Educate Elevate Enlighten

JSS Mahavidyapeetha

JSS Science And Technology University

(Established Under JSS Science and Technology University Act No. 43 of 2013)

(Formerly Known as SJCE)

JSS SCIENCE AND TECHNOLOGY UNIVERSITY MYSURU

Submitted for the fulfillment of the coding lab component for the course Computer Networks

(EC620L)

Submitted by

NAME	Mohamed Farhan Fazal
USN	01JST18EC055

Under the guidance of

Vinayprasad M S & Chandrashekar Murthy B N

Assistant Professors

DEPARTMENT OF ELECTRONICS AND COMMUNICATION

SJCE MYSORE - 570006

Table of content

Dijsktra's Algorithm

Introduction

Dijkstra's algorithm is very similar to Prim's algorithm for a minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with a given source as root. We maintain two sets, one set contains vertices included in the shortest path tree, the other set includes vertices not yet included in the shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest path tree, whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While sptSet doesn't include all vertices.
    a. Pick a vertex u which is not there in sptSet and has minimum distance value.
    b. Include u to sptSet.
    c. Update the distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance

value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Code

```
import sys
from terminaltables import SingleTable
from common import clear


class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printFinalDistances(self,src, dist):
        clear()
        self.printGraph()
        sourceInfo = ["Source"], [src]
        sourceTable = SingleTable(sourceInfo)
        print(sourceTable.table)

                data  =  ["Vertex"]+list(range(self.V)),  ["Distance  from
source"]+dist
        table = SingleTable(data)
        print(table.table)

    def printGraph(self):
        table = SingleTable(self.graph)
        table.inner_row_border = True
        print(table.table)


    def findMinimumDistance(self, dist, sptSet):
        min = sys.maxsize
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v
        return min_index

    def dijkstra(self, src):
        dist = [sys.maxsize] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):
            u = self.findMinimumDistance(dist, sptSet)
            sptSet[u] = True
            for v in range(self.V):
                    if self.graph[u][v] > 0 and sptSet[v] == False and
dist[v] > dist[u] + self.graph[u][v]:
                    dist[v] = dist[u] + self.graph[u][v]
```

```
        self.printFinalDistances(src,dist)


g = Graph(9)
g.graph = [
    [0, 4, 0, 0, 0, 0, 8, 0],
    [4, 0, 8, 0, 0, 0, 0, 11, 0],
    [0, 8, 0, 7, 0, 4, 0, 0, 2],
    [0, 0, 7, 0, 9, 14, 0, 0, 0],
    [0, 0, 0, 9, 0, 10, 0, 0, 0],
    [0, 0, 4, 14, 10, 0, 2, 0, 0],
    [0, 0, 0, 0, 0, 2, 0, 1, 6],
    [8, 11, 0, 0, 0, 0, 1, 0, 7],
    [0, 0, 2, 0, 0, 0, 6, 7, 0]
]

g.dijkstra(int(input("Enter the source node: ")))
```

Result

```
Enter the source node: 0
```

0	4	0	0	0	0	0	8	0
4	0	8	0	0	0	0	11	0
0	8	0	7	0	4	0	0	2
0	0	7	0	9	14	0	0	0
0	0	0	9	0	10	0	0	0
0	0	4	14	10	0	2	0	0
0	0	0	0	0	2	0	1	6
8	11	0	0	0	0	1	0	7
0	0	2	0	0	0	6	7	0

Source
0

Vertex	0	1	2	3	4	5	6	7	8
Distance from source	0	4	12	19	21	11	9	8	14

Prim’s Algorithm

Introduction

Like Kruskal’s algorithm, Prim’s algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two sets of vertices in a graph is called cut in graph theory. So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

Algorithm

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.
3. While mstSet doesn’t include all vertices.
    a. Pick a vertex u which is not there in mstSet and has minimum key value.
    b. Include u to mstSet.
    c. Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the

previous key value of v, update the key value as weight of u-v

The idea of using key values is to pick the minimum weight edge from the cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Code

```
import sys  # Library for INT_MAX
from common import clear, custom_print
from terminaltables import SingleTable


class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printMST(self, parent):
        clear()
        self.printGraph()
        data = [["Start Edge", "End Edge", "Weight"]] + [[parent[i],
                                                          i,
self.graph[i][parent[i]]] for i in range(1, self.V)]
        table = SingleTable(data)
        table.inner_row_border = True
        print(table.table)

    def printFinalDistances(self, src, dist):
        clear()
        self.printGraph()
        sourceInfo = ["Source"], [src]
        sourceTable = SingleTable(sourceInfo)
        print(sourceTable.table)

                data  =  ["Vertex"]+list(range(self.V)),  ["Distance  from
source"]+dist
        table = SingleTable(data)
        print(table.table)

    def printGraph(self):
        table = SingleTable(self.graph)
        table.inner_row_border = True
        print(table.table)

    def minKey(self, key, mstSet):

        min = sys.maxsize
```

```
        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    def primMST(self, src):

        key = [sys.maxsize] * self.V
        parent = [None] * self.V
        key[0] = src
        mstSet = [False] * self.V

        parent[0] = -1

        for cout in range(self.V):

            u = self.minKey(key, mstSet)

            mstSet[u] = True

            for v in range(self.V):

                    if self.graph[u][v] > 0 and mstSet[v] == False and
key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.printMST(parent)


g = Graph(5)
g.graph = [[0, 2, 0, 6, 0],
           [2, 0, 3, 8, 5],
           [0, 3, 0, 0, 7],
           [6, 8, 0, 0, 9],
           [0, 5, 7, 9, 0]]

g.primMST(2)
```

