

*Laboratorio 3 - Primera parte*  
Algoritmos de Enrutamiento

*Stefano Alberto Aragoni Maldonado (20261)*  
*Carol Andreé Arévalo Estrada (20461)*  
*Fátima Priscilla González Sandoval (20689)*

## 1. Introducción

En esta práctica se estudió e implementó diferentes algoritmos de enrutamiento, con el propósito de simular el proceso de envío de mensajes mediante routers de una red. Inicialmente se seleccionaron tres algoritmos específicos: *Dijkstra Link State*, *Flooding*, y *Distance Vector Routing*.

Una fase crucial de este laboratorio consistió en investigar detalladamente el funcionamiento de cada uno de los algoritmos. Este proceso permitió adquirir un conocimiento sólido sobre cómo cada algoritmo aborda el desafío de determinar las rutas más eficientes para dirigir los mensajes a sus destinos finales. Este conocimiento previo fue esencial para llevar a cabo la implementación de los algoritmos utilizando el lenguaje de programación Python. Para finalizar, se realizó una serie de pruebas con el propósito de evaluar el funcionamiento correcto de estos algoritmos.

En conclusión, esta práctica proporcionó una comprensión profunda de los algoritmos de enrutamiento y su aplicación en las redes de comunicación, al tiempo que nos permitió comprender la importancia de estas en nuestro día a día.

## 2. Objetivos

- a. Conocer los algoritmos de enrutamiento utilizados en las implementaciones actuales de Internet.
- b. Comprender cómo funcionan las tablas de enrutamiento.
- c. Implementar los algoritmos de enrutamiento y probarlos.

## 3. Descripción

Como se mencionó anteriormente, en este laboratorio se implementaron los algoritmos de *Dijkstra Link State*, *Flooding*, y *Distance Vector Routing* con el propósito de poder simular el proceso de envío de mensajes mediante routers.

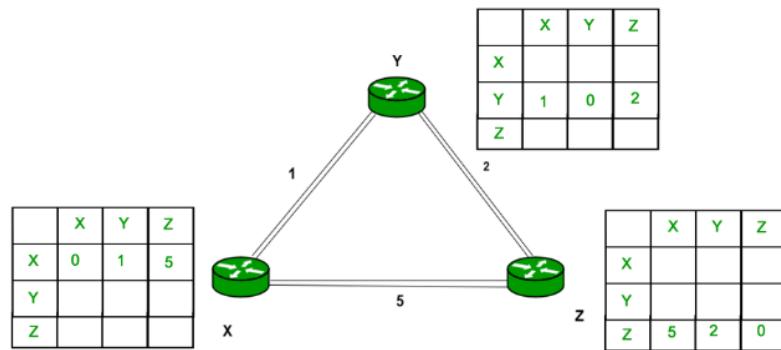
En este caso, los algoritmos de enrutamiento funcionan sobre nodos interconectados entre sí, donde cada nodo conoce únicamente cuáles son los vecinos que tiene. A través de *Dijkstra Link State* y *Distance Vector Routing*, cada nodo puede construir una tabla de enrutamiento que le permite determinar la mejor ruta para reenviar los paquetes hacia su destino. En el caso de Flooding, sin embargo, cada nodo envía los paquetes a sus vecinos (exceptuando al nodo vecino que envió el mensaje o cualquier nodo que ya recibió el mensaje anteriormente). Esto permite que los mensajes lleguen –eventualmente– a su destino.

Estos algoritmos son fundamentales en las redes de comunicación, ya que permiten que un mensaje llegue de forma eficiente a su destino. A continuación se describe más a profundidad cada algoritmo implementado.

*Distance Vector Routing*

Es un algoritmo de enrutamiento para determinar las rutas óptimas para transmitir datos entre dispositivos en una red. Cada nodo en la red mantiene una tabla de enrutamiento que contiene información sobre el costo y la ruta más corta hacia cada posible destino. Este algoritmo opera a través de iteraciones, donde en cada iteración, los nodos actualizan sus tablas de enrutamiento basándose en la información obtenida de sus vecinos. Cabe destacar que dichas actualizaciones se realizan utilizando la ecuación de Bellman-Ford. El proceso continúa hasta que todas las tablas de enrutamiento convergen y se encuentran las rutas óptimas (GeeksForGeeks, 2017).

**Figura 1.** Tabla de enrutamiento inicial de Distance Vector Routing



Aunque este algoritmo es considerado un enfoque sencillo de implementar y puede ser adecuado para redes pequeñas y poco complejas, tiene ciertas limitaciones (GeeksForGeeks, 2017):

1. Counting to infinity: Este problema ocurre cuando hay un enlace roto en la red. Consiste en que un router hace broadcast una ruta a través de otro router que, a su vez, hace broadcast la misma ruta de regreso al router original. Esto crea un ciclo de broadcast donde la distancia a través de la ruta rota sigue incrementándose gradualmente hasta llegar a un valor muy grande (como el infinito).
2. Lenta convergencia en redes grandes: En redes grandes, la convergencia en el enrutamiento por vector de distancia puede ser lenta debido a la forma en que funciona el algoritmo.

Para el presente laboratorio, en base a la teoría anteriormente descrita, se logró implementar el algoritmo de *Distance Vector Routing* en Python de la siguiente manera.

1. Se inicia en una terminal una instancia del programa. El programa pregunta qué nodo representa dicha instancia. Cabe destacar que cada nodo es representado por una diferente terminal.

