**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**COMPUTER NETWORKS**



**CS208n**

**LAB FILE**

**SUBMITTED BY:**

**Nikshay Yadav**

**A-4**

**23/CS/284**

**SUBMITTED TO:**

**Dr. R. K. Yadav**

**INDEX**

| Exp.No. | Objective | Page No. | Sign |
| --- | --- | --- | --- |
| 1 | Write a program to implement: 1)Character Stuffing 2)Bit Stuffing  3)Byte Stuffing | 3-7 |  |
| 2 | Execute the following networking command: rp, ifconfig, hostname, netdiag, netstart, nslookup, pathping, ping, route, tracer | 8-13 |  |
| 3 | Write a program to find the cyclic redundancy check for the given data | 14-16 |  |
| 4 | Write a program to implement the shortest path algorithm. (Dijkstra’s Algorithm) | 17-18 |  |
| 5 | Write a program to implement Kruskal’s Algorithm. | 19-21 |  |
| 6 | Write a program to implement a distance vector algorithm | 22-23 |  |
| 7 | Write a program to implement the Link State Routing Algorithm | 24-25 |  |
| 8 | Write a program to simulate stop and wait protocol | 26-27 |  |
| 9 | Write a program to simulate the sliding window protocol | 28-29 |  |
| 10 | Write a program to implement a client-server architecture | 30-31 |  |

**EXPERIMENT-1a**

**AIM:**

Write a program to implement Character Stuffing.

**INTRODUCTION:**

Character stuffing is a process used in data link layer protocols to differentiate between data and control information. It involves inserting special characters (such as escape sequences) before reserved control characters.

**ALGORITHM:**

1. Read the input data stream.
2. Insert a special character before any occurrence of the reserved control character.
3. Transmit the modified data.
4. On the receiving end, remove the special character to retrieve the original data.

**CODE:**

#include <iostream>

#include <string>

using namespace std;

string characterStuffing(string data, char flag = 'F', char esc = 'E') {

string stuffed = "";

stuffed += flag; // Start flag

for (char ch : data) {

if (ch == flag || ch == esc) {

stuffed += esc; // Add ESC before FLAG or ESC

}

stuffed += ch;

}

stuffed += flag; // End flag

return stuffed;

}

string characterDestuffing(string stuffed, char flag = 'F', char esc = 'E') {

if (stuffed.front() != flag || stuffed.back() != flag) {

throw invalid\_argument("Invalid frame");

}

string destuffed = "";

for (size\_t i = 1; i < stuffed.size() - 1; i++) { // Skip flags

if (stuffed[i] == esc) {

i++; // Skip ESC and take the next character as is

}

destuffed += stuffed[i];

}

return destuffed;

}

int main() {

string data = "AABFBBE";

    cout << "Data: " << data << endl;

string stuffed = characterStuffing(data);

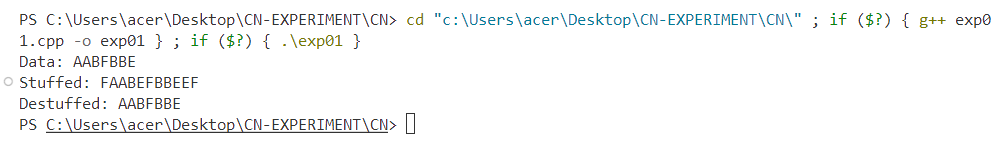
cout << "Stuffed: " << stuffed << endl;

string destuffed = characterDestuffing(stuffed);

cout << "Destuffed: " << destuffed << endl;

return 0;

}

**OUTPUT:**

**EXPERIMENT-1b**

**AIM:**

Write a program to implement Bit Stuffing

**INTRODUCTION:**

Bit stuffing is a technique used to ensure that a specific bit pattern does not occur in the data stream. A bit is inserted after every sequence of five consecutive 1s.

**ALGORITHM:**

1. Scan the input bitstream.
2. Whenever a sequence of five consecutive 1s is encountered, insert a 0 after it.
3. Transmit the modified bitstream.
4. At the receiver end, remove the extra 0s after five consecutive 1s to retrieve the original data.

