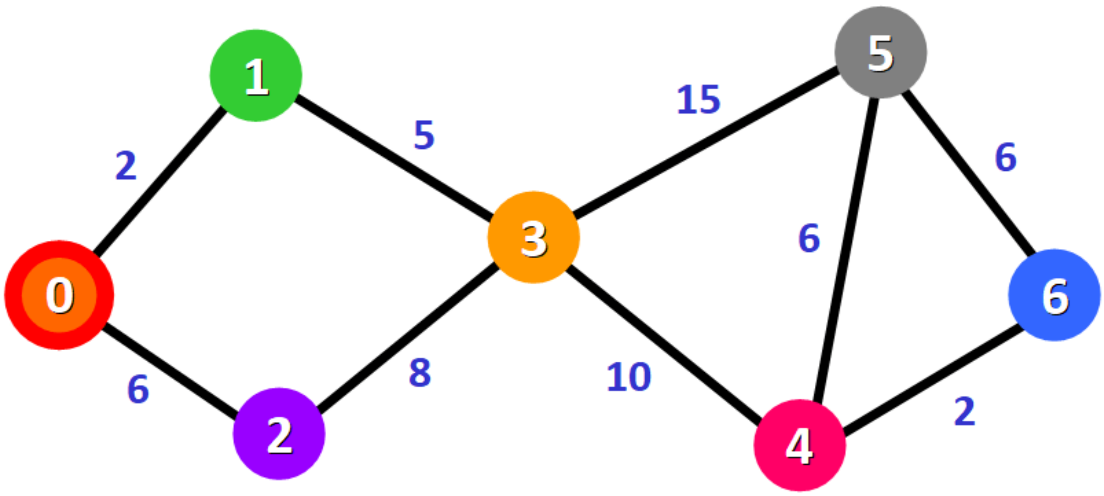
The source node here is node **0**. We assume the weights show the distances.



Initially, we have this list of distances. We mark the initial distances as INF (infinity) because we have not yet determined the actual distance except for node 0. After all, the distance from the node 0 to itself is 0.

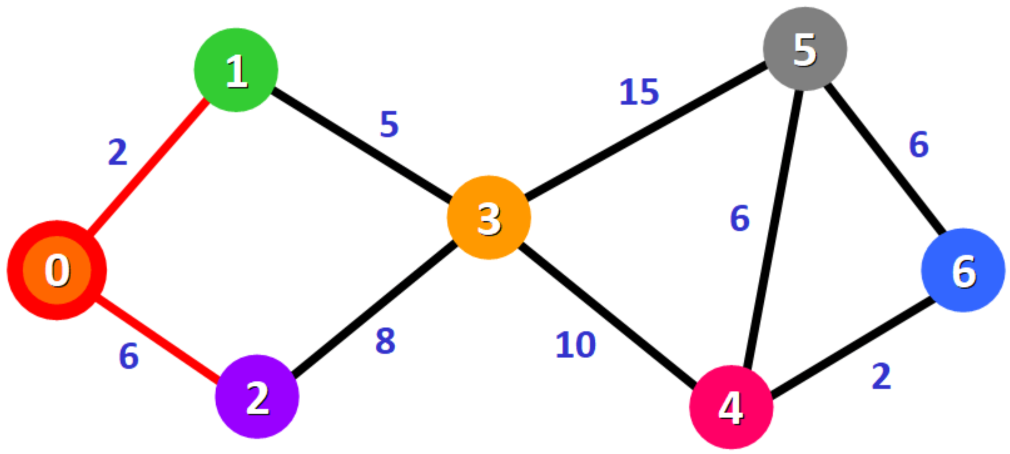
Node	Distance
0	0
1	INF
2	INF
3	INF
4	INF
5	INF
6	INF

We also have a list to keep track of only the visited nodes, and since we have started with node 0, we add it to the list (we denote a visited node by adding an asterisk beside it in the table and a red border around it on the graph).