2. Todos los nodos comparten entre sí sus vectores de distancia. Al recibir algún vector de un vecino, si este genera alguna actualización se comparte con todos los vecinos la tabla actualizada. Si no genera actualización, no se hace nada.
3. Se repite el paso 2 hasta que ya no haya más cambios y los vectores se estabilicen. A partir de aquí se puede empezar el proceso de enviar mensajes.
4. Si desean enviar un mensaje, primero preguntan a qué nodo se dirige el mensaje y cuál es el mensaje.
  - a. Utilizando los vectores y los saltos de enlace, se puede predecir a qué nodo se debe ir después para llegar al destino.
5. Cuando un nodo recibe un paquete, este revisa si es o no el destinatario final.
  - a. En caso de no ser el destinatario, determina a cuál nodo se debe enviar el paquete para que llegue de la forma más rápida al destinatario final.
  - b. De ser el destinatario final, muestra en la terminal el mensaje correspondiente.

#### Dijkstra Link State

Al igual que *Distance Vector Routing*, en *Dijkstra Link State* se busca determinar las rutas óptimas para transmitir mensajes y paquetes en una red de nodos interconectados. Inicialmente, cada nodo cuenta con una tabla que únicamente contiene información sobre el costo de los enlaces a los nodos vecinos. De forma iterativa, a través de un proceso de broadcast (*flooding*), el algoritmo *Dijkstra Link State* busca que los nodos conozcan todos los costos de los enlaces de la red. Para esto, los nodos comparten su información con sus vecinos. Cuando un nodo recibe la tabla de un vecino, actualiza su propia tabla con la nueva información y vuelve a compartir su tabla con sus vecinos. Este proceso de intercambio y actualización de tablas continúa hasta que todos los nodos en la red tengan conocimiento completo de los costos de enlace hacia todos los demás nodos (Javapoint, 2021).

Cuando todos los nodos conocen todos los costos de enlace de la red, cada uno puede calcular la ruta más corta hacia cualquier otro nodo en la red utilizando el algoritmo de Dijkstra (Javapoint, 2021).

**Figura 2.** Algoritmo de Dijkstra para encontrar camino más corto

```

Initialization
N = {A}      // A is a root node.
for all nodes v
  if v adjacent to A
    then D(v) = c(A,v)
    else D(v) = infinity
loop
  find w not in N such that D(w) is a minimum.
  Add w to N
  Update D(v) for all v adjacent to w and not in N:
  D(v) = min(D(v) , D(w) + c(w,v))
Until all nodes in N

```

Cabe destacar que *Dijkstra Link State* es un algoritmo eficiente para determinar rutas óptimas. Sin embargo, debido a su funcionamiento tiene varias limitaciones y desventajas (Haider, 2021):

1. Recursos: Dado a que cada nodo tiene una tabla de enrutamiento conformada por los costos de los enlaces de toda la red, cada uno requiere más memoria. Asimismo, ya que cada nodo es responsable de calcular el camino más corto a cada nodo, también es necesario que estos tengan suficiente capacidad de procesamiento para realizar estos cálculos.
2. Ineficiencia: Dado a que se comparte la información de costos de enlace repetidamente (*broadcast*), se puede generar una gran cantidad de tráfico en la red.

Para el presente laboratorio, en base a la teoría anteriormente descrita, se logró implementar el algoritmo de *Dijkstra Link State* en Python de la siguiente manera.

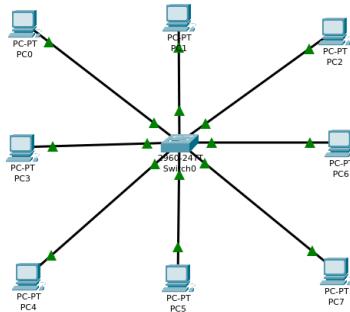
1. Se inicia en una terminal una instancia del programa. El programa pregunta qué nodo representa dicha instancia.
2. El nodo inicial hace un “echo” a los nodos vecinos (cada nodo es representado por una terminal previamente inicializada). Los nodos vecinos reciben el “echo” y le responden a la terminal inicial.
3. El nodo inicial posteriormente le comparte a sus nodos vecinos su tabla con costos de enlace. Esta tabla está vacía, exceptuando la fila que corresponde al nodo. Esta indica los costos de enlace de este nodo a sus vecinos.
4. Los vecinos reciben la tabla con costos de enlace en un formato JSON. Estos actualizan su propia tabla y posteriormente, cada uno, hace un “echo” a sus vecinos.
  - a. Estos nodos esperan un “echo” de regreso de sus vecinos.
5. Posteriormente, los vecinos envían su tabla actualizada a todos sus demás vecinos.
6. Se repite paso 4 y 5 hasta que todos los nodos tengan su tabla llena con los costos de enlace de toda la red. Cuando la tabla está llena, con el algoritmo de Dijkstra se genera la tabla de enrutamiento correspondiente en cada nodo.

7. Finalmente, con esto listo, los nodos son capaces de enviar mensajes. Si desean enviar un mensaje, primero preguntan a qué nodo se dirige el mensaje y cuál es el mensaje.
  - a. Utilizando la tabla de enrutamiento generada con el algoritmo de Dijkstra, los nodos determinan cuál es la ruta más corta para transmitir el mensaje.
8. Cuando un nodo recibe un paquete, este revisa si es o no el destinatario final.
  - a. En caso de no ser el destinatario, utiliza el resultado del algoritmo de Dijkstra para determinar a qué nodo enviar el paquete para que llegue de la forma más rápida al destinatario final.
  - b. De ser el destinatario final, muestra en la terminal el mensaje correspondiente.

### Flooding

Este es un algoritmo de enrutamiento que consiste en reenviar todos los paquetes entrantes a todos los vecinos; exceptuando al vecino del cual se recibió el paquete, y cualquier otro vecino que haya recibido el mensaje anteriormente. De esta manera, se asegura que el paquete llegue a su destino eventualmente, ya que se está enviando a todas las direcciones posibles (Hanna, 2021).

