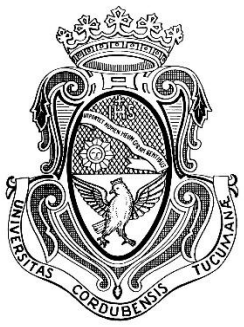


UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

CÁTEDRA DE ALGORITMOS Y ESTRUCTURA DE DATOS

TRABAJO PRÁCTICO FINAL

“Simulación tráfico de datos”

Alumnos:

Guimpelevich Maria Lujan	42.036.128	IComp
Venecia Milagros Ailin	43.610.298	IComp

2025

Introducción

Este proyecto presenta la simulación de tráfico de datos desarrollada en C++ utilizando algoritmos y estructuras de datos. La simulación modela una red compuesta por routers y terminales: las terminales generan páginas de datos (divididas en paquetes) y los routers se encargan de enrutar la información y gestionar colas, existe también un administrador que se encarga de calcular rutas óptimas según la congestión y el ancho de banda de las conexiones entre los routers.

Descripción de clases

Main

La función main() es el punto de entrada del programa y se encarga de configurar y simular el comportamiento de una red compuesta por routers y terminales. La simulación se estructura en los siguientes pasos:

1. Configuración Inicial de la Red

- **Definición de parámetros:**
  Se establecen dos variables:
  - cantidadRouters: cantidad total de routers que formarán la red (valor 5).
  - cantidadTerminales: cantidad de terminales por router (valor 1).
- **Impresión de la configuración:**
  Se muestra en consola la cantidad de routers y terminales definidos, para informar al usuario sobre la configuración inicial de la red.

2. Creación de Routers

- **Instanciación:**
  Se crea un vector<Router> para almacenar los routers.
  En un bucle for, se instancian los routers utilizando el constructor de la clase Router, pasando dos parámetros:
  - El identificador del router (un entero que varía de 0 a 4).
  - La cantidad de terminales que tendrá el router.
- **Método utilizado:**
  - Router(int id, int cantidadTerminales): Constructor que inicializa el router con un ID único y asigna la cantidad indicada de terminales.

3. Configuración de Vecinos entre Routers

- **Asignación de vecinos:**
  Se recorre el vector de routers y, mediante una estructura switch basada en el atributo idR (ID del router), se configura la topología de la red agregando vecinos a cada router.
  Para cada router se llama al método:
  Método utilizado:
  - agregarVecino(int vecinoID, int costo): Este método asigna un router vecino, donde:
    - vecinoID: identifica al router vecino.
    - costo: representa el costo o peso asociado a la conexión (por ejemplo, 3 o 2 según el caso).

- *Ejemplo:*
  Para el router con idR == 0, se agregan dos vecinos:
  - Router con ID 1 y costo 3.
  - Router con ID 4 y costo 2.

4. Envío de Páginas desde Terminales a Routers

- **Generación y envío de páginas:**
  Se itera sobre cada router y, para cada uno, se accede a su contenedor de terminales (presumiblemente una estructura asociativa, donde la clave identifica al terminal y el valor es un objeto de tipo Terminal).
  Por cada terminal se realizan dos operaciones:
  1. **Generar y enviar páginas:**
     Se invoca el método:
     - generarYEnviarPaginas(int cantidadRouters, int cantidadTerminales)
       Este método, perteneciente a la clase Terminal, se encarga de generar las páginas de datos y enviarlas. Los parámetros indican la cantidad total de routers y terminales, lo que puede influir en el contenido o en el proceso de envío.
  2. **Recepción de páginas en el router:**
     Luego, se llama al método:
     - recibirPagina(Terminal terminal)
       Este método, perteneciente a la clase Router, permite que el router reciba la página generada por el terminal.

5. Inicio del Tráfico y Administración de la Red

- **Creación del Administrador:**
  Finalmente, se instancia un objeto de la clase Administrador pasando el vector de routers.
  Se asume que el constructor de Administrador se encarga de iniciar y coordinar el tráfico de la red, gestionando la interacción entre los routers y posiblemente controlando el flujo de datos.
  Método utilizado:
  - Administrador(vector<Router> routers): Constructor que recibe la lista de routers y se encarga de administrar la red.