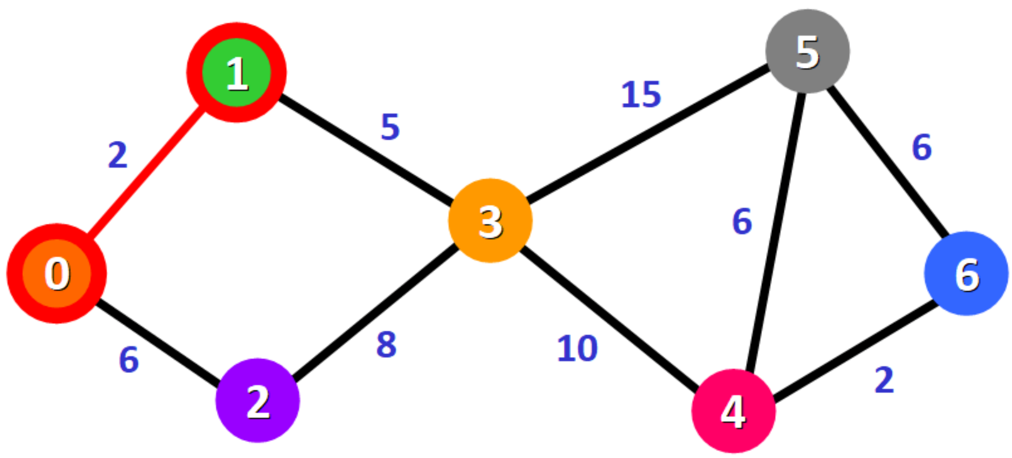
{0}

We check the distances 0 -> 1 and 0 -> 2, which are 2 and 6, respectively. We first update the distances from nodes 1 and 2 in the table.



Node	Distance
0	0
I	2
2	6
3	INF
4	INF
5	INF
6	INF

We then choose the shortest one, which is 0 -> I and mark node I as visited and add it to the visited path list.



Node	Distance
0	0
I	2*
2	6
3	INF
4	INF
5	INF
6	INF

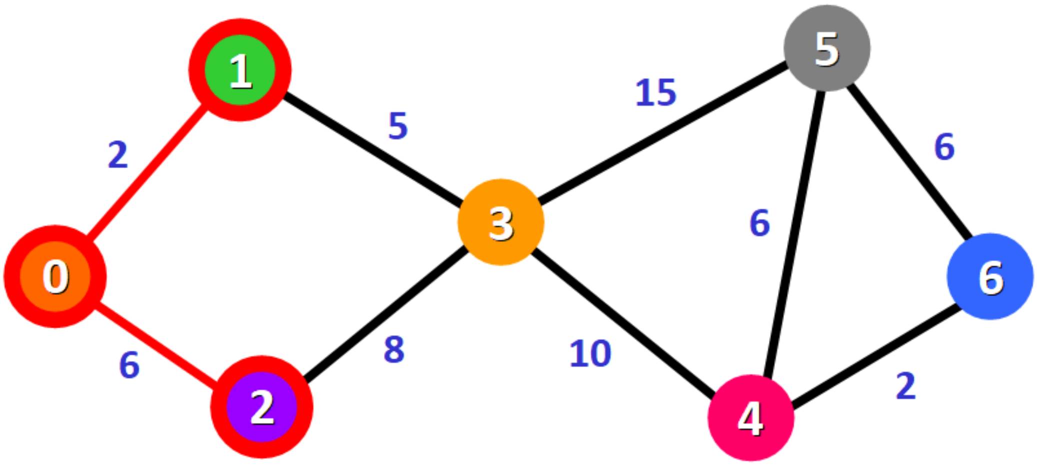
{0,I}

Next, we check the nodes adjacent to the nodes added to the path

(Nodes 2 and 3). We then update our distance table with the distance from the source node to the new adjacent node, node 3

(2 + 5 = 7).

To choose what to add to the path, we select the node with the shortest currently known distance to the source node, which is 0 -> 2 with distance 6.



