Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías

Licenciatura en Ingeniería en Computación

Arquitectura de Computadoras

Mtro. Jorge Ernesto López Arce Delgado

Actividad: Optimización de transferencia de Archivos en una VPN con Algoritmos Voraces

Adrián Arturo García López (220577088): Administrador de Proyecto
Javier Alberto Galindo Parra (215422122): Configuración VPN, Script latencia
David de Jesús Pánuco Rodríguez (219704815): Protocolo Comunicación, Kruskal
Hugo Julian Valadez Paez (219519724): Dijkstra, Interfaz gráfica

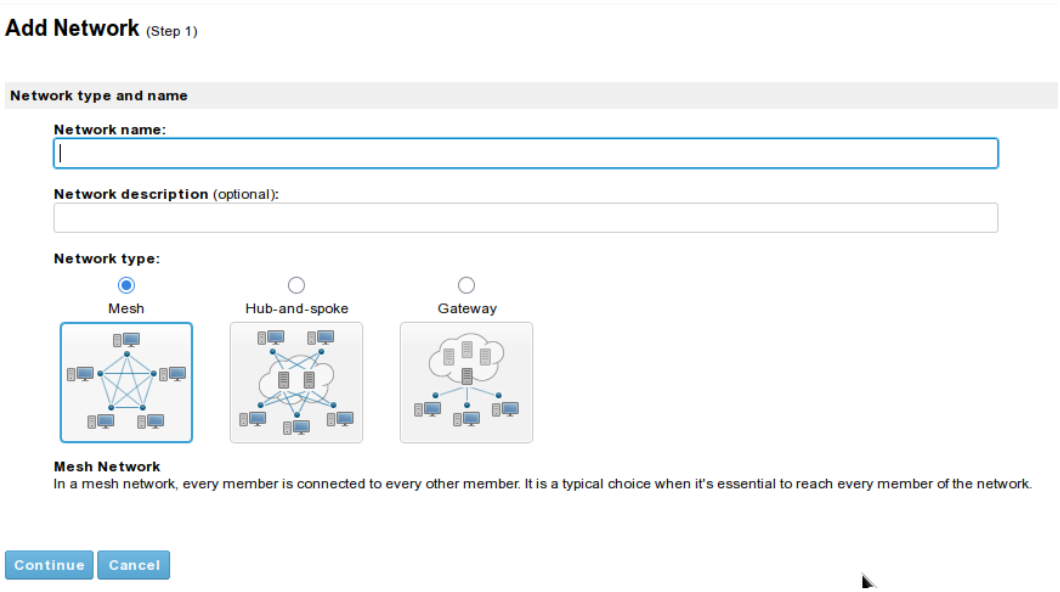
---

Guadalajara, Jalisco, 14 de mayo de 2025

Parte 1: Configuración de la VPN

El objetivo principal de esta parte fue crear una red virtual a la que estuvieran conectados todos nuestros dispositivos, de forma que tuvieran direcciones IP estáticas.

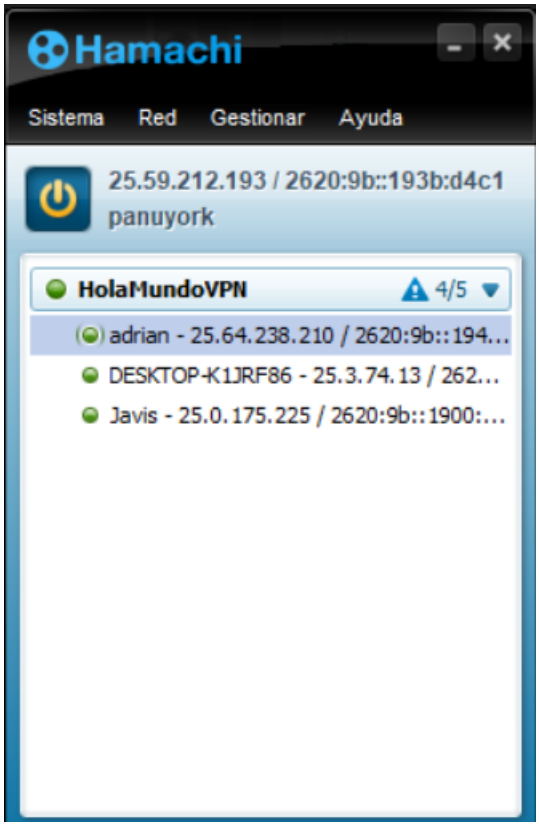
Como software de vpn utilizamos Hamachi[https://vpn.net]. Para crear la red solo tienes que crear una cuenta en la página, iniciar sesión, y hacer click en el botón de *Add Network*.