Result

0	2	0	6	0
2	0	3	8	5
0	3	0	0	7
6	8	0	0	9
0	5	7	9	0

Start Edge	End Edge	Weight
0	1	2
1	2	3
0	3	6
1	4	5

Kruskal's Algorithm

Introduction

This algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

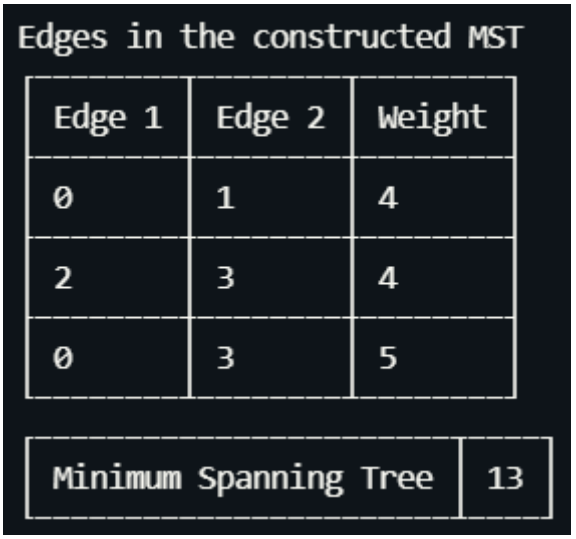
Code

```
from terminaltables import SingleTable
from common import clear


class Graph:

    def __init__(self, vertices):
        self.V = vertices
        self.graph = []
```

```
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    def KruskalMST(self):
        result = []
        i = 0
        e = 0
        self.graph = sorted(self.graph,
                            key=lambda item: item[2])

        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        minimumCost = 0
        clear()
        print("Edges in the constructed MST")
        data = [["Edge 1", "Edge 2", "Weight"]]
        for u, v, weight in result:
            minimumCost += weight
            data.append([u, v, weight])
        table = SingleTable(data)
        table.inner_row_border = True
        print(table.table)
                        print(SingleTable([["Minimum   Spanning   Tree",
minimumCost]]).table)
```

```
clear()
number_of_nodes = int(input("Enter the number of nodes: "))
number_of_edges = int(input("Enter the number of edges: "))
g = Graph(number_of_nodes)
for edge in range(number_of_edges):
    clear()
    g.addEdge(int(input("Enter from edge: ")), int(
        input("Enter to edge: ")), int(input("Enter edge weight: ")))

g = Graph(4)
g.addEdge(0, 1, 4)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.KruskalMST()
```

Result



Bellman Ford Algorithm

Introduction

Given a graph and a source vertex src in the graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges. Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs.

Bellman-Ford is also simpler than Dijkstra and suits well for distributed systems. But the time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

Algorithm

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1. This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2. This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in a given graph.
   Do following for each edge u-v

   If dist[v] > dist[u] + weight of edge uv, then update dist[v]

   dist[v] = dist[u] + weight of edge uv
3. This step reports if there is a negative weight cycle in the graph.
   Do following for each edge u-v

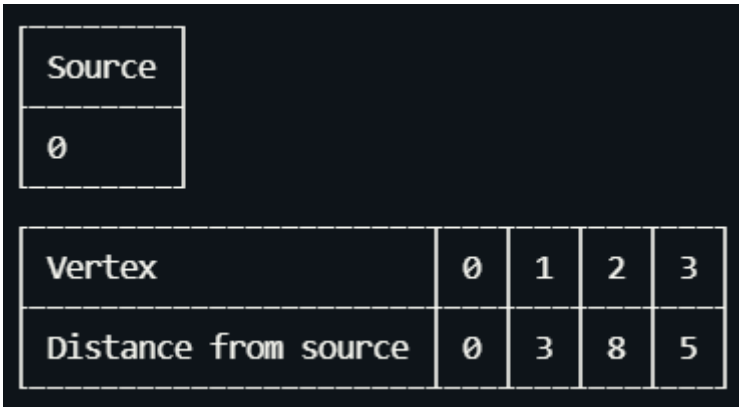
   If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"