Node	Distance
0	0
1	2*
2	6*
3	7
4	INF
5	INF
6	INF

{0,1,2}

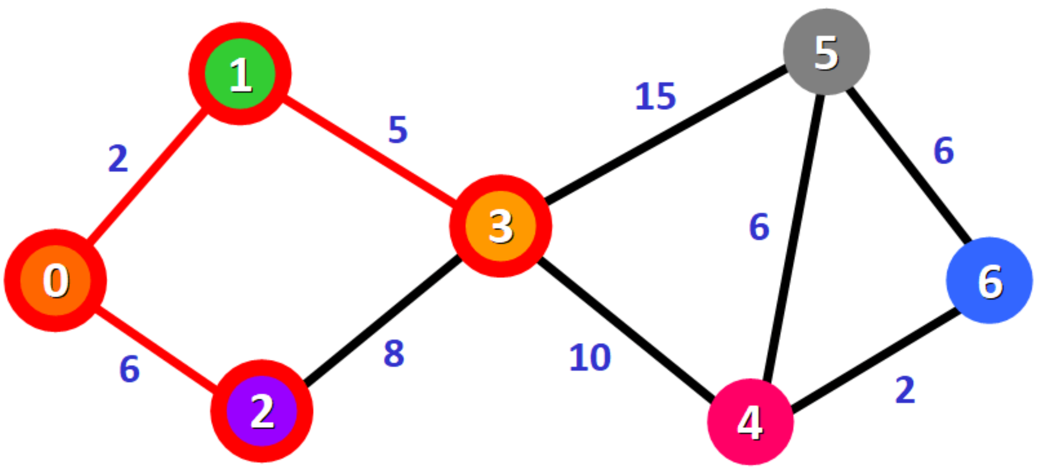
Next, we have the distances

0 -> 1 -> 3

(2 + 5 = 7) and

0 -> 2 -> 3

(6 + 8 = 14) in which 7 is clearly the shorter distance, so we add node 3 to the path and mark it as visited.

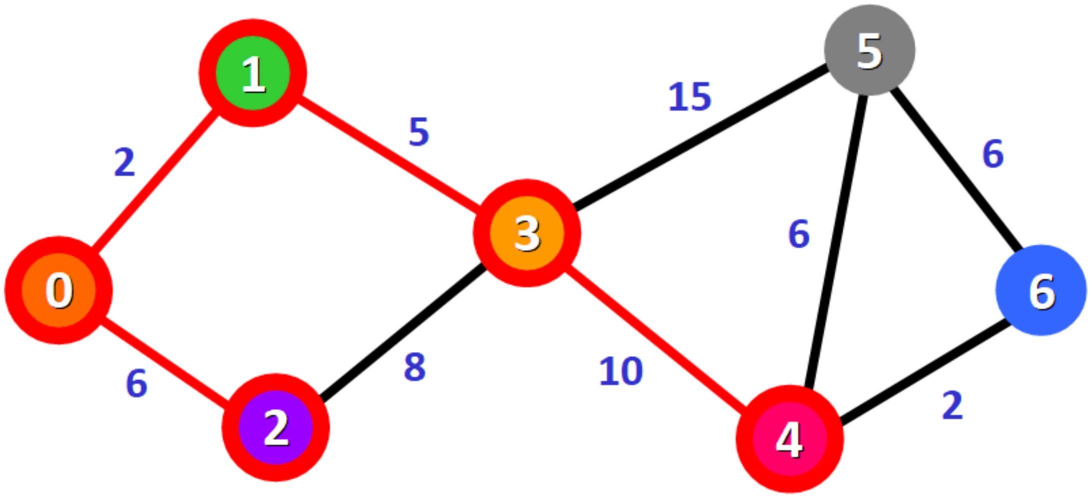


Node	Distance
0	0
I	2*
2	6*
3	7*
4	INF
5	INF
6	INF

{0,I,2,3}

We then check the next adjacent nodes (node 4 and 5) in which we have 0 -> I -> 3 -> 4 (7 + I0 = I7) for node 4 and 0 -> I -> 3 -> 5

(7 + I5 = 22) for node 5. We add node 4.



Node	Distance
0	0
I	2*
2	6*
3	7*
4	I7*

Node	Distance
5	22
6	INF

{0,1,2,3,4}

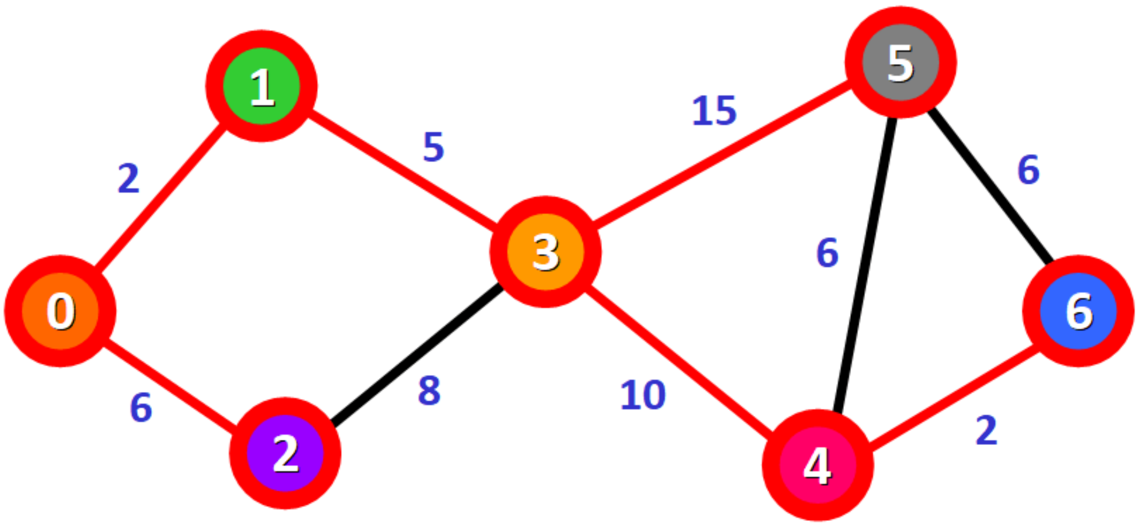
In the same way, we check the adjacent nodes (nodes 5 and 6).