**CODE:**

#include <iostream>

using namespace std;

string bitStuffing(string data) {

string stuffed = "";

int count = 0;

for (size\_t i = 0; i < data.size(); ++i) {

char bit = data[i];

stuffed += bit;

if (bit == '1') {

count++;

if (count == 5) { // If 5 consecutive 1s, stuff a 0

stuffed += '0';

count = 0; // Reset count

}

} else {

count = 0;

}

}

return stuffed;

}

string bitDestuffing(string stuffed) {

string destuffed = "";

int count = 0;

for (size\_t i = 0; i < stuffed.size(); i++) {

destuffed += stuffed[i];

if (stuffed[i] == '1') {

count++;

if (count == 5) { // If five 1s are found, skip the next bit (stuffed 0)

i++;

count = 0;

}

} else {

count = 0;

}

}

return destuffed;

}

int main() {

string bitData = "00110011010";

cout << "Bit Data: " << bitData << endl;

string stuffed = bitStuffing(bitData);

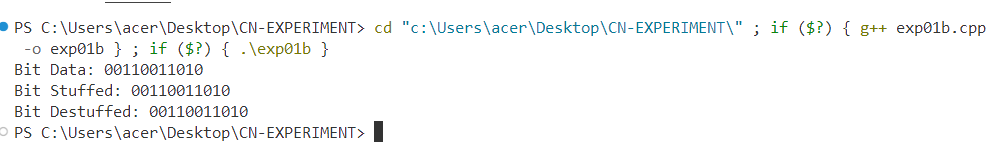
cout << "Bit Stuffed: " << stuffed << endl;

string destuffed = bitDestuffing(stuffed);

cout << "Bit Destuffed: " << destuffed << endl;

return 0;

}

**OUTPUT:**

**EXPERIMENT-1c**

**AIM:**

Write a program to implement Byte Stuffing

**INTRODUCTION:**

Byte stuffing is similar to character stuffing but uses byte-level escape sequences to differentiate between data and control information.

**ALGORITHM:**

1. Read the input data.
2. If any control characters (such as FLAG or ESC) appear, insert an escape character before them.
3. Transmit the modified data.
4. At the receiver end, remove the escape characters to retrieve the original data.

**CODE:**

#include <iostream>

#include <string>

using namespace std;

string byteStuffing(string data, char flag, char esc) {

string stuffed = "";

stuffed += flag;

for (size\_t i = 0; i < data.length(); i++) {

if (data[i] == flag || data[i] == esc)

stuffed += esc;

stuffed += data[i];

}

stuffed += flag;

return stuffed;

}

int main() {

string data;

char flag = 'F', esc = 'E';

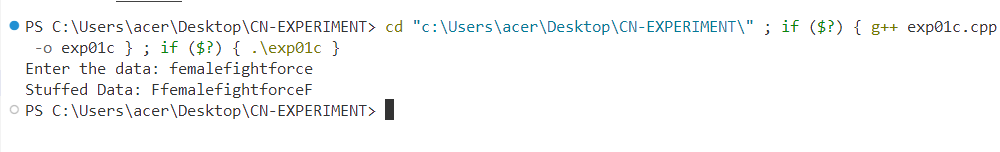
cout << "Enter the data: ";

cin >> data;

cout << "Stuffed Data: " << byteStuffing(data, flag, esc) << endl;

return 0;

}

**OUTPUT:**

**EXPERIMENT-2**

**AIM:**

Execute the following networking commands:

rp , ipconfig, hostname, netdiag, netstart, nslookup, pathping, ping, route,tracert

**INTRODUCTION:**

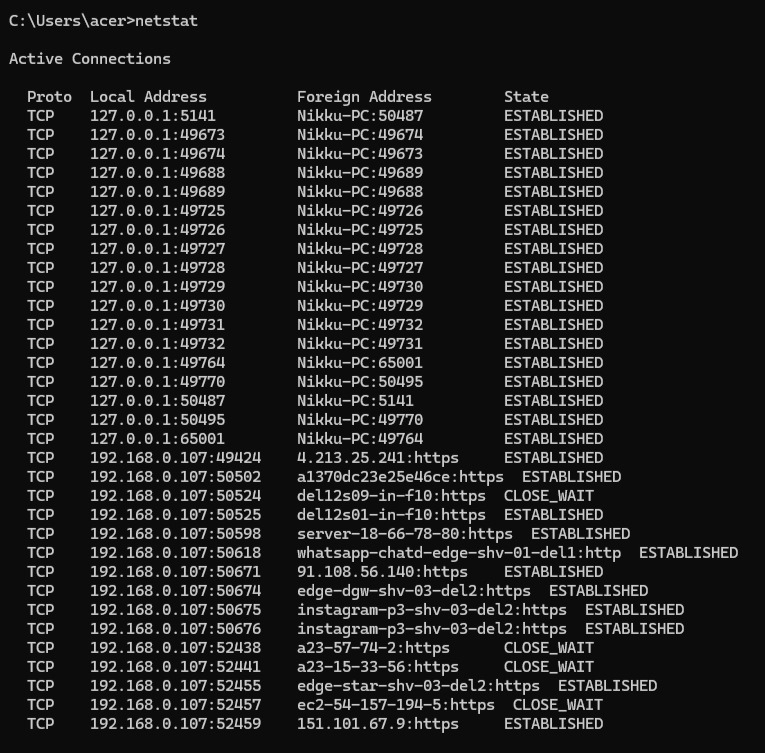
Networking commands are essential tools used to diagnose, troubleshoot, and analyze network connectivity issues. These commands help system administrators and users inspect network configurations, check connectivity, and track data transmission routes. By executing these commands, users can gain insights into IP configuration, routing tables, domain name resolution, and network path performance.