Configuración de la red

Una vez que se considera el nombre y la contraseña de la red. Solo se tiene que instalar el cliente de Hamachi en cada dispositivo e unirse a la red.

Nos conectamos todos a la red. No se requirió configuración adicional, ya que Hamachi siempre utiliza direcciones fijas para los dispositivos.



Captura de pantalla de la red

El protocolo implementado funciona de la siguiente manera:

Existen 3 tipos de mensajes:

- `OP_SEND (0x02)`

  Propósito: el cliente envía (sube) un archivo al servidor.

Campo	Tamaño	Descripción
Opcode	1 byte	0x02
longitud nombre	4 bytes	longitud del nombre del archivo (`uint32`)
nombre	N bytes	nombre del archivo
tamaño archivo	8 bytes	número de bytes en el archivo (`uint64`)
datos archivo	M bytes	contenido del archivo

- `OP_RELAY (0x03)`

  Propósito: El cliente solicita al servidor que reenvíe un archivo a otro nodo de la red.

Campo	Tamaño	Descripción
Opcode	1 byte	0x03
longitud nombre	4 bytes	longitud del nombre del archivo (`uint32`)
nombre	N bytes	nombre del archivo
número nodos	1 byte	número de nodos en la ruta (`uint8`)
lista nodos	K ×22 bytes	entradas `IP:PUERTO` de 22 bytes cada una
tamaño archivo	8 bytes	número de bytes en el archivo (`uint64`)
datos archivo	M bytes	contenido del archivo

- `OP_RESPONSE (0x04)`

  Propósito: Confirmar el resultado de una operación anterior (éxito o error).

Campo	Tamaño	Descripción
Opcode	1 byte	0x04
Resultado	1 byte	0x00 = OK, 0x01 = Error
longitud mensaje	4 bytes	longitud del mensaje de estado (`uint32`)
mensaje	N bytes	mensaje de estado

Estos mensajes se envían del servidor al cliente por medio de *TCP*. Se eligió este protocolo para la trasmisión de los datos, ya que garantiza la integridad de los datos enviados y recibidos.

El protocolo se implementó en los archivos `server.py` y `cliente.py`.

la función principal del servidor es `handle_client`: Esta función recibe el socket conectado y su dirección, e implementa la parte del servidor de los mensajes descritos anteriormente