Node 5:

- Option 1: 0 -> 1 -> 3 -> 5(7 + 15 = 22)
- Option 2: 0 -> 1 -> 3 -> 4 -> 5(17 + 6 = 23)
- Option 3: 0 -> 1 -> 3 -> 4 -> 6 -> 5(17 + 2 + 6 = 25) We choose 22.

Node 6:

0 -> 1 -> 3 -> 4 -> 6(17 + 2 = 19)



Node	Distance
0	0
1	2*
2	6*

Node	Distance
3	7*
4	17*
5	22*
6	19*

{0,1,2,3,4,5,6}

DESIGN TECHNIQUE USED – GREEDY METHOD

The greedy method is an algorithmic technique in which the algorithm makes locally optimal choices at each step with the hope of finding a global optimum. In other words, at each step, the algorithm chooses the best option available, without considering the future consequences of that choice.

- **USAGE**

The greedy method is often used for optimization problems in which the goal is to find the maximum or minimum value of a function, given a set of constraints. The basic idea is to start with an empty solution and then add elements to it one by one, choosing the element that gives the greatest benefit at each step. The algorithm continues until a complete solution is obtained.

- **ADVANTAGE**

One advantage of the greedy method is its efficiency. The algorithm can often find a good solution quickly, especially for problems with a large number of elements. However, the greedy method does not guarantee finding the optimal solution, and sometimes it can lead to suboptimal solutions.

To overcome this limitation, sometimes a modified version of the greedy method called "greedy with backtracking" is used. In this approach, the algorithm makes a locally optimal choice at each step, but if it turns out to be a bad choice in the future, the algorithm backtracks and tries another option.

Overall, the greedy method is a useful technique for solving optimization problems, but it should be used with caution, and its limitations should be considered when choosing an algorithm for a specific problem.

DIJKSTRA ALGORITHM FOR THE PROBLEM

Algorithm:

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

EXPLANATION OF ALGORITHM WITH EXAMPLE:

Input:

Network topology with routers and links.

Traffic flows with source and destination nodes and traffic volumes.

Output:

- Routing paths for each traffic flow.
- For each traffic flow, find the shortest path from the source node to the destination node using Dijkstra's algorithm.
- For each traffic flow, allocate traffic volume to each link in the shortest path, taking into account the link capacities.
- For each traffic flow, calculate the total cost of the path, based on the link capacities and traffic volumes.
- Choose the traffic flow with the highest benefit-to-cost ratio.
- Allocate the traffic volume of the chosen flow to the links in its shortest path, taking into account the link capacities.
- Repeat steps 4-5 until all traffic flows are allocated.

Let's say we want to route internet traffic from node 0 to node 6, and we have the following traffic demands.

(0, 6, 10)

(3, 6, 5)

(5, 1, 7)

This means that there is a demand of 10 units of traffic from node 0 to node 6, a demand of 5 units from node 3 to node 6, and a demand of 7 units from node 5 to node 1.

The algorithm starts by running Dijkstra's shortest path algorithm from the source node 0. This gives us the shortest paths and distances from node 0 to all other nodes in the network. The resulting distances are:

dist[0] = 0

dist[1] = 2

dist[2] = INF

dist[3] = INF

dist[4] = INF

dist[5] = 1

dist[6] = INF

Next, we loop through the traffic demands and find the one with the highest ratio of flow to cost. The cost of a demand is the sum of the distances along the shortest path from the source to the destination node. For example, for the demand (0, 6, 10), the cost is dist[6] - dist[0] = INF - 0 = INF. The flow is the amount of traffic that needs to be routed from the source to the destination. Initially, the entire demand is available for routing.