**Figura 3. Representación visual de Flooding**



Aunque el enrutamiento por inundación garantiza la entrega de paquetes, tiene varias limitaciones y desventajas. Debido a estas limitaciones, el enrutamiento por inundación se utiliza raramente en redes modernas y eficientes (TutorialsPoint, s.f.). A continuación se presentan algunas desventajas:

1. Ineficiencia: Dado que los paquetes se envían a todos los vecinos (exceptuando al vecino que envió el mensaje, y otros que ya hayan recibido el mismo), esto puede generar una gran cantidad de tráfico en la red.
2. Duplicación de paquetes: Debido a que se envían copias del paquete por todos los nodos, puede haber duplicación de paquetes en la red.
3. Falta de una topología: Este enfoque no tiene en cuenta la topología de la red ni la disponibilidad de rutas más eficientes.

4. Seguridad y privacidad: El enrutamiento por inundación puede exponer información a dispositivos no deseados.

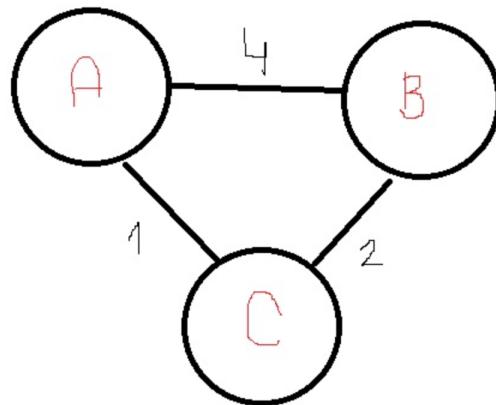
Para el presente laboratorio, en base a la teoría anteriormente descrita, se logró implementar el algoritmo de *Flooding* en Python de la siguiente manera.

1. Se inicia en una terminal una instancia del programa. El programa pregunta qué nodo representa dicha instancia.
2. Si el nodo quiere enviar un mensaje, se le pregunta el mensaje y el destinatario.
  - a. Posteriormente, se crea un paquete de tipo “info” al que se le agrega como *visitado* el nodo origen.
3. El programa posteriormente envía el paquete generado a sus nodos vecinos (representadas por instancias del programa) que no han sido anteriormente visitados.
4. Cuando estos reciben el paquete, revisan si son o no el destinatario final. Asimismo, se revisa si el mensaje ya había sido enviado o recibido por el nodo; en caso que sí, se descarta.
  - a. En caso de no ser el destinatario final, se agrega a la lista de *visitado*. Posteriormente, le envía el paquete a los nodos vecinos que no han sido visitados anteriormente.
  - b. De ser el destinatario final, muestra en la terminal el mensaje correspondiente.

#### 4. Resultados / Pruebas

En esta práctica se implementó diferentes algoritmos de enrutamiento, con el propósito de simular el proceso de envío de mensajes mediante routers de una red. Utilizando el código desarrollado para los algoritmos de *Dijkstra Link State*, *Flooding*, y *Distance Vector Routing*, se realizó una serie de pruebas con el propósito de evaluar el funcionamiento correcto de estos (incluyendo el proceso de transmisión de tablas de enrutamiento, cálculo de rutas más cortas, y el proceso de transmisión de mensajes). Para la realización de estas pruebas, se utilizó la siguiente red compuesta de tres nodos (*figura 4*). A continuación se presentan los resultados.

**Figura 4.** Diseño de red a utilizar en pruebas



#### Distance Vector Routing

Para el algoritmo de Distance Vector Routing, se inicializaron tres diferentes terminales representando cada uno de los nodos que se requerían para la topología. Como se puede observar, cada nodo le comparte a sus vecinos sus vectores de distancia, para que se empiece a usar la ecuación de Bellman Ford.

**Figura 5.** Inicialización de nodos

```
stefanoaragoni@Stefanos-MacBook-Pro lab3_rede$ python3 distanceVector.py
---DISTANCE VECTOR ROUTING---
Seleccione un nodo de la lista:
1. A
2. B
3. C
Ingrese el número del nodo: 1
Nodo seleccionado: A
Vecinos:
A: 0
B: 4
C: 1
{"type": "info", "headers": {"from": "A", "to": "B", "hop_count": 1}, "payload": "[0, 4, 1"]}

stefanoaragoni@Stefanos-MacBook-Pro lab3_rede$ python3 distanceVector.py
---DISTANCE VECTOR ROUTING---
Seleccione un nodo de la lista:
1. A
2. B
3. C
Ingrese el número del nodo: 2
Nodo seleccionado: B
Vecinos:
A: 4
B: 0
C: 2
{"type": "info", "headers": {"from": "B", "to": "A", "hop_count": 1}, "payload": "[4, 0, 1"]}

stefanoaragoni@Stefanos-MacBook-Pro lab3_rede$ python3 distanceVector.py
---DISTANCE VECTOR ROUTING---
Seleccione un nodo de la lista:
1. A
2. B
3. C
Ingrese el número del nodo: 3
Nodo seleccionado: C
Vecinos:
A: 1
B: 2
C: 0
{"type": "info", "headers": {"from": "C", "to": "A", "hop_count": 1}, "payload": "[1, 2, 0"]}
```

Al recibir los vectores de distancia, los nodos utilizan la ecuación de Bellman Ford para actualizar sus vectores. En caso haya una actualización, los nodos le comparten a sus vecinos las tablas actualizadas.