Esta función interpreta los distintos mensajes explicados en el protocolo. Para el tipo de mensaje `OP_SEND` recibe el archivo enviado por el cliente. Para el tipo de mensaje `OP_RELAY` recibe el archivo enviado por el cliente, y lo envía al siguiente nodo con el tipo de mensaje apropiado.

```
def handle_client(conn, addr):
    print(f"[+] Conectado: {addr}")
    try:
        while True:
            op = conn.recv(1)
            if not op:
                break
            op = ord(op)

            if op == OP_SEND:
                name_len = struct.unpack('>I', recv_all(conn, 4))[0]
                filename = recv_all(conn, name_len).decode()
                filesize = struct.unpack('>Q', recv_all(conn, 8))[0]
                with open(filename, 'wb') as f:
```

```
                received = 0
                while received < filesize:
                    chunk = conn.recv(min(4096, filesize - received))
                    if not chunk:
                        break
                    f.write(chunk)
                    received += len(chunk)
            print(f"[+] Archivo recibido: {filename} ({filesize}
            ↪  bytes)")

            # Enviar confirmación
            msg = "Archivo recibido correctamente"
            conn.sendall(struct.pack('B', OP_RESPONSE))
            conn.sendall(struct.pack('B', 0x00))  # Código 0x00 = OK
            conn.sendall(struct.pack('>I', len(msg)) + msg.encode())


        elif op == OP_RELAY:
            # Read filename
            name_len = struct.unpack('>I', recv_all(conn, 4))[0]
            filename = recv_all(conn, name_len).decode()

            # Read nodes list
            node_count = ord(recv_all(conn, 1))
            nodes = []
            for _ in range(node_count):
                ip_port = recv_all(conn, 22).decode().strip()
                nodes.append(ip_port)

            # Read file size
            filesize = struct.unpack('>Q', recv_all(conn, 8))[0]

            # Pop off the next hop
            next_ip_port = nodes.pop(0)
            next_ip, next_port = next_ip_port.split(':')
            next_port = int(next_port)

            try:
                with socket.create_connection((next_ip, next_port)) as
                ↪  s:
                    if nodes:
```

```
            # More hops remain → forward as OP_RELAY
            print(f"[*] Relaying to {next_ip}:{next_port}
            ↪  (relay, {len(nodes)} left)")
            s.sendall(struct.pack('B', OP_RELAY))
            # filename
            s.sendall(struct.pack('>I', len(filename)) +
            ↪  filename.encode())
            # new node count
            s.sendall(struct.pack('B', len(nodes)))
            # remaining node list
            for ip_p in nodes:
                # pad/truncate each to 22 bytes if needed
                s.sendall(ip_p.ljust(22).encode())
            # file size
            s.sendall(struct.pack('>Q', filesize))
        else:
            # This is the last hop → send as OP_SEND
            print(f"[*] Relaying to {next_ip}:{next_port}
            ↪  (final send)")
            s.sendall(struct.pack('B', OP_SEND))
            s.sendall(struct.pack('>I', len(filename)) +
            ↪  filename.encode())
            s.sendall(struct.pack('>Q', filesize))

        # Stream the file bytes through
        bytes_remaining = filesize
        while bytes_remaining > 0:
            chunk = recv_all(conn, min(4096,
            ↪  bytes_remaining))
            s.sendall(chunk)
            bytes_remaining -= len(chunk)

        # Wait for response back from downstream...
        op_code = ord(recv_all(s, 1))
        if op_code != OP_RESPONSE:
            raise Exception("Nodo destino no respondió
            ↪  correctamente")

        cod = ord(recv_all(s, 1))
        msg_len = struct.unpack('>I', recv_all(s, 4))[0]
        msg = recv_all(s, msg_len).decode()
```

```
                        if cod == 0x00:
                            print(f"[OK] Confirmación del siguiente nodo:
                            ↪ {msg}")
                            conn.sendall(struct.pack('B', OP_RESPONSE))
                            conn.sendall(struct.pack('B', 0x00))
                            conn.sendall(struct.pack('>I', len(msg)) +
                            ↪ msg.encode())
                        else:
                            print(f"[ERROR] Nodo intermedio falló: {msg}")
                            conn.sendall(struct.pack('B', OP_RESPONSE))
                            conn.sendall(struct.pack('B', 0x01))
                            conn.sendall(struct.pack('>I', len(msg)) +
                            ↪ msg.encode())

                except Exception as e:
                    error_msg = f"Fallo al retransmitir: {e}"
                    print(f"[!] {error_msg}")
                    conn.sendall(struct.pack('B', OP_RESPONSE))
                    conn.sendall(struct.pack('B', 0x01))
                    conn.sendall(struct.pack('>I', len(error_msg)) +
                    ↪ error_msg.encode())


    except Exception as e:
        print(f"[!] Error: {e}")
    finally:
        conn.close()
```

Las funciones principales del cliente son `enviar_archivo` y `enviar_por_ruta`:

`enviar_archivo`: Esta función recibe un `host`, un `puerto`, y envía el `archivo` en esa dirección y puerto.

```
def enviar_archivo(host, puerto, archivo):
    nombre = os.path.basename(archivo)
    tamano = os.path.getsize(archivo)
    datos = open(archivo, 'rb').read()

    with socket.create_connection((host, puerto)) as sock:
        print(f"[INFO] Conectado a {host}:{puerto}")
```

```
        header = struct.pack("!B I {}s Q".format(len(nombre)),
                             OP_SEND,
                             len(nombre),
                             nombre.encode(),
                             tamano)
        sock.sendall(header + datos)
        leer_confirmacion(sock)
```