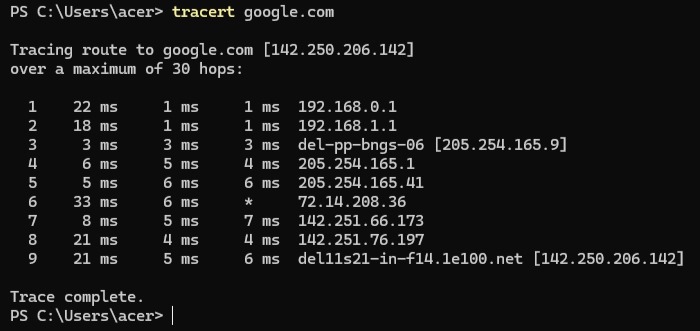
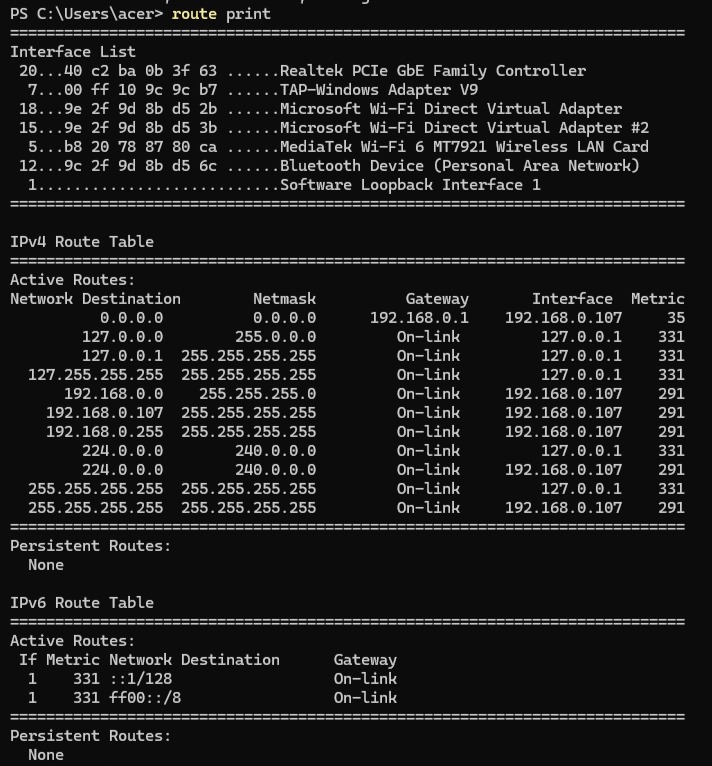
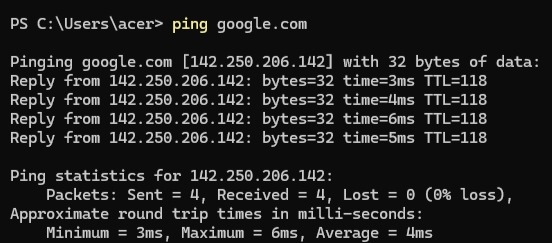
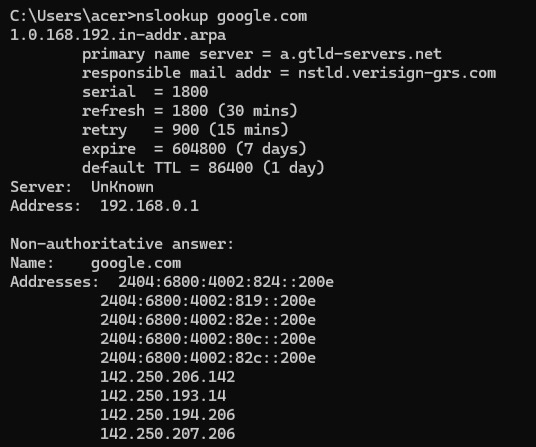
**ALGORITHM:**