**Figura 6. Transmisión de Vectores de Distancia**

```

lab3_redes - Python distanceVector.py - 45x24
--> Vector de distancia recibido de C.
{"type": "info", "headers": {"from": "A", "to": "B", "hop_count": 1}, "payload": "[0, 3, 1"]}

lab3_redes - Python distanceVector.py - 45x24
--> Vector de distancia recibido de C.
{"type": "info", "headers": {"from": "B", "to": "A", "hop_count": 1}, "payload": "[3, 0, 2"]}

lab3_redes - Python distanceVector.py - 45x24
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingrese el número de la opción deseada: 1
Ingresu su paquete JSON: {"type": "info", "headers": {"from": "A", "to": "C", "hop_count": 1}, "payload": "[0, 3, 1"]}

---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingrese el número de la opción deseada: 1
Ingresu su paquete JSON: {"type": "info", "headers": {"from": "B", "to": "A", "hop_count": 1}, "payload": "[3, 0, 2"]"}

--> Vector de distancia recibido de B.

lab3_redes - Python distanceVector.py - 45x24
--> Vector de distancia recibido de A.
{"type": "info", "headers": {"from": "C", "to": "B", "hop_count": 1}, "payload": "[3, 0, 2"]}

---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 1
Ingresu su paquete JSON: {"type": "info", "headers": {"from": "B", "to": "C", "hop_count": 1}, "payload": "[0, 3, 1"]}

---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 1
Ingresu su paquete JSON: {"type": "info", "headers": {"from": "B", "to": "C", "hop_count": 1}, "payload": "[0, 3, 1"]"}

---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 1
Ingresu su paquete JSON: {"type": "info", "headers": {"from": "C", "to": "B", "hop_count": 1}, "payload": "[3, 0, 2"]"}

--> Vector de distancia recibido de B.

```

Después de haber convergido, se puede empezar a transmitir mensajes. Como se puede observar, del nodo A (izquierda) se quería enviar un mensaje a B. Sin embargo, se logró detectar que el camino más corto es a través de C, a pesar de que B es un vecino directo de A.

**Figura 7. Transmisión de Mensajes**

```

lab3_redes - Python distanceVector.py - 45x24
--> Vector de distancia recibido de B.
1}, "payload": "[0, 3, 1"]"

lab3_redes - Python distanceVector.py - 45x24
--> Vector de distancia recibido de A.
{"type": "info", "headers": {"from": "A", "to": "B", "hop_count": 3}, "payload": "[0, 3, 1"]}

lab3_redes - Python distanceVector.py - 45x24
---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 3
Ingresu su paquete JSON: {"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 3}, "payload": "hola!"}

---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 3
Ingresu su paquete JSON: {"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 3}, "payload": "hola!"}

Mensaje recibido. Enviando...
*Enviar el siguiente paquete a C...*
{"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 3}, "payload": "hola!"}

---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 3
Ingresu su paquete JSON: {"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 3}, "payload": "hola!"}

Mensaje de A: hola!
El mensaje ha llegado a su destino.

lab3_redes - Python distanceVector.py - 45x24
---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 3
Ingresu su paquete JSON: {"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 2}, "payload": "hola!"}

---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 3
Ingresu su paquete JSON: {"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 2}, "payload": "hola!"}

Mensaje recibido. Retransmitiendo...
*Enviar el siguiente paquete a B...*
{"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 2}, "payload": "hola!"}

---- DISTANCE VECTOR ROUTING ----
1) Recibir distance vector.
2) Enviar mensaje.
3) Recibir | retransmitir mensaje.
4) Salir
Ingresu el número de la opción deseada: 3
Ingresu su paquete JSON: {"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 2}, "payload": "hola!"}

```

### Dijkstra Link State

En el caso de *Dijkstra Link State*, como primer paso se inicializaron las terminales representativas de los nodos. Cada nodo conoce a sus vecinos así como los costos de enlace hacia los mismos.

**Figura 8. Inicialización de nodos**

The figure consists of three separate terminal windows, each running the script `lab3_redes - Python dijkstra.py`.  
Terminal 1 (Node A):  
Last login: Tue Aug 29 07:54:31 on ttys002  
stefanoaragoni@Stefanos-MacBook-Pro lab3\_redes % python3 dijkstra.py  
---DIJKSTRA---  
Seleccione un nodo de la lista:  
1. A  
2. B  
3. C  
Ingrese el número del nodo: 1  
Nodo seleccionado: A  
Vecinos:  
A: 0  
B: 4  
C: 1  
Terminal 2 (Node B):  
Last login: Tue Aug 29 07:54:58 on ttys003  
stefanoaragoni@Stefanos-MacBook-Pro lab3\_redes % python3 dijkstra.py  
---DIJKSTRA---  
Seleccione un nodo de la lista:  
1. A  
2. B  
3. C  
Ingrese el número del nodo: 2  
Nodo seleccionado: B  
Vecinos:  
A: 4  
B: 0  
C: 2  
Terminal 3 (Node C):  
Last login: Tue Aug 29 07:55:10 on ttys002  
stefanoaragoni@Stefanos-MacBook-Pro lab3\_redes % python3 dijkstra.py  
---DIJKSTRA---  
Seleccione un nodo de la lista:  
1. A  
2. B  
3. C  
Ingrese el número del nodo: 3  
Nodo seleccionado: C  
Vecinos:  
A: 1  
B: 2  
C: 0

Posteriormente, el nodo “principal” se encarga de mandarle un paquete tipo “echo” a sus vecinos. Esto con el propósito de verificar que estén activos y presentes. Finalmente, este nodo espera a que los vecinos le envíen un mensaje “echo” de vuelta. Como se puede observar a continuación, el nodo A (terminal a la izquierda) le envía un “echo” a sus dos vecinos y espera su respuesta.