6. Finalización del Programa

- Se muestra un mensaje indicando la finalización de la red y se retorna 0 para terminar la ejecución del programa.

Paquete

La clase Paquete se utiliza para representar un paquete de datos que se transfiere en el sistema. Cada objeto de esta clase encapsula información relevante sobre el paquete, incluyendo su identificador, los puntos de origen y destino, el contenido del mensaje y un identificador de página asociado.

Atributos Principales

- **id (int):**
  Identificador único del paquete.
- **origen (pair<int, int>):**
  Par que representa el router y el terminal de origen.
  - `origen.first`: ID del router de origen.
  - `origen.second`: ID del terminal de origen.
- **destino (pair<int, int>):**
  Par que representa el router y el terminal de destino.
  - `destino.first`: ID del router de destino.
  - `destino.second`: ID del terminal de destino.
- **contenido (string):**
  Cadena que contiene el mensaje o los datos del paquete.
- **paginaId (int):**
  Identificador de la página a la que pertenece el paquete.

Página

La clase Pagina se encarga de gestionar y almacenar un conjunto de paquetes que serán enviados o procesados en el sistema. Cada objeto de esta clase tiene un identificador único y contiene una lista de paquetes, donde cada paquete se crea con un identificador, información de origen y destino, un contenido representativo y el ID de la página a la que pertenece.

El constructor de **Página** automatiza la creación de un conjunto de paquetes de datos al momento de instanciar una nueva página, permitiendo establecer de manera centralizada tanto la cantidad de paquetes como la información de origen y destino.

Terminal

La clase **Terminal** representa un punto final dentro de la red, encargado de generar y enviar "páginas" (que a su vez contienen paquetes de información) hacia otros routers, y de recibir páginas que han sido reconstruidas. Entre sus responsabilidades se encuentran:

- **Identificación de la Terminal:**
  Cada terminal se identifica mediante un número único que se forma concatenando dos enteros (por ejemplo, una IP simplificada). Esta identificación se usa para diferenciar las terminales dentro de la red.
- **Gestión de la Comunicación:**
  La terminal puede generar páginas de datos con paquetes, enviarlas a los routers y, además, recibir páginas completas desde la red.

- **Registro de Actividad:**
  La clase utiliza colas para almacenar tanto las páginas enviadas como las páginas recibidas, y un mapa para almacenar el tamaño (cantidad de paquetes) de cada página generada.

Métodos de la Clase Terminal

1. Constructor: `Terminal::Terminal(pair<int,int> ip)`

Propósito:
Inicializar una nueva instancia de la clase **Terminal** a partir de un par de enteros que representan su "IP".

Funcionamiento:

- **Generación del ID:**
  - Se convierte cada parte del par (`ip.first` y `ip.second`) a cadena y se concatenan para formar un identificador único (`concatenado_id_terminal`).
  - Luego, se convierte esa cadena a un entero (`idCompleto`), que representa de forma numérica el ID completo de la terminal.
- **Asignación de la IP y Router Conectado:**
  - Aunque se observa que se realiza la asignación `ip = ip;`, la intención es que la terminal almacene la IP recibida (se recomienda revisar posibles problemas de *shadowing* de variables).
  - Se asigna el primer valor del par (`ip.first`) a la variable `routerConectado`, lo que indica el identificador del router al que está conectada la terminal.

2. Método: `void Terminal::recibirPagina(const Pagina& pagina)`

Propósito:
Recibir una página reconstruida y almacenarla en la cola de páginas recibidas.

Funcionamiento:

- **Almacenamiento:**
  Se utiliza el método `push` de la cola `paginasRecibidas` para agregar la página recibida.
- **Notificación:**
  Se imprime un mensaje en consola indicando que la terminal (identificada por su `concatenado_id_terminal`) ha recibido la página completa identificada por `pagina.id`.