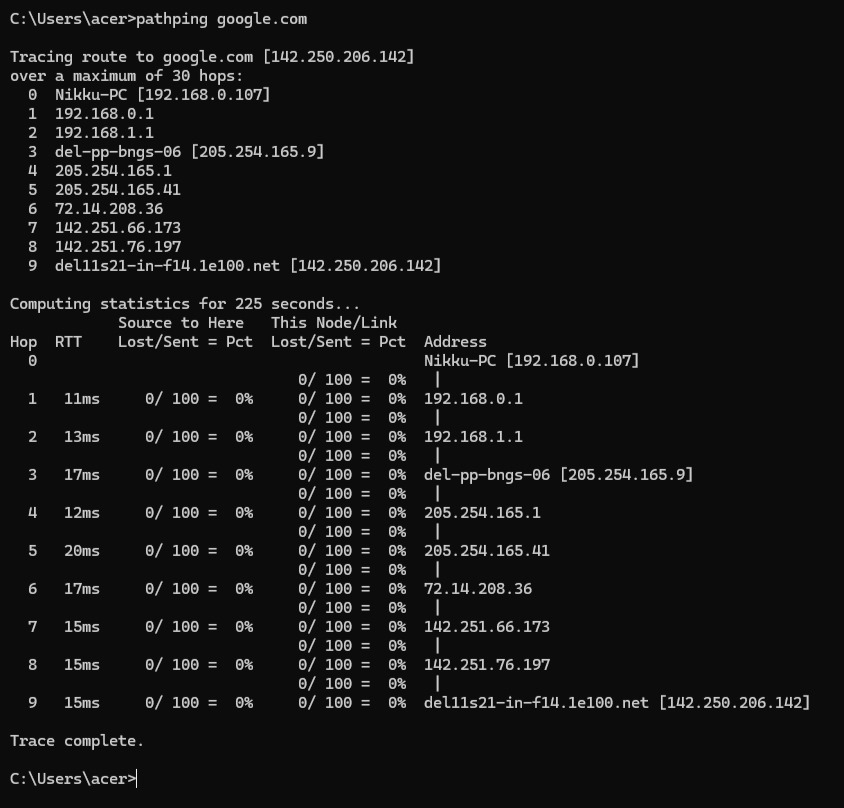
1. Open the Command Line Interface (CLI):
   * + On Windows: Open Command Prompt (cmd.exe) or PowerShell.
     + On Mac/Linux: Open Terminal.
2. Execute Networking Commands One by One:
   * + Check network configuration:
       - * ipconfig
     + Find the system’s hostname:
       - * hostname
     + Display active network connections:
       - * netstat
     + Test domain name resolution (DNS):
       - * nslookup google.com
     + Check network path latency:
       - * pathping google.com
     + Ping a remote server to test connectivity:
       - * ping google.com
     + Display routing table information:
       - * route print
     + Trace the path of packets to a destination:
       - * tracert google.com

**OUTPUT:**

****

****

****



**EXPERIMENT-3**

**AIM:**

Write a program to find the cyclic redundancy check for the given data

**INTRODUCTION:**

Cyclic Redundancy Check (CRC) is an error-detecting code used in digital networks to detect accidental changes to raw data. It uses polynomial division to generate a checksum appended to the transmitted data. The receiver performs the same calculation to verify the integrity of the data.

**ALGORITHM:**

1. Append n-1 zeroes to the data, where n is the length of the divisor (generator polynomial).
2. Perform binary division using XOR operation.
3. The remainder obtained after division is the CRC checksum.
4. Append the checksum to the original data and transmit it.
5. The receiver performs the same division. If the remainder is zero, the data is correct; otherwise, an error is detected.

**CODE:**

#include <iostream>

#include <string>

using namespace std;

// Function to perform XOR operation

string xorOperation(string dividend, string divisor) {

for (int i = 0; i < divisor.length(); i++)

dividend[i] = (dividend[i] == divisor[i]) ? '0' : '1';

return dividend;

}

// Function to perform CRC division

string divideData(string data, string generator) {

int dataLen = data.length();

int genLen = generator.length();

string temp = data.substr(0, genLen);

cout << "Before XOR Division (Padded Data): " << data << endl;

for (int i = genLen; i <= dataLen; i++) {

if (temp[0] == '1')

temp = xorOperation(temp, generator);

temp = temp.substr(1); // Remove first bit

if (i < dataLen)

temp += data[i]; // Append next bit

}

cout << "Remainder after XOR Division: " << temp << endl;

return temp;

}

int main() {

string dataBits, generator;

cout << "Enter data bits: ";

cin >> dataBits;

cout << "Enter generator polynomial: ";

cin >> generator;