The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

Code

```
from terminaltables import SingleTable
from common import clear
class Graph:

    def __init__(self, vertices):
        self.V = vertices # No. of vertices
        self.graph = []

    # function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # utility function used to print the solution
    def printArr(self,src, dist):
        clear()
        sourceInfo = ["Source"], [src]
        sourceTable = SingleTable(sourceInfo)
        print(sourceTable.table)

                data  =  ["Vertex"]+list(range(self.V)),  ["Distance  from
source"]+dist
        table = SingleTable(data)
        print(table.table)

    def BellmanFord(self, src):
        dist = [float("Inf")] * self.V
        dist[src] = 0
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                        dist[v] = dist[u] + w
        for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                        print("Graph contains negative weight cycle")
                        return
        self.printArr(src,dist)

g = Graph(4)
g.addEdge(0,1,4)
g.addEdge(0,3,5)
g.addEdge(1,3,4)
g.addEdge(3,2,3)
g.addEdge(2,1,-5)

g.BellmanFord(int(input("Enter the source node: ")))
```

Result



Source
0

Vertex	0	1	2	3
Distance from source	0	3	8	5

CRC - Cyclic Redundancy Check

Introduction

A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents. On retrieval, the calculation is repeated and, in the event the check values do not match, corrective action can be taken against data corruption. CRCs can be used for error detection and correction.

CRC uses Generator Polynomial which is available on both sender and receiver sides. An example generator polynomial is of the form like $x^3 + 1$. This generator polynomial represents key **1001**. Another example is $x^2 + x$ that represents key **110**.

Algorithm

Sender Side

1. The task is to send a string data to the receiver side.
2. The sender sends a string.
3. First, this string is converted to a binary string "1000101101011010011110".
   The key is known to both the side sender and receiver.
4. This data is encoded using the CRC code using the key in the sender side.
5. This encoded data is sent to the receiver.
6. Receiver later decodes the encoded data string to verify whether there was any error or not.

Receiver Side

1. The receiver receives the encoded data string from the sender.
2. Receiver with the help of the same **key** decodes the data and finds out the remainder.
3. If the **remainder** is **zero** then it means there is **no error** in data sent by the sender to the receiver.
4. If the **remainder** comes out to be **non-zero** it means there was an **error**, a **negative acknowledgement** is sent to the sender. The sender then resends the data until the receiver receives correct data.

Code

Sender Side

```
import socket

def xor(a, b):

    # initialize result
    result = []
    for i in range(1, len(b)):
        if a[i] == b[i]:
            result.append('0')
        else:
            result.append('1')
    return ''.join(result)

def bitstring_to_bytes(s):
    v = int(s, 2)
    b = bytearray()
    while v:
        b.append(v & 0xff)
        v >>= 8
    return bytes(b[::-1])


# Performs Modulo-2 division
def mod2div(divident, divisor):
    pick = len(divisor)
    tmp = divident[0 : pick]
    while pick < len(divident):
        if tmp[0] == '1':
            tmp = xor(divisor, tmp) + divident[pick]
        else:
            tmp = xor('0'*pick, tmp) + divident[pick]
        pick += 1
    if tmp[0] == '1':
```

```
        tmp = xor(divisor, tmp)
    else:
        tmp = xor('0'*pick, tmp)
    checkword = tmp
    return checkword

def encodeData(data, key):

    l_key = len(key)

    # Appends n-1 zeroes at end of data
    appended_data = data + '0'*(l_key-1)
    remainder = mod2div(appended_data, key)

    # Append remainder in the original data
    codeword = data + remainder
    return codeword

# Create a socket object
s = socket.socket()

# Define the IP & PORT on which you want to connect
PORT = 12345
IP = '127.0.0.1'

# connect to the server on local computer
s.connect((IP, PORT))

input_string = input("Enter data you want to send: ")
key = input("Enter the polynomial key: ")

data = (''.join(format(ord(x), 'b') for x in input_string))
print(data)

ans = encodeData(data,key)
print(ans)
s.sendall(bytes(ans, "utf-8"))


# receive data from the server
print (s.recv(1024).decode("utf-8"))

# close the connection
s.close()
```