3. Método: `void Terminal::generarYEnviarPaginas(int cantidadRouters, int cantidadTerminales)`

Propósito:
Generar páginas de datos de forma aleatoria y simular su envío a los routers de la red.

Funcionamiento:

- **Inicialización de Generadores Aleatorios:**
  Se emplean:
  - `random_device` y `mt19937` para la semilla y generación de números aleatorios.

    - Distribuciones uniformes para determinar:
        - **Cantidad de páginas a enviar:** Un número aleatorio entre 1 y 2.
        - **Cantidad de paquetes por página:** Un número aleatorio entre 2 y 3.
        - **Destino del router:** Un número aleatorio entre 0 y `cantidadRouters - 1`.
        - **Destino del terminal:** Un número aleatorio entre 0 y `cantidadTerminales - 1`.
- **Generación de Páginas:**
    - Para cada página a enviar (iteración controlada por `cantidadPaginas`):
        - **Selección de Destino:**
          Se elige un router destino de forma aleatoria, asegurándose de que no sea el mismo que el `router Conectado` (es decir, la terminal no se envía a sí misma). Luego se selecciona el terminal destino dentro del rango permitido.
        - **Creación de la Página:**
          Se crea un objeto `Página` utilizando:
            - Un identificador (el índice `i` de la iteración).
            - La cantidad de paquetes generada aleatoriamente.
            - La IP (o identificación) de la terminal origen.
            - Un par que indica el destino (router y terminal).
        - **Registro y Notificación:**
            - Se imprime en consola un mensaje informando que la terminal ha generado la página con el número de paquetes y el destino específico.
            - Se agrega la página a la cola `páginas Enviadas` para llevar un registro de las páginas generadas.
            - Se almacena el tamaño (cantidad de paquetes) de la página en el mapa `tamanioPaginas`, utilizando el `id` de la página como clave.

Router

La clase **Router** se encarga de modelar un nodo en la red que gestiona terminales, administra conexiones con otros routers (vecinos) y se ocupa de procesar páginas y paquetes de información. Esta clase interactúa estrechamente con las clases **Terminal**, **Paquete** y **Página**, permitiendo la comunicación y transmisión de datos entre diferentes puntos de la red.

Constructor: `Router::Router(int id_router, int cantidad_terminales)`

Propósito:
Inicializa un objeto de tipo Router asignándole un identificador único y conectándole una cantidad específica de terminales.

Acciones realizadas:

- **Asignación del ID:**
  Se asigna el valor de `id_router` al atributo `idR` del objeto, identificando de forma única al router.

- **Visualización de información:**
  Se imprime en consola un mensaje indicando el ID del router y la cantidad de terminales que se conectarán.
- **Creación de terminales:**
  Mediante un bucle `for` (iterando desde 0 hasta `cantidad_terminales - 1`), se realiza lo siguiente para cada terminal:
  - Se crea un par `{idR, i}` donde `i` es el índice del terminal.
  - Se genera un identificador único para el terminal concatenando el ID del router y el índice del terminal, luego se convierte a entero y se asigna a `id_terminal_completo`.
  - Se instancia un objeto **Terminal** utilizando este identificador.
  - Se inserta la terminal en el contenedor `terminales` (posiblemente un mapa), utilizando como clave el índice del terminal (`id_terminal.second`).
  - Se muestra un mensaje indicando que el terminal ha sido conectado al router.

Método: `void Router::agregarVecino(int vecinoR, int anchoDeBanda)`

Propósito:
Configurar la conexión entre el router actual y otro router (vecino), especificando el ancho de banda de dicha conexión.

Acciones realizadas:

- **Registro del vecino:**
  Se inserta en el contenedor `vecinos` el par formado por el ID del vecino (`vecinoR`) y el ancho de banda (`anchoDeBanda`).
- **Inicialización de estructuras de tráfico:**
  - Se crea una cola de paquetes vacía para el vecino en el contenedor `colasPorVecino`.
  - Se inicializa la carga (número de paquetes en cola) para el vecino en el contenedor `cargaPorVecino` con el valor 0.
- **Visualización de información:**
  Se imprime un mensaje indicando que el vecino ha sido agregado, junto con el ancho de banda asignado.