// Append zeros to data

string paddedData = dataBits + string(generator.length() - 1, '0');

// Get remainder

string remainder = divideData(paddedData, generator);

// Transmitted data

string transmittedData = dataBits + remainder;

cout << "Transmitted data: " << transmittedData << endl;

// Receiving side

string receivedData;

cout << "Enter received data: ";

cin >> receivedData;

cout << "Before Checking, Received Data: " << receivedData << endl;

// Compute remainder for received data

string checkRemainder = divideData(receivedData, generator);

if (checkRemainder.find('1') != string::npos)

cout << "Error detected in received data." << endl;

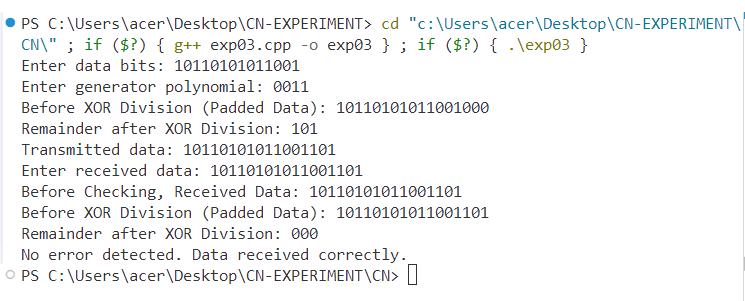
else

cout << "No error detected. Data received correctly." << endl;

return 0;

}

**OUTPUT:**

****

**EXPERIMENT-4**

**AIM:**

Write a program to implement the shortest path algorithm. (Dijkstra’s Algorithm)

**INTRODUCTION:**

Dijkstra’s algorithm is a graph search algorithm that finds the shortest path between nodes in a weighted graph. It is commonly used in routing and network flow problems.

**ALGORITHM:**

1. Initialise the distance of the source node to 0 and all other nodes to infinity.
2. Mark all nodes as unvisited and set the source node as the current node.
3. For the current node, update the distance of all its unvisited neighbours.
4. Select the unvisited node with the smallest distance and repeat step 3 until all nodes are visited.
5. The shortest path from the source node to all other nodes is determined.

**CODE:**

#include <iostream>

#include <vector>

#include <limits.h>

using namespace std;

#define V 5 // Number of vertices

int minDistance(int dist[], bool visited[]){

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (!visited[v] && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

void dijkstra(int graph[V][V], int src){

int dist[V];

bool visited[V] = {false};

for (int i = 0; i < V; i++)

dist[i] = INT\_MAX;

dist[src] = 0;

for (int count = 0; count < V - 1; count++){

int u = minDistance(dist, visited);

visited[u] = true;

for (int v = 0; v < V; v++)

if (!visited[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

for (int i = 0; i < V; i++)

cout << "Vertex " << i << " Distance from Source: " << dist[i] << endl;

}

int main(){

int graph[V][V] = {

{0, 15, 0, 25, 80},

{15, 0, 40, 0, 0},

{0, 40, 0, 15, 5},

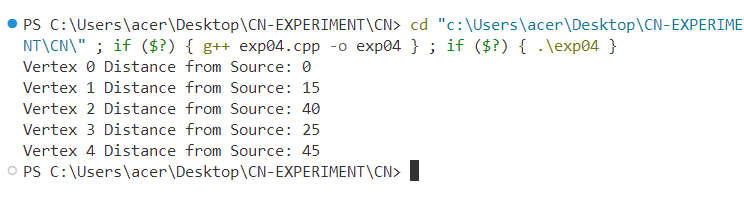
{25, 0, 15, 0, 45},

{80, 0, 5, 45, 0}};

dijkstra(graph, 0);

return 0;

}

**OUTPUT:**

**EXPERIMENT-5**

**AIM:**

Write a program to implement Kruskal’s Algorithm.

**INTRODUCTION:**

Kruskal’s algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a graph. It sorts all edges in non-decreasing order of weight and picks the smallest edge that does not form a cycle.

**ALGORITHM:**

1. Sort all edges in increasing order of weight.
2. Pick the smallest edge. Check if it forms a cycle using the union-find method.
3. If it does not form a cycle, include it in the MST.
4. Repeat until the MST contains (V-1) edges.