Receiver Side

```
import socket

def xor(a, b):
    result = []
    for i in range(1, len(b)):
        if a[i] == b[i]:
            result.append('0')
        else:
            result.append('1')
    return ''.join(result)
```

```
# Performs Modulo-2 division
def mod2div(divident, divisor):
    pick = len(divisor)
    tmp = divident[0: pick]

    while pick < len(divident):
        if tmp[0] == '1':
            tmp = xor(divisor, tmp) + divident[pick]

        else:
            tmp = xor('0'*pick, tmp) + divident[pick]
        pick += 1
    if tmp[0] == '1':
        tmp = xor(divisor, tmp)
    else:
        tmp = xor('0'*pick, tmp)

    checkword = tmp
    return checkword


def decodeData(data, key):

    l_key = len(key)

    # Appends n-1 zeroes at end of data
    appended_data = data + '0'*(l_key-1)
    remainder = mod2div(appended_data, key)

    return remainder


# Creating Socket
s = socket.socket()
print("Socket successfully created")

# reserve a port on your computer in our
# case it is 12345 but it can be anything
PORT = 12345
IP = "127.0.0.1"

s.bind(('', PORT))
print("socket binded to %s" % (PORT))
# put the socket into listening mode
maxConnections = 5
s.listen(maxConnections)
print("socket       is       listening       with       maximum       {}
connections".format(maxConnections))

key = input("Enter the polynomial key: ")

while True:
    connection, address = s.accept()
    print('Got connection from', address)

    # Get data from client
    data = connection.recv(1024)
    print(data)
```

```
    data = data.decode("utf-8")

    print(data)

    if not data:
        break

    ans = decodeData(data, key)
    # print("Remainder after decoding is->"+ans)

    # If remainder is all zeros then no error occurred
    temp = "0" * (len(key) - 1)
    if ans == temp:
        connection.sendall(bytes("Data: " + data + " Received. No error
FOUND", "utf-8"))
    else:
            connection.sendall(bytes("The received data is corrupted.",
"utf-8"))

    connection.close()
```

Result

Receiver Side

```
PS D:\CPT\11. Socket + CRC\CRC> python .\receiver.py
Socket successfully created
socket binded to 12345
socket is listening with maximum 5 connections
Enter the polynomial key: 110011
Got connection from ('127.0.0.1', 60997)
b'1001000110010111011001101100110111110000011001101110010110111111101101100000111010011010001100101
1100000111001111001011101110110010011001011110010100000111001111010011100100110010110111010000010
1001111001011101110110010011010011101110110011110000011001001100001111010011000011000001110111110
1001111010011010001000001000011101001010000111001'
1001000110010111011001101100110111110000011001101110010110111111101101100000111010011010001100101
0000011100111100101110111011001001100101111001010000011100111101001110010011001011011101000001010
0111100101110111011001001101001110111011001111000001100100110000111101001100001100000111011111010
0111101001101000100000100001110100101000011110010
PS D:\CPT\11. Socket + CRC\CRC> python.exe .\sender.py
Enter data you want to send: Hello from the sender side. Sending data with CRC
Enter the polynomial key: 110011
1001000110010111011001101100110111110000011001101110010110111111101101100000111010011010001100101
0000011100111100101110111011001001100101111001010000011100111101001110010011001011011101000001010
0111100101110111011001001101001110111011001111000001100100110000111101001100001100000111011111010
0111101001101000100000100001110100101000011
1001000110010111011001101100110111110000011001101110010110111111101101100000111010011010001100101
0000011100111100101110111011001001100101111001010000011100111101001110010011001011011101000001010
0111100101110111011001001101001110111011001111000001100100110000111101001100001100000111011111010
0111101001101000100000100001110100101000011110010
Data: 1001000110010111011001101100110111110000011001101110010110111111101101100000111010011010001100
1010110000011100111100101110111011001001100101111001010000011100111101001110010011001011011101000
0010100111100101110111011001001101001110111011001111000001100100110000111101001100001100000111011
1110100111101001101000100000100001110100101000011110010 Received. No error FOUND
```

Sockets

Introduction

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while another socket reaches out to the other to form a connection. Server forms the listener socket while the client reaches out to the server.

They are the real backbones behind web browsing. In simpler terms there is a server and a client.

Code

Client Side

```
import socket

IP = "127.0.0.1"

s = socket.socket()
s.connect((IP, 9945))

received = ''

while True:
    msg = s.recv(4096)
    if (len(msg) <= 0):
        break

    received += msg.decode("utf-8")

print(received)
```

Server Side

```
import socket

IP = "127.0.0.1"

s = socket.socket()
s.bind((IP, 9945))

print("IP address of the server is {}".format(IP))
```

```
s.listen()
message = input("Enter the message to send to the receiver: ")

while True:
    connection, clientAddress = s.accept()
    print(f"Connection to {clientAddress} establised")
    connection.send(bytes(message, "utf-8"))
    connection.close()
```

Result

Server Side

```
PS D:\CPT\11. Socket + CRC\simpleSocket> python.exe .\server.py
IP address of the server is 127.0.0.1
Enter the message to send to the receiver: Hello from server listening on
port 9945
Connection to ('127.0.0.1', 50388) establised
```

Client Side

```
PS D:\CPT\11. Socket + CRC\simpleSocket> python .\client.py
Hello from server listening on port 9945
```