Método: `void Router::actualizarCarga()`

Propósito:
Actualizar la "carga" o cantidad de paquetes pendientes para cada vecino, de acuerdo al tamaño de la cola de paquetes correspondiente.

Acciones realizadas:

- **Iteración sobre vecinos:**
  Se recorre el contenedor `vecinos` y, para cada vecino, se obtiene el tamaño de su cola en `colasPorVecino`.

- **Actualización de la carga:**
  Se asigna en `cargaPorVecino` el tamaño (cantidad de paquetes) de la cola asociada a cada vecino.

Este método es útil para conocer la congestión de cada enlace y posiblemente tomar decisiones de enrutamiento basadas en la carga.

Método: `void Router::recibirPagina(Terminal& terminal)`

Propósito:
Recibir una página enviada por una terminal, dividirla en paquetes y colocar estos en la cola intercalada del router para su posterior procesamiento o reenvío.

Acciones realizadas:

- **Procesamiento de páginas enviadas:**
  Se utiliza un bucle `while` para procesar todas las páginas presentes en la cola `paginasEnviadas` del objeto **Terminal**:
  - Se extrae la primera página de la cola y se elimina de ella.
  - Se muestra un mensaje indicando que el router está recibiendo la página, mencionando el ID de la página y el identificador de la terminal.
- **División en paquetes:**
  Para cada página extraída, se itera sobre el vector de paquetes (`pagina.paquetes`) que la conforman.
  - Cada paquete se inserta en la cola intercalada (`colaIntercalada`) del router.
  - Se imprime un mensaje indicando que el paquete, junto con el ID de la página a la que pertenece, ha sido agregado a la cola intercalada.

Método: `void Router::recibirPaquete(const Paquete& paquete, Terminal& terminal)`

Propósito:
Procesar un paquete recibido de otro router, almacenarlo y verificar si se ha recibido la totalidad de los paquetes que conforman una página. De ser así, se procede a reconstruir la página.

Acciones realizadas:

- **Visualización de información:**
  Se imprime un mensaje indicando que el router está recibiendo un paquete, mostrando su ID, el ID de la página a la que pertenece y el router de origen.
- **Almacenamiento del paquete:**
  El paquete se almacena en el contenedor `paquetesPorPagina` bajo la clave correspondiente al ID de la página (`paquete.getPaginaId()`). Se agrega al vector de paquetes asociados a esa página.

- **Verificación de completitud:**
  Se compara la cantidad de paquetes almacenados para la página con el tamaño original de la misma, obtenido de `terminal.tamanioPaginas[paquete.getPaginaId()]`.
    - Si se han recibido todos los paquetes, se invoca el método `reconstruirPagina` para ensamblar la página.

Método: `void Router::reconstruirPagina(int paginaId, Terminal& terminal)`

Propósito:
Reconstruir una página a partir de los paquetes recibidos, ordenarlos y ensamblarlos para formar la página completa que luego se envía a la terminal correspondiente.

Acciones realizadas:

- **Mensaje de reconstrucción:**
  Se imprime un mensaje indicando que el router ha reconstruido la página identificada por `paginaId`.
- **Ordenamiento de paquetes:**
  Se obtiene el vector de paquetes correspondiente a la página a reconstruir.
  Se utiliza el algoritmo de burbuja para ordenar los paquetes en función de su ID, garantizando que queden en el orden correcto.
- **Creación y envío de la página:**
    - Se crea un objeto **Pagina** utilizando:
        - El `paginaId`.
        - El tamaño de la página (obtenido de `terminal.tamanioPaginas[paginaId]`).
        - El origen y destino, extraídos del primer paquete del vector.
    - Se asigna al objeto **Pagina** el vector ordenado de paquetes.
    - Se llama al método `recibirPagina` de la terminal para enviar la página reconstruida.
- **Limpieza de datos:**
  Se elimina la entrada correspondiente a `paginaId` en el contenedor `paquetesPorPagina`, liberando memoria y evitando duplicados.