In this case, the demand with the highest ratio is (0, 6, 10), since it has the lowest cost and the highest flow. We then find the minimum capacity along the shortest path from node 0 to node 6. In this case, the minimum capacity is 2, since that is the capacity of the edge from node 0 to node 1. We route 2 units of traffic along this edge, updating the capacities of the edges accordingly.

Next, we loop through the traffic demands again and find the demand with the highest ratio, which is now (0, 6, 8), since we have already routed 2 units of traffic for this demand. We find the minimum capacity along the shortest path from node 0 to node 6, which is 5, and route 5 units of traffic along this path. We update the capacities of the edges accordingly.

Finally, we loop through the traffic demands one last time and find that all demands have been satisfied, so we terminate the algorithm.

C++ CODE FOR THE PROBLEM

```cpp
#include <iostream>
#include <vector>

#define INT_MAX 10000000

using namespace std;

void DijkstrasTest();

int main() {
  DijkstrasTest();
  return 0;
}

class Node;
class Edge;

void Dijkstras();
vector<Node*>* AdjacentRemainingNodes(Node* node);
Node* ExtractSmallest(vector<Node*>& nodes);
int Distance(Node* node1, Node* node2);
bool Contains(vector<Node*>& nodes, Node* node);
void PrintShortestRouteTo(Node* destination);

vector<Node*> nodes;
vector<Edge*> edges;

class Node {
   public:
  Node(char id)
    : id(id), previous(NULL), distanceFromStart(INT_MAX) {
    nodes.push_back(this);
  }

   public:
  char id;
  Node* previous;
  int distanceFromStart;
};

class Edge {
   public:
  Edge(Node* node1, Node* node2, int distance)
    : node1(node1), node2(node2), distance(distance) {
    edges.push_back(this);
  }
  bool Connects(Node* node1, Node* node2) {
    return (
      (node1 == this->node1 &&
       node2 == this->node2) ||
      (node1 == this->node2 &&
       node2 == this->node1));
```

```cpp
    }

    public:
    Node* node1;
    Node* node2;
    int distance;
};
```

```cpp

void DijkstrasTest() {
    Node* a = new Node('0');
    Node* b = new Node('1');
    Node* c = new Node('2');
    Node* d = new Node('3');
    Node* e = new Node('4');
    Node* f = new Node('5');
    Node* g = new Node('6');

    Edge* e1 = new Edge(a, b, 2);
    Edge* e2 = new Edge(a, c, 6);
    Edge* e3 = new Edge(b, d, 5);
    Edge* e4 = new Edge(c, d, 8);
    Edge* e5 = new Edge(d, e, 10);
    Edge* e6 = new Edge(d, f, 15);
    Edge* e7 = new Edge(e, f, 6);
    Edge* e8 = new Edge(f, g, 6);
    Edge* e9 = new Edge(e, g, 2);

    a->distanceFromStart = 0;
    Dijkstras();
    PrintShortestRouteTo(f);
}


void Dijkstras() {
    while (nodes.size() > 0) {
        Node* smallest = ExtractSmallest(nodes);
        vector<Node*>* adjacentNodes =
```

```cpp
            AdjacentRemainingNodes(smallest);

        const int size = adjacentNodes->size();
        for (int i = 0; i < size; ++i) {
            Node* adjacent = adjacentNodes->at(i);
            int distance = Distance(smallest, adjacent) +
                    smallest->distanceFromStart;

            if (distance < adjacent->distanceFromStart) {
                adjacent->distanceFromStart = distance;
                adjacent->previous = smallest;
            }
        }
        delete adjacentNodes;
    }
}
```

```cpp


Node* ExtractSmallest(vector<Node*>& nodes) {
  int size = nodes.size();
  if (size == 0) return NULL;
  int smallestPosition = 0;
  Node* smallest = nodes.at(0);
  for (int i = 1; i < size; ++i) {
    Node* current = nodes.at(i);
    if (current->distanceFromStart <
      smallest->distanceFromStart) {
      smallest = current;
      smallestPosition = i;
    }
  }
  nodes.erase(nodes.begin() + smallestPosition);
  return smallest;
}


vector<Node*>* AdjacentRemainingNodes(Node* node) {
  vector<Node*>* adjacentNodes = new vector<Node*>();
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    Node* adjacent = NULL;
    if (edge->node1 == node) {
      adjacent = edge->node2;
    } else if (edge->node2 == node) {
      adjacent = edge->node1;
    }
    if (adjacent && Contains(nodes, adjacent)) {
      adjacentNodes->push_back(adjacent);
    }
  }
  return adjacentNodes;
}


int Distance(Node* node1, Node* node2) {
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    if (edge->Connects(node1, node2)) {
      return edge->distance;
    }
  }
  return -1;
}


bool Contains(vector<Node*>& nodes, Node* node) {
  const int size = nodes.size();
```

```
  for (int i = 0; i < size; ++i) {
    if (node == nodes.at(i)) {
      return true;
    }
  }
  return false;
}



void PrintShortestRouteTo(Node* destination) {
  Node* previous = destination;
  cout << "Distance from start: "
     << destination->distanceFromStart << endl;
  while (previous) {
    cout << previous->id << " ";
    previous = previous->previous;
  }
  cout << endl;
}


vector<Edge*>* AdjacentEdges(vector<Edge*>& Edges, Node* node);
void RemoveEdge(vector<Edge*>& Edges, Edge* edge);

vector<Edge*>* AdjacentEdges(vector<Edge*>& edges, Node* node) {
  vector<Edge*>* adjacentEdges = new vector<Edge*>();

  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    if (edge->node1 == node) {
      cout << "adjacent: " << edge->node2->id << endl;
      adjacentEdges->push_back(edge);
    } else if (edge->node2 == node) {
      cout << "adjacent: " << edge->node1->id << endl;
      adjacentEdges->push_back(edge);
    }
  }
  return adjacentEdges;
}

void RemoveEdge(vector<Edge*>& edges, Edge* edge) {
  vector<Edge*>::iterator it;
  for (it = edges.begin(); it < edges.end(); ++it) {
    if (*it == edge) {
      edges.erase(it);
      return;
    }
  }
}
```