**Figura 9. Echo a sus vecinos**

The figure consists of three separate terminal windows, each running the script `lab3_redes - Python dijkstra.py`.  
Terminal 1 (Node A):  
ECHO: {"type": "echo", "headers": {"from": "A", "to": "B", "hop\_count": 1}, "payload": "ping"}  
ECHO enviado exitosamente. Esperando respuesta...  
Ingrese su paquete JSON: {"type": "echo", "headers": {"from": "B", "to": "A", "hop\_count": 2}, "payload": "ping"}  
ECHO recibido exitosamente  
-----  
ECHO: {"type": "echo", "headers": {"from": "A", "to": "C", "hop\_count": 1}, "payload": "ping"}  
ECHO enviado exitosamente. Esperando respuesta...  
Ingrese su paquete JSON: {"type": "echo", "headers": {"from": "C", "to": "A", "hop\_count": 2}, "payload": "ping"}  
ECHO recibido exitosamente  
-----  
Terminal 2 (Node B):  
stefanoaragoni@Stefanos-MacBook-Pro lab3\_redes % python3 dijkstra.py  
---DIJKSTRA---  
Seleccione un nodo de la lista:  
1. A  
2. B  
3. C  
Ingrese el número del nodo: 2  
Nodo seleccionado: B  
Vecinos:  
A: 4  
B: 0  
C: 2  
Terminal 3 (Node C):  
stefanoaragoni@Stefanos-MacBook-Pro lab3\_redes % python3 dijkstra.py  
---DIJKSTRA---  
Seleccione un nodo de la lista:  
1. A  
2. B  
3. C  
Ingrese el número del nodo: 3  
Nodo seleccionado: C  
Vecinos:  
A: 1  
B: 2  
C: 0  
Ingresar su paquete JSON: []

Cuando se ha realizado el proceso de “echo” exitosamente, el nodo principal empieza a compartir su tabla a vecinos directos. Esta tabla está vacía, exceptuando la fila que corresponde al nodo. Esta indica los costos de enlace de este nodo a sus vecinos. Cuando los otros nodos reciben una tabla, estos actualizan su propia tabla y comparten la versión actualizada con sus vecinos. Esto se repite hasta que todos los nodos conozcan los costos de enlace de toda la red.

**Figura 10.** Broadcast de información de costos de enlace

```

Terminal 1 (A):
ECHO: {"type": "echo", "headers": {"from": "B", "to": "C", "hop_count": 2}, "payload": "ping"}
Ingresar su paquete JSON: {"type": "info", "headers": {"from": "A", "to": "C", "hop_count": 1}, "payload": "[[0, 4, 1], [9999, 9999, 9999], [1, 2, 0]]"}  

Tabla actualizada.  

-----  

TABLA ACTUAL:  

[[0, 4, 1], [4, 0, 2], [1, 2, 0]]  

Nodo listo para recibir y enviar mensajes.  

-----  

---- MENSAJES ----  

1) Enviar mensaje.  

2) Recibir | retransmitir mensaje.  

3) Salir  

Ingresar el número de la opción deseada: 1

Terminal 2 (B):
ECHO: {"type": "echo", "headers": {"from": "A", "to": "C", "hop_count": 2}, "payload": "ping"}
Ingresar su paquete JSON: {"type": "info", "headers": {"from": "C", "to": "B", "hop_count": 1}, "payload": "[[0, 4, 1], [9999, 9999, 9999], [1, 2, 0]]"}  

Tabla actualizada.  

-----  

TABLA ACTUAL:  

[[0, 4, 1], [4, 0, 2], [1, 2, 0]]  

Nodo listo para recibir y enviar mensajes.  

-----  

---- MENSAJES ----  

1) Enviar mensaje.  

2) Recibir | retransmitir mensaje.  

3) Salir  

Ingresar el número de la opción deseada: 1

Terminal 3 (C):
ECHO: {"type": "echo", "headers": {"from": "B", "to": "C", "hop_count": 1}, "payload": "ping"}
Ingresar su paquete JSON: {"type": "info", "headers": {"from": "B", "to": "C", "hop_count": 1}, "payload": "[[0, 4, 1], [4, 0, 2], [1, 2, 0]]"}  

Tabla actualizada.  

-----  

TABLA ACTUAL:  

[[0, 4, 1], [4, 0, 2], [1, 2, 0]]  

Nodo listo para recibir y enviar mensajes.  

-----  

---- MENSAJES ----  

1) Enviar mensaje.  

2) Recibir | retransmitir mensaje.  

3) Salir  

Ingresar el número de la opción deseada: 1

```

Cuando ya se conocen todos los costos de enlace de la red, los nodos permiten enviar y recibir mensajes. Cuando se desea enviar un mensaje, se solicita el mensaje y el nodo destinatario. Con esta información, el nodo utiliza el algoritmo de Dijkstra para determinar cuál es el camino más corto.

Como se puede observar a continuación, se quería enviar un mensaje del nodo A al nodo B. La ruta más corta (teóricamente) era A-C-B, a pesar de que B era un vecino directo de A. En este caso, la terminal del nodo A (izquierda) fue capaz de indicar la ruta más corta de forma correcta, e indicó que se transmitiera el paquete al nodo C. El nodo C recibió el paquete e identificó que el mensaje no era para él. Por tal razón, calculó nuevamente el camino más corto de sí mismo para el nodo destino (B). Finalmente, ya que enviar el mensaje directamente a B era el camino más corto, se lo envió.