**CODE:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Edge {

int src, dest, weight;

};

// Comparator function for sorting edges based on weight

bool compareEdges(Edge a, Edge b) {

return a.weight < b.weight;

}

struct Graph {

int V, E;

vector<Edge> edges;

};

Graph createGraph(int V, int E) {

Graph graph;

graph.V = V;

graph.E = E;

return graph;

}

struct subset {

int parent, rank;

};

int find(subset subsets[], int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

void Union(subset subsets[], int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

void KruskalMST(Graph graph) {

vector<Edge> result;

int e = 0, i = 0;

// Sort edges based on weight

sort(graph.edges.begin(), graph.edges.end(), compareEdges);

subset\* subsets = new subset[graph.V];

for (int v = 0; v < graph.V; v++) {

subsets[v].parent = v;

subsets[v].rank = 0;

}

while (e < graph.V - 1 && i < graph.E) {

Edge next\_edge = graph.edges[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

if (x != y) {

result.push\_back(next\_edge);

Union(subsets, x, y);

e++;

}

}

cout << "Edges in MST:\n";

for (size\_t j = 0; j < result.size(); j++) {

cout << result[j].src << " - " << result[j].dest << " : " << result[j].weight << endl;

}

delete[] subsets;

}

int main() {

int V = 4, E = 5;

Graph graph = createGraph(V, E);

// Randomized edges and weights

graph.edges.push\_back((Edge){0, 1, 12});

graph.edges.push\_back((Edge){0, 2, 8});

graph.edges.push\_back((Edge){0, 3, 7});

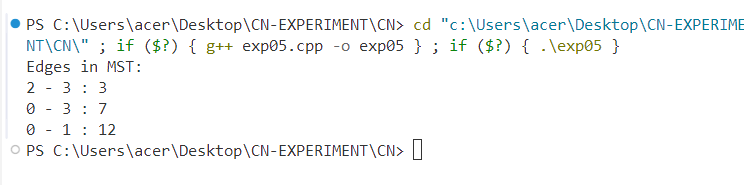
graph.edges.push\_back((Edge){1, 2, 14});

graph.edges.push\_back((Edge){2, 3, 3});

KruskalMST(graph);

return 0;

}

**OUTPUT:**

**EXPERIMENT-6**

**AIM:**

Write a program to implement a distance vector algorithm.

**INTRODUCTION:**

The Distance Vector Routing Algorithm is a distributed routing algorithm used in computer networks. Each router maintains a routing table, which stores the shortest known distance to each destination and updates the table based on information received from its neighbours.

**ALGORITHM:**

1. Initialise the distance table with direct link costs to neighbours.
2. Periodically exchange distance vectors with neighbouring routers.
3. Update the routing table based on received information.
4. If a better path is found, update the routing table and propagate changes.
5. Repeat the process until no more updates occur.

**CODE:**

#include <iostream>

using namespace std;

#define INF 99999

#define NODES 4

void distanceVector(int costMatrix[NODES][NODES]) {

int dist[NODES][NODES];

for (int i = 0; i < NODES; i++) {

for (int j = 0; j < NODES; j++) {

dist[i][j] = costMatrix[i][j];

}

}

for (int k = 0; k < NODES; k++) {

for (int i = 0; i < NODES; i++) {

for (int j = 0; j < NODES; j++) {

if (dist[i][k] + dist[k][j] < dist[i][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

cout << "Routing Table:\n";

for (int i = 0; i < NODES; i++) {

for (int j = 0; j < NODES; j++) {

cout << dist[i][j] << " ";

}

cout << endl;

}

}

int main() {

int costMatrix[NODES][NODES] = {

{0, 7, INF, 5},

{7, 0, 2, INF},

{INF, 2, 0, 3},

{5, INF, 3, 0}

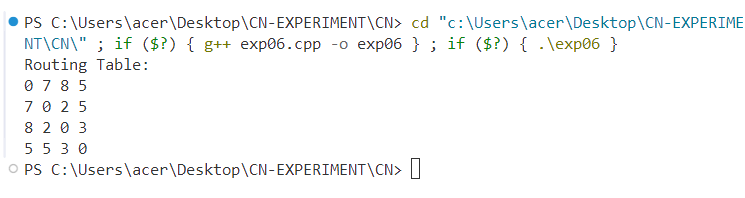
};

distanceVector(costMatrix);

return 0;

}

**OUTPUT:**



**EXPERIMENT-7**

**AIM:**

Write a program to implement the Link State Routing Algorithm.

**INTRODUCTION:**

The Link State Routing Algorithm is a dynamic routing technique used in computer networks. Unlike Distance Vector Routing, each router constructs a complete map of the network topology and independently computes shortest paths using Dijkstra’s Algorithm.

**ALGORITHM:**

1. Each router discovers its neighbours and measures link costs.
2. The router floods link state packets (LSPs) to all other routers.
3. Each router constructs a topology map based on received LSPs.
4. Dijkstra’s Algorithm is used to compute the shortest path to all nodes.
5. Routing tables are updated accordingly.