`enviar_por_ruta`: Esta función recibe un `host`, un `puerto`, y una `ruta`, y envía el `archivo` en esa dirección y puerto, solicitándole al servidor que continue transmitiendo el archivo por la ruta dada.

```
def enviar_por_ruta(host, puerto, archivo, ruta):
    nombre = os.path.basename(archivo)
    nombre_bytes = nombre.encode()
    num_nodos = len(ruta)
    tamano = os.path.getsize(archivo)

    with open(archivo, 'rb') as f:
        datos = f.read()

    with socket.create_connection((host, puerto)) as sock:
        print(f"[INFO] Conectado a {host}:{puerto} para retransmisión")

        sock.sendall(struct.pack("!B", OP_RELAY))
        sock.sendall(struct.pack(">I", len(nombre_bytes)))
        sock.sendall(nombre_bytes)
        sock.sendall(struct.pack("B", num_nodos))

        for nodo in ruta:
            nodo_bytes = nodo.encode().ljust(22, b' ')
            sock.sendall(nodo_bytes)

        sock.sendall(struct.pack(">Q", tamano))
        sock.sendall(datos)

        leer_confirmacion(sock)
```

Parte 2: Medición de Métricas de Red

El objetivo principal de esta parte fue crear una herramienta que permitiera medir la latencia y el ancho de banda entre varios nodos de una red.

Las IPs de los nodos se leen desde un archivo de texto que se especifica desde la línea de comandos con la opción `--nodos`.

Se utiliza la librería `argparse` para que el usuario pueda configurar los siguientes valores:

- `--nodos`: El archivo con las ip's los nodos en la vpn.
- `--local`: La ip local del nodo.
- `--puerto-iperf`: El puerto para conectarse con iperf3.
- `--duracion-iperf`: La duración de la prueba de ancho de banda.
- `--conteo-ping`: Cuántos pings enviar para medir la latencia.

Las funciones principales son las siguientes:

- `medir_latencia`:

  Esta función utiliza el comando `ping`, e interpreta su salida de consola para determinar la latencia promedio entre 2 nodos.

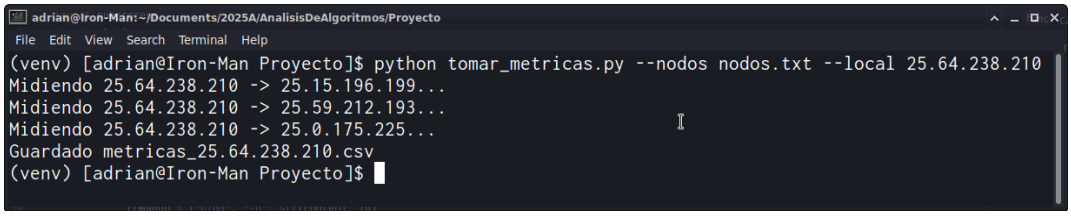
```python
def medir_latencia(ip, conteo=4):
    try:
        sistema = platform.system()
        if sistema == "Windows":
            comando = ["ping", "-n", str(conteo), ip]
        else:
            comando = ["ping", "-c", str(conteo), ip]

        resultado = subprocess.run(comando, capture_output=True,
        ↪  text=True, encoding='cp850', errors='ignore')

        for linea in resultado.stdout.splitlines():
            if sistema == "Windows" and ("Media" in linea or "media"
            ↪  in linea):
                # Extrae el último número antes de 'ms'
                numeros = re.findall(r'\d+', linea)
                if numeros:
                    return float(numeros[-1])
            elif sistema != "Windows" and ("rtt min" in linea or
            ↪  "round-trip" in linea):
                partes = linea.split(' = ')[1].split(' ')[0].split('/')
                return float(partes[1])

    except subprocess.CalledProcessError:
```

```
        return None
```

- `medir_ancho_banda`

  Esta función utiliza el comando `iperf3`, e interpreta su salida de consola para medir el ancho de banda promedio entre 2 nodos.

```
def medir_ancho_banda(ip, puerto=5201, duracion=10):
    try:
        resultado = subprocess.run([
            "iperf3", "-c", ip, "-p", str(puerto), "-t", str(duracion),
            ↪  "-f", "m"
        ], capture_output=True, text=True, check=True)

        for linea in reversed(resultado.stdout.splitlines()):
            if "sender" in linea and "Mbits/sec" in linea:
                campos = linea.split()
                for i, campo in enumerate(campos):
                    if campo == "Mbits/sec" and i > 0:
                        try:
                            ancho = float(campos[i - 1])
                            return ancho
                        except ValueError:
                            continue
    except subprocess.CalledProcessError:
        return None
```



Ejemplo de la ejecución del programa

Las mediciones se guardan en un archivo `metricas_{ip_local}.csv` con el siguiente formato:

Origen	Destino	Latencia [ms]	Ancho de Banda [Mbps]

Parte 3: Implementación Dijkstra "File Transfer Optimizer" (con GUI)

El objetivo principal de esta parte es implementar un sistema que calcule la ruta óptima para transferir archivos entre nodos de una red virtual (VPN), utilizando el algoritmo de Dijkstra

El algoritmo de Dijkstra esta implementado en el archivo dijkstra.py

`dijkstra`

Esta función encuentra la ruta de menor costo en el grafo `grafo`, desde el nodo `origen` hasta el nodo `destino`.