**Figura 11.** Transmisión de mensajes y cálculo de camino más corto

```

Terminal 1 (A):
-----  

---- MENSAJES ----  

1) Enviar mensaje.  

2) Recibir | retransmitir mensaje.  

3) Salir  

Ingresar el número de la opción deseada: 1  

A qué nodo lequieres enviar el mensaje?: B  

Ingrésame tu mensaje: hola mundo  

Mensaje recibido. Envioando...  

Camino más corto: ['A', 'C', 'B']  

Enviar el siguiente paquete a C ...  

{"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 3}, "payload": "hola mundo"}  

-----  

---- MENSAJES ----

Terminal 2 (C):
-----  

---- MENSAJES ----  

1) Enviar mensaje.  

2) Recibir | retransmitir mensaje.  

3) Salir  

Ingresar el número de la opción deseada: 2  

Ingresar su paquete JSON: {"type": "message", "headers": {"from": "A", "to": "B", "hop_count": 3}, "payload": "hola mundo"}  

Mensaje de A: hola mundo  

El mensaje ha llegado a su destino.  

-----  

---- MENSAJES ----

Terminal 3 (B):
-----  

---- MENSAJES ----  

1) Enviar mensaje.  

2) Recibir | retransmitir mensaje.  

3) Salir  

Ingresar el número de la opción deseada: 2  

Ingresar su paquete JSON: {"type": "message", "headers": {"from": "C", "to": "B", "hop_count": 2}, "payload": "hola mundo"}  

Mensaje recibido. Retransmitiendo...  

Camino más corto: ['C', 'B']  

Enviar el siguiente paquete a B ...  

{"type": "message", "headers": {"from": "C", "to": "B", "hop_count": 2}, "payload": "hola mundo"}  

-----  

---- MENSAJES ----

```

Al haber evaluado la lógica del algoritmo, se quiso determinar y evaluar más a profundidad si el algoritmo era capaz de indicar las rutas más cortas de forma acertada. Para esto, se realizó una serie de experimentos, donde el programa debía de indicar la ruta más corta entre dos nodos. A continuación se presentan los resultados.

**Tabla 1.** Cálculos de caminos más cortos (Dijkstra)

Emisor - Receptor	Camino (teórico)	Camino (calculado)
A - B	A -> C -> B	Camino más corto: ['A', 'C', 'B'] Enviar el siguiente paquete a C ...
B - C	B -> C	Camino más corto: ['B', 'C'] Enviar el siguiente paquete a C ...
A - C	A -> C	Camino más corto: ['A', 'C'] Enviar el siguiente paquete a C ...
B - A	B -> C -> A	Camino más corto: ['B', 'C', 'A'] Enviar el siguiente paquete a C ...
C - B	C -> B	Camino más corto: ['C', 'B'] Enviar el siguiente paquete a B ...
C - A	C -> A	Camino más corto: ['C', 'A'] Enviar el siguiente paquete a A ...

Como se puede observar, el algoritmo de Dijkstra desarrollado fue capaz de calcular de forma correcta todas las rutas más cortas de un nodo a otro. Esto indica que este fue desarrollado de forma correcta. Asimismo, se logró demostrar el funcionamiento correcto y adecuado del proceso de transmisión de tablas, así como de mensajes, para el algoritmo de *Dijkstra Link State*.

## Flooding

A diferencia de los algoritmos anteriormente descritos, *Flooding* se caracteriza por no calcular el camino más corto de un nodo a otro. Sino, en este algoritmo los nodos envían cualquier paquete entrante a todos los nodos vecinos directos; exceptuando a cualquier vecino que haya recibido el mensaje anteriormente. De esta manera, se asegura que el paquete llegue a su destino eventualmente, ya que se está enviando a todas las direcciones posibles.

Por tal razón, como primer paso, se inicializaron las terminales representativas de los nodos. En este caso, nuevamente, cada nodo conoce a sus vecinos.

**Figura 12. Inicialización de nodos**

```
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
---FLOODING SIMULATION---
1. Node A
2. Node B
3. Node C
Ingrese el número del nodo: 1

----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 1

stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
---FLOODING SIMULATION---
1. Node A
2. Node B
3. Node C
Ingrese el número del nodo: 2

----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 1

stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
---FLOODING SIMULATION---
1. Node A
2. Node B
3. Node C
Ingrese el número del nodo: 3

----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 1
```

Debido al funcionamiento del algoritmo, este no requiere que se comparten las tablas de enrutamiento entre los nodos. Como resultado, directamente permite enviar mensajes a cualquier nodo, así como recibir mensajes. En la siguiente figura se muestra cómo enviar un mensaje del nodo A al nodo B.

Como primer paso, el nodo A pregunta a quién se le desea enviar el mensaje, así como el mensaje. Posteriormente, este indica que se le debe enviar el paquete generado a los nodos B y C (sus vecinos directos). Asimismo, se agrega el nodo A en la lista de nodos anteriormente visitados.

**Figura 13. Transmisión inicial de mensaje por Flooding**

```
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
---FLOODING SIMULATION---
1. Node A
2. Node B
3. Node C
Ingrese el número del nodo: 1

----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 1
A qué nodo lequieres enviar el mensaje?: B
Ingrese su mensaje: Hola!

*copiar y pegar en terminales: ['B', 'C']*
Enviando mensaje: {"type": "message", "headers": {"from": "A", "to": "B", "visited": ["A"]}, "payload": "Hola!"}

stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
---FLOODING SIMULATION---
1. Node A
2. Node B
3. Node C
Ingrese el número del nodo: 2

----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 2
Recibiendo mensaje: {"type": "message", "headers": {"from": "A", "to": "B", "visited": ["A"]}, "payload": "Hola!"}
Hola!

stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
---FLOODING SIMULATION---
1. Node A
2. Node B
3. Node C
Ingrese el número del nodo: 3

----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 3
Recibiendo mensaje: {"type": "message", "headers": {"from": "A", "to": "B", "visited": ["A"]}, "payload": "Hola!"}
Hola!
```

Al enviarle el paquete generado por A a los nodos vecinos (B y C), cada uno de estos muestra un mensaje diferente. Ya que B era el destinatario, este indica que recibió un mensaje y lo muestra en pantalla. Por otro lado, el nodo C reconoce que él no era el destinatario oficial. Por tal razón, C le reenvía el paquete a sus vecinos

(exceptuando a A, que se encuentra en la lista de nodos anteriormente visitados). Finalmente, se agrega a sí mismo a la lista de nodos visitados.

