Netaji Subhas University
of Technology

LAB REPORT

ADVANCED COMPUTER NETWORKS

Name	**Kushagra Lakhwani**
Roll No.	***2021UCI8036***
Semester	**5th**
Course	***CICPC16***

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

September 11, 2023

Abstract

The practical lab report *"Advanced Computer Networks"* is the original and unmodified content submitted by *Kushagra Lakhwani* (Roll No. 2021UCI8036).

The report is submitted to *Mr. Vishal Gupta*, Department of Computer Science and Engineering, NSUT, Delhi, for the partial fulfillment of the requirements of the course (CICPC16).

Index

1 IPv4 Address Conversion

Objective

To convert a binary IP address into dotted decimal and vice versa.

Source Code

```
/*
 * Function to convert binary IP address to dotted decimal
 * @param binaryIP - binary IP address
 * @return dottedDecimal - dotted decimal IP address
 * */
string binaryToDottedDecimal(const string &binaryIP) {
  string dottedDecimal = "";
  for (int i = 0; i < 32; i += 8) {
    bitset<8> octet(binaryIP.substr(i, 8));
    dottedDecimal += to_string(octet.to_ulong());
    if (i < 24)
      dottedDecimal += ".";
  }
  return dottedDecimal;
}

/*
 *Function to convert dotted decimal IP address to binary
 * @param dottedDecimal - dotted decimal IP address
 * @return binaryIP - binary IP address
 * */
string dottedDecimalToBinary(const string &dottedDecimal) {
  string binaryIP = "";
  size_t start = 0;
  size_t end = dottedDecimal.find(".");
  while (end != string::npos) {
    int octet = stoi(dottedDecimal.substr(start, end - start));
    binaryIP += bitset<8>(octet).to_string();
    start = end + 1;
    end = dottedDecimal.find(".", start);
  }
  int octet = stoi(dottedDecimal.substr(start));
  binaryIP += bitset<8>(octet).to_string();
  return binaryIP;
}
```

Output

Binary to dotted decimal IP address

```
$ ./ipv4
1. Binary to dotted decimal IP address
2. Dotted decimal to binary IP address
Enter your choice: 1
Enter binary IP address (32 bits): 11000000101010000000000100000001
Dotted Decimal IP address: 192.168.1.1
```

Dotted decimal to binary IP address

```
$ ./ipv4
1. Binary to dotted decimal IP address
2. Dotted decimal to binary IP address
Enter your choice: 2
Enter dotted decimal IP address (e.g., 192.168.1.1): 203.128.56.2
Binary IP address: 11001011100000000011100000000010
```

2 IP Address Classes

Objective

To identify the class of an IP address.

Theory

In IPv4, IP addresses are divided into five classes: A, B, C, D, and E. Each class has its own range of valid IP addresses and is used for specific purposes.

Class A:
- Range: 1.0.0.0 to 126.255.255.255
- Subnet Mask: 255.0.0.0
- Address Allocation: Class A addresses are typically used by large organizations and corporations. They can support a very large number of hosts on a single network.

Class B:
- Range: 128.0.0.0 to 191.255.255.255
- Subnet Mask: 255.255.0.0
- Address Allocation: Class B addresses are used by medium-sized organizations. They offer a moderate number of network and host addresses.

Class C:
- Range: 192.0.0.0 to 223.255.255.255

- Subnet Mask: 255.255.255.0
- Address Allocation: Class C addresses are commonly used by small organizations and businesses. They provide a limited number of network addresses but a larger number of host addresses.

Class D:
- Range: 224.0.0.0 to 239.255.255.255
- Address Allocation: Class D addresses are reserved for multicast groups and are not used for traditional unicast communication. They are used for one-to-many or many-to-many communication.

Class E:
- Range: 240.0.0.0 to 255.255.255.255
- Address Allocation: Class E addresses are reserved for experimental or research purposes and are not typically used in public networks. They are reserved for future use and not intended for general use.

Source Code

```cpp
/**
 * Determine the class based on the first octet
 * @param ipAddress - IP address
 * @return - 'A', 'B', 'C', 'D', 'E', or 'X'
 * */
char getIPv4Class(const string &ipAddress) {
  int firstOctet = stoi(ipAddress.substr(0, ipAddress.find(".")));
  if (firstOctet >= 1 && firstOctet <= 126) {
    return 'A';
  } else if (firstOctet >= 128 && firstOctet <= 191) {
    return 'B';
  } else if (firstOctet >= 192 && firstOctet <= 223) {
    return 'C';
  } else if (firstOctet >= 224 && firstOctet <= 239) {
    return 'D';
  } else if (firstOctet >= 240 && firstOctet <= 255) {
    return 'E';
  } else {
    return 'X'; // 'X' indicates an invalid IPv4 address
  }
}
```

Output

```
$ ./ipv4class
Enter an IPv4 address: 192.168.1.1
Class: C
```

3 Bellman-Ford Algorithm

3.1 Objective

To implement the Bellman-Ford algorithm to find the shortest path in a weighted graph.

3.2 Theory

The Bellman-Ford algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph, even when the graph contains negative weight edges. While it's not the most efficient algorithm for all cases (especially for graphs with non-negative weights, where Dijkstra's algorithm is typically faster),

3.3 Source Code

```
struct Edge {
  int source, destination, weight;
};

class Graph {
  int V, E;
  vector<Edge> edges;

public:
  Graph(int vertices, int edges);
  void addEdge(int source, int destination, int weight);
  void bellmanFord(int source);
};

Graph::Graph(int vertices, int edges) : V(vertices), E(edges) {}

void Graph::addEdge(int source, int destination, int weight) {
  edges.push_back({source, destination, weight});
}

void Graph::bellmanFord(int source) {
```

```
vector<int> distance(V, numeric_limits<int>::max());
distance[source] = 0;

for (int i = 1; i < V; ++i) {
  for (const Edge &edge : edges) {
    int u = edge.source, v = edge.destination, w = edge.weight;
    if (distance[u] != numeric_limits<int>::max() &&
        distance[u] + w < distance[v]) {
      distance[v] = distance[u] + w;
    }
  }
}

// Check for negative weight cycles
for (const Edge &edge : edges) {
  int u = edge.source, v = edge.destination, w = edge.weight;
  if (distance[u] != numeric_limits<int>::max() &&
      distance[u] + w < distance[v]) {
    cout << "Graph contains a negative weight cycle.\n";
    return;
  }
}

// Print shortest distances from the source vertex
cout << "Vertex\tDistance from Source\n";
for (int i = 0; i < V; ++i) {
  cout << i << "\t" << distance[i] << "\n";
}
```

3.4 Output

```
Enter the number of vertices and edges: 3 4
Enter edge 1 (source, destination, weight): 0 1 5
Enter edge 2 (source, destination, weight): 1 0 3
Enter edge 3 (source, destination, weight): 1 2 -1
Enter edge 4 (source, destination, weight): 2 0 1
Enter the source vertex: 2
Vertex  Distance from Source
0       1
1       6
2       0
```