El algoritmo realiza varias iteraciones, seleccionando un nodo en cada una. En la primera iteración elige el nodo origen, y a partir de la segunda, elige el nodo no visitado alcanzable con menor costo.

En cada iteración, agrega a una cola de prioridad todos los vertices alcanzables desde el nodo seleccionado con el costo en el que se alcanzaría si se llega desde el nodo seleccionado. También guarda el mejor nodo previo para llegar a cada nodo.

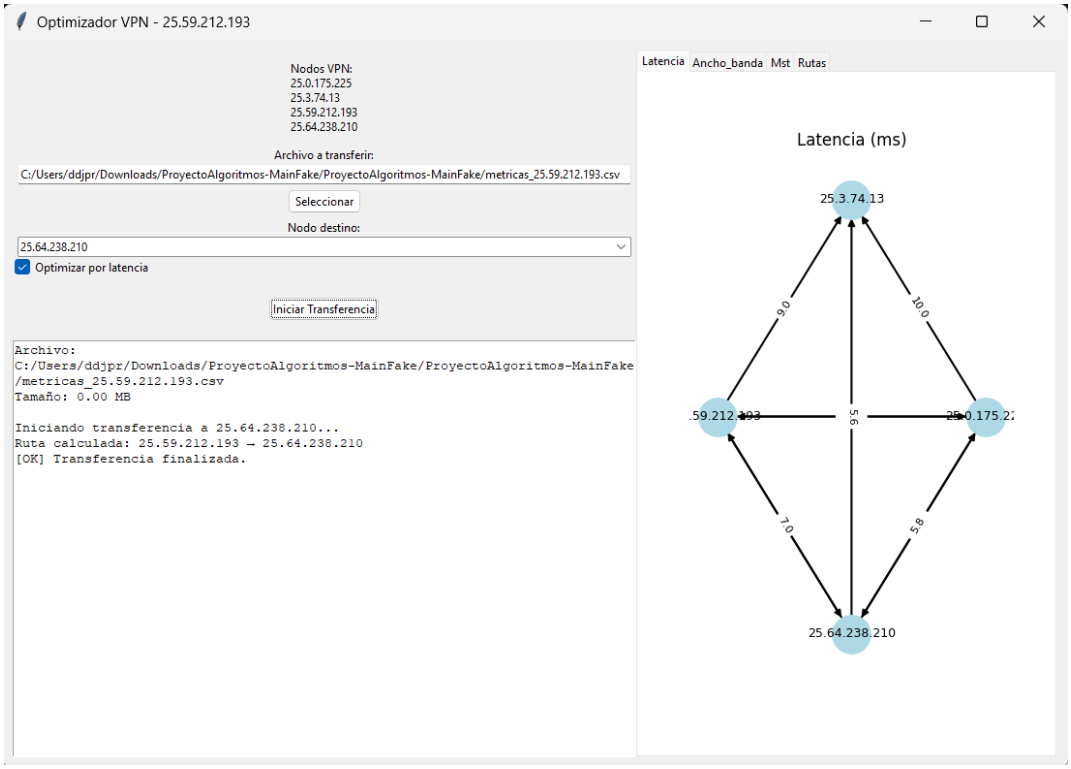
La función se detiene una vez que alcanza el nodo destino.