**CODE:**

#include <iostream>

#include <climits>

#define V 4

using namespace std;

int minDistance(int dist[], bool visited[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (!visited[v] && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void dijkstra(int graph[V][V], int src) {

int dist[V];

bool visited[V];

for (int i = 0; i < V; i++) {

dist[i] = INT\_MAX;

visited[i] = false;

}

dist[src] = 0;

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, visited);

visited[u] = true;

for (int v = 0; v < V; v++) {

if (!visited[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

cout << "Router " << src << " shortest paths:\n";

for (int i = 0; i < V; i++) {

cout << "To node " << i << " distance: " << dist[i] << endl;

}

}

int main() {

int graph[V][V] = {

{0, 3, 7, 0},

{3, 0, 1, 8},

{7, 1, 0, 4},

{0, 8, 4, 0}

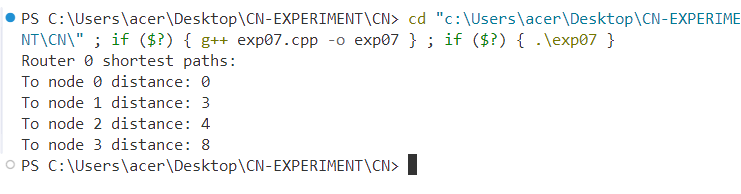
};

dijkstra(graph, 0);

return 0;

}

**OUTPUT:**



**EXPERIMENT-8**

**AIM:**

Write a program to simulate stop and wait protocol

**INTRODUCTION:**

The Stop and Wait Protocol is a flow control mechanism in which the sender transmits a frame and waits for an acknowledgment (ACK) before sending the next frame. It ensures reliable data transmission but can lead to inefficiencies due to waiting time.

**ALGORITHM:**

1. The sender transmits a frame and waits for an acknowledgment.
2. The receiver receives the frame, processes it, and sends an acknowledgment.
3. If the acknowledgment is received, the sender transmits the next frame.
4. If no acknowledgment is received within a timeout, the sender retransmits the frame.
5. The process continues until all frames are successfully transmitted.

**CODE:**

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

void stopAndWait(int totalFrames) {

int ack;

srand(time(NULL));

for (int i = 1; i <= totalFrames; i++) {

cout << "Sending frame " << i << "...\n";

if (rand() % 2 == 0) {

cout << "ACK received for frame " << i << "\n";

} else {

cout << "Frame " << i << " lost. Retransmitting...\n";

i--; // Retransmit the same frame

}

}

cout << "All frames sent successfully!\n";

}

int main() {

int totalFrames;

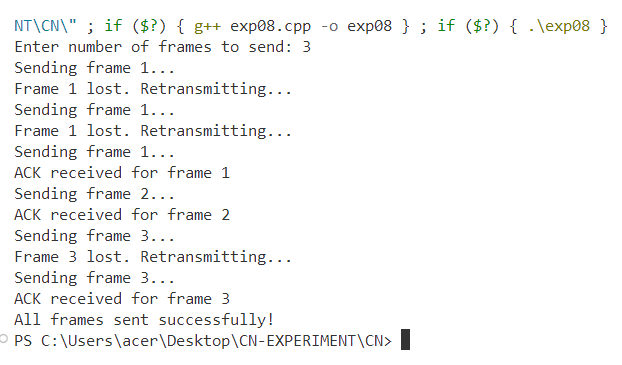
cout << "Enter number of frames to send: ";

cin >> totalFrames;

stopAndWait(totalFrames);

return 0;

}

**OUTPUT:**

**EXPERIMENT-9**

**AIM:**

Write a program to simulate the sliding window protocol

**INTRODUCTION:**

The Sliding Window Protocol is a data flow control technique used in networking. It allows multiple frames to be sent before requiring an acknowledgment, increasing efficiency compared to the Stop and Wait Protocol.

**ALGORITHM:**

1. The sender transmits multiple frames up to the window size.
2. The receiver acknowledges received frames.
3. If an acknowledgment is received, the sender slides the window and transmits the next set of frames.
4. If an acknowledgment is not received, the sender retransmits the lost frame and all subsequent frames.
5. The process continues until all frames are successfully transmitted.