**Figura 14. Recepción de mensajes por Flooding**

```

lab3_redes - Python flooding.py - 45x24
2. Node B
3. Node C
Ingrese el número del nodo: 2
----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 2
Ingrese su paquete JSON: {"type": "message",
"headers": {"from": "A", "to": "B", "visited": ["A"]}, "payload": "Hola!"}
Mensaje recibido:
*retransmitir mensaje a ['B']*
Emisor: A
Mensaje: Hola!

lab3_redes - Python flooding.py - 45x24
----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 2
Ingrese su paquete JSON: {"type": "message",
"headers": {"from": "A", "to": "B", "visited": ["A"]}, "payload": "Hola!"}
Mensaje recibido:
Nuevo paquete JSON: {"type": "message",
"headers": {"from": "A", "to": "B", "visited": ["A", "C"]}, "payload": "Hola!"}

```

Finalmente, el nodo C envía a B el paquete con el mensaje. Sin embargo, el nodo B detecta que ya lo había recibido. Como resultado, se descarta el mensaje.

**Figura 15. Reenvío de mensaje por Flooding**

```

lab3_redes - Python flooding.py - 45x24
3) Salir
Ingrese el número de la opción deseada: 2
Ingrese su paquete JSON: {"type": "message",
"headers": {"from": "A", "to": "B", "visited": ["A"]}, "payload": "Hola!"}
Mensaje recibido:
Emisor: A
Mensaje: Hola!

----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 2
Ingrese su paquete JSON: {"type": "message",
"headers": {"from": "A", "to": "B", "visited": ["A"]}, "payload": "Hola!"}
Mensaje recibido:
*retransmitir mensaje a ['B']*
Nuevo paquete JSON: {"type": "message",
"headers": {"from": "A", "to": "B", "visited": ["A", "C"]}, "payload": "Hola!"}

lab3_redes - Python flooding.py - 45x24
----- FLOODING -----
1) Enviar mensaje.
2) Recibir mensaje.
3) Salir
Ingrese el número de la opción deseada: 2

```

Debido a que este algoritmo no calcula los caminos más cortos entre nodos, no fue posible realizar una serie de pruebas para verificar el cálculo correcto de los caminos más cortos. Sin embargo, sí se logró demostrar el funcionamiento correcto del algoritmo durante el proceso de transmisión de mensajes.

## 5. Discusión

En el presente laboratorio se implementaron tres algoritmos de enrutamiento: *Dijkstra Link State*, *Flooding*, y *Distance Vector Routing*. Los objetivos planteados se alcanzaron con éxito, permitiendo comprender el funcionamiento de los distintos algoritmos de enrutamiento utilizados en las implementaciones actuales de internet. Así mismo, a través del proceso de desarrollo se pudo determinar ciertas ventajas y desventajas de cada algoritmo.

Durante la presente práctica se pudo observar cómo el algoritmo de *Flooding* destacó por su sencillez a comparación de los otros algoritmos implementados. En este algoritmo, los nodos no comparten su información, ni calculan las rutas más cortas. Sino, en este caso cada nodo reenvía los paquetes de datos a todos sus vecinos (exceptuando a aquellos nodos que ya procesaron el paquete con anterioridad). Esto asegura que los paquetes lleguen –eventualmente– al destinatario. Sin embargo, con este algoritmo se pudo observar cómo este proceso (a pesar de ser más fácil) podía ser algo ineficiente.

Más específicamente, en una de las pruebas realizadas, se pudo observar como un mensaje *intentó* llegar dos veces a su destino. Esta duplicación de mensajes surge porque múltiples copias del mismo paquete pueden circular por la red simultáneamente, resultando en que intenten llegar al mismo destinatario más de una vez. Esto puede ocasionar problemas de congestión en la red, ya que innecesariamente se están transmitiendo paquetes que posiblemente ya llegaron a su destino. Sin embargo, cabe destacar que debido a la implementación realizada, un mismo mensaje no se imprime más de una vez en la terminal del destinatario.

En el caso de *Dijkstra Link State*, se pudo observar que este algoritmo constantemente calculaba de forma correcta el camino más corto a cualquier otro nodo. Esto era posible debido a que cada nodo tenía un mapa completo de la topología de la red. A diferencia del algoritmo de *Flooding*, este algoritmo de enrutamiento era capaz de enviar un mensaje directamente a su destino, así evitando esa duplicación de paquetes que anteriormente se explicó. Asimismo, resguardaba la privacidad del paquete ya que solo pasaba por donde era necesario.

Sin embargo, el algoritmo de *Dijkstra Link State* también presentó ciertas desventajas. Por ejemplo, se pudo observar que el proceso de llenar las tablas de información (de costos de enlace) era muy complicado y –relativamente– ineficiente. Esto debido a que cada nodo hacía un “echo” a sus vecinos para comprobar su presencia. Asimismo, se debía de esperar a que estos respondieran para proseguir. Posteriormente, los nodos compartían de manera repetitiva sus tablas entre sí mismos, hasta que todos los nodos tuvieran los costos de los enlaces de la red completa. Este proceso podría resultar muy tardado al momento de tener una red más grande, ya que la cantidad de información (posiblemente duplicada) transmitida aumenta significativamente.

