

# Netaji Subhas University of Technology

# Lab Report

# Advanced Computer Networks

Name Kushagra Lakhwani

Roll No. 2021UCI8036

Semester 5th

Course CICPC16

#### **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	3 .	
_	TD A 11 CI	_	
2	IP Address Classes	5 .	
3	Classless Inter-Domain Routing (CIDR)	7	
4	Bellman-Ford Algorithm	9 .	
	Diikstra's Algorithm	11	

#### 1 IPv4 Address Conversion

#### 1.1 Objective

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

#### 1.2 Source Code

```
* Function to convert binary IP address to dotted decimal
 * @param binaryIP - binary IP address
 st Oreturn 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
 * Oparam 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;
}
```

#### 1.3 Output

#### 1.3.1 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): 1100000010100000000000010000001
Dotted Decimal IP address: 192.168.1.1
```

#### 1.3.2 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: 1100101110000000001110000000010
```

#### 2 IP Address Classes

#### 2.1 Objective

To identify the class of an IP address.

#### 2.2 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.

#### 2.3 Source Code

```
/**
 * Determine the class based on the first octet
 * Oparam 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) {</pre>
    return 'A';
  } else if (firstOctet \geq 128 && firstOctet \leq 191) {
    return 'B';
  } else if (firstOctet >= 192 && firstOctet <= 223) {
   return 'C';
  } else if (firstOctet \geq= 224 && firstOctet \leq= 239) {
    return 'D';
  } else if (firstOctet \geq 240 && firstOctet \leq 255) {
    return 'E';
  } else {
    return 'X'; // 'X' indicates an invalid IPv4 address
  }
}
```

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

### 3 Classless Inter-Domain Routing (CIDR)

#### 3.1 Objective

To find the subnet mask, network address, and broadcast address of a given IP

#### 3.2 Theory

Classless Inter-Domain Routing (CIDR) is an IP addressing scheme that improves the allocation of IP addresses and routing efficiency on the Internet. It allows for more flexible allocation of IP addresses than the original system of IP address classes (Class A, B, and C). CIDR is based on variable-length subnet masking (VLSM), which enables IP addresses to be allocated based on the actual need, rather than predefined classes.

CIDR addresses are written in the form IP\_address/prefix\_length. For example, 192.168.1.0/24 represents a CIDR address where the first 24 bits are the network address, and the remaining 8 bits are available for host addresses.

#### 3.3 Source Code

```
class CIDR {
 friend int main();
private:
 struct AddressRange {
   string networkAddress;
   int prefixLength;
 };
 vector<AddressRange> addressRanges;
public:
 void addAddressRange(string networkAddress, int prefixLength) {
    addressRanges.push_back({networkAddress, prefixLength});
 }
 bool isIPAddressInRanges(string ipAddress) {
   for (const auto &range : addressRanges) {
      string networkAddress = range.networkAddress;
      int prefixLength = range.prefixLength;
      bitset<32> networkBits(stoul(networkAddress, nullptr, 0)); // Network address to binary
      bitset<32> ipBits(stoul(ipAddress, nullptr, 0)); // IP address to binary
      bool match = true; // Check if IP address falls within the CIDR range
      for (int i = 0; i < prefixLength; ++i) {</pre>
        if (networkBits[i] != ipBits[i]) {
          match = false;
          break;
```

```
}
      if (match) return true;
    return false;
};
int main() {
  CIDR cidr;
  // Add CIDR address ranges
  cidr.addAddressRange("192.168.1.0", 24);
  cidr.addAddressRange("10.0.0.0", 8);
  cidr.addAddressRange("172.16.0.0", 12);
  // Print CIDR address ranges
  for (const auto &range : cidr.addressRanges) {
    cout << "Network Address: " << range.networkAddress << "/" << range.prefixLength << endl;</pre>
  }
  // Check if IP addresses are within the CIDR address ranges
  vector<string> testIPs = {"192.168.1.1", "10.1.2.3", "172.16.1.1", "8.8.8.8"};
  for (const auto &ip : testIPs) {
    if (cidr.isIPAddressInRanges(ip))
      cout << ip << " is within the CIDR address ranges." << endl;</pre>
      cout << ip << " is NOT within the CIDR address ranges." << endl;</pre>
  }
  return 0;
}
```

```
$ code git:(main) ./cidr
Network Address: 192.168.1.0/24
Network Address: 10.0.0.0/8
Network Address: 172.16.0.0/12
192.168.1.1 is within the CIDR address ranges.
10.1.2.3 is within the CIDR address ranges.
172.16.1.1 is within the CIDR address ranges.
8.8.8 is NOT within the CIDR address ranges.
```

### 4 Bellman-Ford Algorithm

#### 4.1 Objective

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

#### 4.2 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),

#### 4.3 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]) {</pre>
        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";
}
}</pre>
```

```
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
```

# 5 Dijkstra's Algorithm

#### 5.1 Objective

To implement Dijkstra's algorithm to find the shortest path in a weighted graph.

#### 5.2 Theory

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

Its time complexity is  $O(|E| + |V| \log |V|)$ , where |E| is the number of edges and |V| is the number of vertices. However, this algorithm is only applicable to graphs with positive edge weights.

#### 5.3 Source Code

```
void dijkstra(vector<vector<pair<int, 11>>> &G, int src, vector<11> &dist) {
 vector<bool> used(G.size(), false);
 dist[src] = 0;
 auto cmp = [&](int i, int j) { return dist[i] > dist[j]; };
 priority_queue<int, vector<int>, decltype(cmp)> pq(cmp);
 pq.push(src);
 while (!pq.empty()) {
   auto u = pq.top();
   pq.pop();
   used[u] = true;
   for (auto [v, w] : G[u]) {
      if (!used[v] \&\& dist[v] > dist[u] + w) {
        dist[v] = dist[u] + w;
        pq.push(v);
     }
   }
 }
}
int main() {
 int n, m; // Number of nodes and edges respectively
 cin >> n >> m;
 vector<vector<pair<int, 11>>> G(n); // Initialize the graph with n nodes
 // Read the edges and their weights
 for (int i = 0; i < m; ++i) {
   int u, v;
   11 w;
   cin >> u >> v >> w;
    G[u].emplace_back(v, w);
```

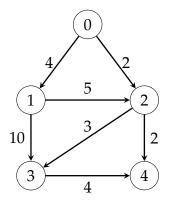


Figure 1: Graph

```
5 7
    0 1 4
    0 2 2
    1 2 5
    1 3 10
    2 3 3
    2 4 2
    3 4 4

0

Shortest distances from node 0:

Node 0: 0

Node 1: 4

Node 2: 2

Node 3: 5

Node 4: 4
```