**CODE:**

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

void slidingWindow(int totalFrames, int windowSize) {

int ack, i = 1;

srand(time(NULL));

while (i <= totalFrames) {

int j;

for (j = i; j < i + windowSize && j <= totalFrames; j++) {

cout << "Sending frame " << j << "...\n";

}

for (j = i; j < i + windowSize && j <= totalFrames; j++) {

if (rand() % 2 == 0) {

cout << "ACK received for frame " << j << "\n";

} else {

cout << "Frame " << j << " lost. Retransmitting...\n";

break;

}

}

if (j == i + windowSize || j > totalFrames) {

i += windowSize;

} else {

i = j; // Retransmit from the lost frame

}

}

cout << "All frames sent successfully!\n";

}

int main() {

int totalFrames, windowSize;

cout << "Enter number of frames to send: ";

cin >> totalFrames;

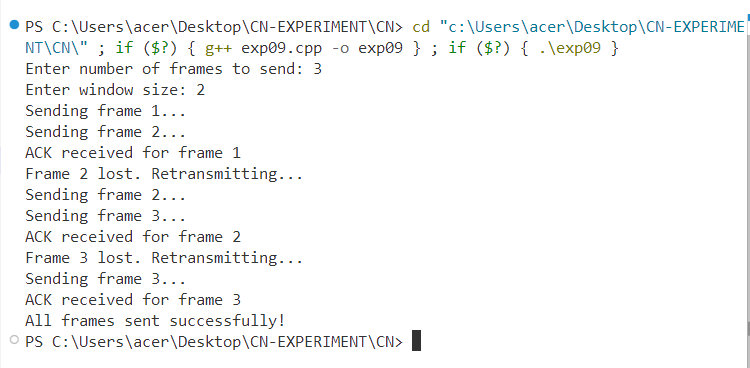
cout << "Enter window size: ";

cin >> windowSize;

slidingWindow(totalFrames, windowSize);

return 0;

}

**OUTPUT:**

**EXPERIMENT-10**

**AIM:**

Write a program to implement a client-server architecture.

**INTRODUCTION:**

The Client-Server Architecture is a network model where a server provides resources or services, and clients request and consume those services. It is commonly used in distributed systems, web applications, and database systems.

**ALGORITHM:**

1. The server initialises and waits for client connections.
2. The client requests a connection to the server.
3. Once connected, the client sends a message to the server.
4. The server processes the message and sends a response back to the client.
5. The communication continues until the client terminates the connection.

**CODE:**

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <pthread.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

using namespace std;

void\* serverFunction(void\* arg) {

int server\_fd, new\_socket;

struct sockaddr\_in address;

int addrlen = sizeof(address);

char buffer[BUFFER\_SIZE] = {0};

server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_fd == -1) {

cout << "Socket creation failed" << endl;

return NULL;

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

if (bind(server\_fd, (struct sockaddr\*)&address, sizeof(address)) < 0) {

cout << "Bind failed" << endl;

close(server\_fd);

return NULL;

}

if (listen(server\_fd, 3) < 0) {

cout << "Listen failed" << endl;

close(server\_fd);

return NULL;

}

cout << "Server: Waiting for connections..." << endl;

new\_socket = accept(server\_fd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen);

if (new\_socket < 0) {

cout << "Accept failed" << endl;

close(server\_fd);

return NULL;

}

read(new\_socket, buffer, BUFFER\_SIZE);

cout << "Server received: " << buffer << endl;

send(new\_socket, "Hello from Server", strlen("Hello from Server"), 0);

close(new\_socket);

close(server\_fd);

return NULL;

}

void clientFunction() {

int sock;

struct sockaddr\_in server\_addr;

char buffer[BUFFER\_SIZE] = {0};

sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (sock == -1) {

cout << "Socket creation failed" << endl;

return;

}

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(PORT);

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

sleep(1); // Ensure the server starts first

if (connect(sock, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) < 0) {

cout << "Connection failed" << endl;

close(sock);

return;

}

send(sock, "Hello from Client", strlen("Hello from Client"), 0);

read(sock, buffer, BUFFER\_SIZE);

cout << "Client received: " << buffer << endl;

close(sock);

}

int main() {

pthread\_t serverThread;

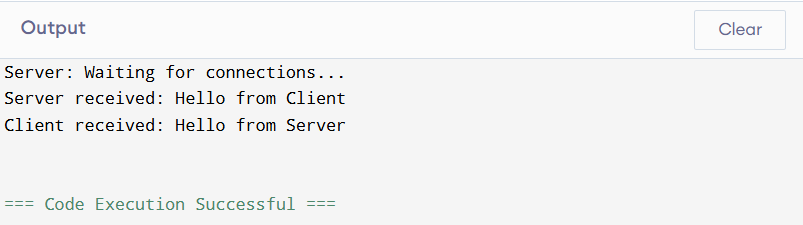
pthread\_create(&serverThread, NULL, serverFunction, NULL);

clientFunction();

pthread\_join(serverThread, NULL);

return 0;

}

**OUTPUT:**