Finalmente, cabe destacar que con el algoritmo de *Distance Vector Routing* se tuvo una situación similar a la de *Dijkstra Link State*. Esto debido a que

nuevamente se estaba compartiendo –de forma repetitiva– información entre los nodos. Sin embargo, en este caso se estaba usando la ecuación de Bellman Ford en cada intercambio para poder actualizar las rutas de los caminos más cortos. Este procedimiento se repitió hasta que ya no había más cambios en las tablas de caminos más cortos. Este intercambio de información “manual” (copiar y pegar la información entre terminales) nuevamente resultó ser un proceso sumamente lento. Sin embargo, se logró comprender que así es como funciona en topologías reales en redes con distintos dispositivos conectados.

En conclusión, en esta práctica se comprendió a profundidad los algoritmos de enrutamiento utilizados en las implementaciones actuales de internet. Asimismo, se logró identificar las restricciones, limitaciones y ventajas específicas de cada algoritmo. Esto siendo esencial para poder identificar qué algoritmo utilizar a la hora de diseñar una red de comunicaciones. Finalmente, cabe destacar que los tres algoritmos son valiosos en sus respectivos contextos y ofrecen diferentes enfoques para abordar la problemática de transmisión de mensajes.

## 6. Comentario

Esta práctica ha demostrado ser un valioso refuerzo para nuestra comprensión de los algoritmos de enrutamiento en redes de comunicación. A través de la implementación de los algoritmos *Distance Vector Routing*, *Dijkstra Link State* y *Flooding*, hemos adquirido un conocimiento profundo sobre cómo estos enfoques abordan el desafío de transmitir datos en una red. La simulación de los procesos de envío de mensajes y la evaluación de sus resultados nos ha permitido apreciar las ventajas y limitaciones de cada algoritmo.

Más específicamente, hemos experimentado cómo el algoritmo de *Flooding*, a pesar de su simplicidad, puede generar ineficiencias y redundancias en la transmisión de paquetes. Por otro lado, *Dijkstra Link State* ha demostrado su capacidad para calcular rutas óptimas, sin embargo, se detectó cierta ineficiencia que podría ralentizar el proceso de transmisión de mensajes en redes más grandes. Finalmente, *Distance Vector Routing*, también presentó buenos resultados para calcular rutas óptimas. Sin embargo, requirió de una comprensión profunda del algoritmo y de la ecuación de Bellman Ford para poder implementarlo.

Cabe destacar que lo que más nos costó en este laboratorio fue pensar cómo representar un nodo con cada instancia de los programas realizados. Asimismo, fue un reto desarrollar la lógica que permitió que los nodos se comunicaran entre sí mismos. Esto fue esencial para la ejecución del laboratorio, ya que se necesitaba que cada nodo conociera la información de los otros nodos para poder hacer uso de los algoritmos desarrollados (por lo menos para *Dijkstra Link State* y *Distance Vector Routing*). Finalmente, es importante mencionar que los algoritmos implementados fueron analizados en clase y en cursos anteriores, por lo cual su implementación fue relativamente sencilla.

En conclusión, este laboratorio nos ha brindado una comprensión profunda de los algoritmos de enrutamiento y cómo influyen en la eficiencia y confiabilidad de

las redes de comunicación. La capacidad de implementar, probar y comparar estos algoritmos en un entorno controlado nos ha proporcionado una valiosa experiencia que será relevante en el campo de las redes y la ingeniería de computación.

## Conclusiones

1. El objetivo principal de esta práctica fue logrado pues se logró comprender a detalle el funcionamiento de los algoritmos de enrutamiento de *Distance Vector Routing*, *Dijkstra Link State* y *Flooding*.
2. El algoritmo de *Flooding*, a pesar de su sencillez, puede resultar ineficiente debido a la duplicación de paquetes y a la posible congestión de red. Como se pudo observar en las pruebas, un mensaje intentó llegar a su destino más de una vez.
3. Los algoritmos de *Distance Vector Routing* y *Dijkstra Link State* presentaron resultados óptimos debido a que podían calcular rápidamente la ruta óptima para que un mensaje llegara a su destino. Sin embargo, el proceso de llenar las tablas de enrutamiento de forma repetitiva fue complejo. Cabe destacar que de incrementar el tamaño de la topología, se podría tardar mucho este proceso.
4. El mayor reto del laboratorio fue desarrollar la lógica para compartir información entre nodos previo al proceso de enviar mensajes. Esto fue la base para posteriormente poder encontrar las rutas más cortas entre los distintos nodos.

## 7. Referencias

1. Hanna, K. T. (2021). Flooding (network). Networking. <https://www.techtarget.com/searchnetworking/definition/flooding>
2. TutorialsPoint (s. f.). Flooding in computer network. <https://www.tutorialspoint.com/flooding-in-computer-network>
3. GeeksForGeeks. (2017). Distance Vector Routing DVR Protocol. GeeksforGeeks: <https://www.geeksforgeeks.org/distance-vector-routing-dvr-protocol/>
4. Javapoint. (2021). Computer Network | Link State Routing Algorithm - Javatpoint. <https://www.javatpoint.com/link-state-routing-algorithm>
5. Haider, A. (2021). Advantages and disadvantages of link-state routing. Retrieved August 28, 2023, from Blogspot.com website: <http://basicitnetworking.blogspot.com/2012/11/advantages-and-disadvantages-of-link.html>