Output:

```
Distance from start: 22
5 3 1 0
```

COMPLEXITY ANALYSIS

The time complexity of the greedy algorithm for routing internet traffic depends on the time complexity of the shortest path algorithm that is used. In the case of Dijkstra's algorithm, the time complexity is O((E+V)logV), where E is the number of edges in the graph, V is the number of vertices, and logV is the time complexity of the priority queue used to store the distances.

In addition to the time complexity of the shortest path algorithm, the time complexity of the greedy algorithm also depends on the number of traffic demands that need to be routed. In the worst case, if there are k demands and each demand requires the full capacity of the network, then the time complexity of the algorithm would be O(k(E+V)logV). However, in practice, the number of demands and the amount of traffic they require are usually much smaller than the capacity of the network, so the actual time complexity is often much lower.

The space complexity of the algorithm is O(E), since we need to store the capacities of each edge in the network.

- Time Complexity: `O(E Log V)`

where, E is the number of edges and V is the number of vertices.

- Space Complexity: `O(V)`

CONCLUSION

The routing of internet traffic is a critical problem in modern networking, and it is essential to optimize network flow to ensure efficient and reliable data transmission. The greedy algorithm for routing internet traffic is a practical and efficient approach to network flow optimization.

The algorithm is based on finding the shortest paths between the source and destination nodes using a shortest path algorithm such as Dijkstra's algorithm. Once the shortest paths have been identified, traffic is routed along these paths based on the highest ratio of flow to cost. This means that traffic is directed along the paths that offer the greatest amount of flow relative to their capacity, which helps to minimize congestion and ensure efficient data transmission.

The time complexity of the algorithm depends on the time complexity of the shortest path algorithm used and the number of traffic demands that need to be routed. In practice, the algorithm is often very efficient, especially when the number of demands and the amount of traffic they require are relatively small compared to the capacity of the network.

Overall, the greedy algorithm for routing internet traffic is an important tool for network flow optimization. It is practical, efficient, and can help to ensure reliable and efficient data transmission across modern networks. As the demand for data transmission continues to grow, it is likely that this algorithm and others like it will become even more critical for optimizing network performance and ensuring efficient data transmission.

REFERENCES

1. Javatpoint - https://www.javatpoint.com
2. Tutorials point - https://www.tutorialspoint.com
3. GeeksforGeeks - https://www.geeksforgeeks.org