Después reconstruye el camino utilizando la información de nodos previos guardada durante las iteraciones.

```python
def dijkstra(grafo, origen, destino):
    distancias = {nodo: float('inf') for nodo in grafo.nodes}
    previos = {nodo: None for nodo in grafo.nodes}
    distancias[origen] = 0

    # Cola de prioridad con heapq
    heap = [(0, origen)]
    visitados = set()

    while heap:
        distancia_actual, nodo_actual = heapq.heappop(heap)

        if nodo_actual in visitados:
            continue
        visitados.add(nodo_actual)

        if nodo_actual == destino:
            break

        for vecino in grafo[nodo_actual]:
            peso = grafo[nodo_actual][vecino].get('weight', 1)
            nueva_dist = distancia_actual + peso
            if nueva_dist < distancias[vecino]:
                distancias[vecino] = nueva_dist
                previos[vecino] = nodo_actual
                heapq.heappush(heap, (nueva_dist, vecino))
```

```
    # Reconstrucción del camino
    if distancias[destino] == float('inf'):
        return None, float('inf')

    camino = []
    nodo = destino
    while nodo:
        camino.insert(0, nodo)
        nodo = previos[nodo]

    return camino, distancias[destino]
```

Esta fue la GUI que se implementó:



Interfaz de usuario en funcionamiento

Cuando se inicia se encarga de iniciar el servidor de `iperf3` (usado para la toma de mediciones de latencia y ancho de banda), así como el servidor para la recepción y el reenvío de archivos. Al iniciar lee todos los archivos csv cuyo nombre comience con "métricas" y los combina en un solo grafo.

La interfaz te permite elegir un nodo al que enviar un archivo, elegir qué archivo enviar, enviarlo, así como visualizar los grafos generados y la ruta utilizada para la última transferencia realizada.

Parte 4: Implementación de Kruskal "Topología Eficiente"

El objetivo principal de esta parte es implementar el algoritmo de Kruskal para obtener un árbol de expansión mínima (MST), en el que el camino entre cualesquiera dos nodos maximize el mínimo ancho de banda.

`kruskal`

Esta función primero ordena los aristas de menor a mayor peso.

Se mantiene una serie de conjuntos de nodos ya conectados por aristas.

El algoritmo verifica que cada arista conecte conjuntos que no habían sido conectados aún. De ser el caso el arista forma parte del MST.

```
import networkx as nx

def kruskal(grafo):
    # Lista de aristas con sus pesos reales
    edges = [(u, v, d['weight']) for u, v, d in
    ↪  grafo.to_undirected().edges(data=True)]
    edges.sort(key=lambda x: x[2])  # Ordenar por peso

    parent = {}
    def find(n):
        while parent[n] != n:
            parent[n] = parent[parent[n]]
            n = parent[n]
        return n

    def union(a, b):
        ra, rb = find(a), find(b)
        if ra != rb:
            parent[rb] = ra
            return True
        return False

    # Inicializar conjuntos
    for nodo in grafo.nodes:
        parent[nodo] = nodo

    mst_edges = []
    for u, v, peso in edges:
        if union(u, v):
```

```
            mst_edges.append((u, v, peso))

    # Construir nuevo grafo MST
    mst = nx.Graph()
    for nodo in grafo.nodes:
        mst.add_node(nodo)

    for u, v, peso in mst_edges:
        datos_originales = grafo.get_edge_data(u, v) or
        ↪  grafo.get_edge_data(v, u)
        mst.add_edge(u, v, **datos_originales)

    return mst
```

GitHub

Todo el código generado para este proyecto se encuentra disponible en el repositorio:

https://github.com/AdrArtGar/ProyectoAlgoritmos

Conclusión

En este trabajo se logró diseñar e implementar exitosamente una aplicación con interfaz gráfica que permite a usuarios de una red enviar y recibir archivos entre ellos, con la posibilidad de utilizar nodos intermediarios en la red y algoritmos como Dijkstra o Kruskal para optimizar el envío.

Para la comprobación de los resultados y durante el desarrollo, Hamachi resultó Súmamente útil, ya que nos permitió realizar todas las pruebas y demostraciones necesarias sin ser difícil de configurar ni requerir una inversión monetaria.

Para la implementación de este proyecto, el Algoritmo de Dijkstra fue ideal para la optimización de la ruta para la transferencia sobre el grafo generado, se pudo aplicar directamente a la situación de una red de nodos conectados por aristas con diferentes costos (latencias) entre sí. Kruskal también fue útil, sin embargo no pudimos aplicarlo directamente sobre el grafo de ancho de banda, ya que Kruskal conserva los aristas con menor costo, pero queríamos los aristas con mayor ancho de banda. Por lo que tuvimos que utilizar el inverso del ancho de banda para aplicar el algoritmo.

La aplicación que implementamos terminó siendo satisfactoria y útil para demostrar los conceptos aplicados. Aunque hay características que hubiera sido deseable agregar, nos permitió realizar todas las pruebas necesarias.