Administrador

La clase **Administrador** se encarga de gestionar y coordinar el flujo de paquetes dentro de una red compuesta por múltiples routers. Su responsabilidad principal es procesar el envío y la recepción de paquetes, así como recalcular las rutas de los routers en función de la carga actual de la red. Para ello, la clase invoca diversos métodos de los objetos **Router** y actúa como el "cerebro" que asegura que los paquetes sean enrutados correctamente.

1. Constructor: `Administrador::Administrador(vector<Router>& routers)`

- **Propósito:**
  Inicializar una instancia de **Administrador** y comenzar inmediatamente el proceso de envío/recepción de paquetes a través de la red.
- **Funcionamiento:**
  - Recibe por referencia un vector de objetos **Router**.
  - Inmediatamente llama al método `procesarPaquetes(routers)` para iniciar la simulación del tráfico en la red.
- **Observaciones:**
  La llamada al método `procesarPaquetes` dentro del constructor implica que la administración de la red se inicia de forma automática al crear el objeto **Administrador**.

2. Método: `void Administrador::recalcularRutas(vector<Router>& routers)`

- **Propósito:**
  Recalcular las rutas de los routers para optimizar el enrutamiento de paquetes basándose en la carga actual de cada router y sus conexiones con vecinos.
- **Funcionamiento:**
  - Se imprime un mensaje indicando el inicio del recalculo de rutas.
  - Para cada **Router** en el vector:
    - Se actualiza la carga del router mediante el método `actualizarCarga()`, que probablemente mide la cantidad de paquetes en cola o el tráfico actual.
    - Se recorre la tabla de enrutamiento del router. Para cada destino (identificado por un índice en la tabla), se invoca el método `establecerRuta(router, destino)`, el cual determina la mejor ruta para enviar paquetes a ese destino en función de la carga relativa de los vecinos.
- **Observaciones:**
  Este método permite que la red se adapte dinámicamente a cambios en la congestión, mejorando el rendimiento del enrutamiento de paquetes.

3. Método: `void Administrador::procesarPaquetes(vector<Router>& routers)`

- **Propósito:**
  Coordinar el procesamiento y enrutamiento de paquetes en la red. Este método simula el tráfico en la red, moviendo los paquetes desde los routers de origen hacia sus destinos finales.
- **Funcionamiento:**
  - **Bucle Principal (Ciclos):**
    Se utiliza un ciclo `while` que se ejecuta mientras existan paquetes pendientes en las colas de los routers.
    - **Inicio de Ciclo:**
      Se imprime el número de ciclo actual y un mensaje de que se están procesando paquetes.
    - **Procesamiento por Router:**
      Para cada router en el vector:

- Se muestra un mensaje indicando el inicio del proceso de paquetes para ese router.
- **Procesamiento de Cola Intercalada:**
  Mientras la cola intercalada del router no esté vacía:
  - Se extrae el paquete del frente de la cola.
  - Si el paquete tiene como destino el mismo router (comparando `paquete.getDestinoRouter()` con `router.idR`):
    - Se llama al método `recibirPaquete` del router, pasando el paquete y el terminal de destino correspondiente (extraído de `router.terminales`).
    - Se imprime un mensaje indicando que el paquete ha llegado a su destino para su reconstrucción.
  - En caso contrario (el paquete debe ser reenviado a otro router):
    - Se actualiza la ruta al destino mediante una llamada a `establecerRuta(router, paquete.getDestinoRouter())`.
    - Se determina el vecino de destino usando la tabla de enrutamiento del router.
    - El paquete se coloca en la cola correspondiente del vecino (dentro de `router.colasPorVecino`).
    - Se imprime un mensaje detallando el cálculo de ruta para ese paquete.
- **Envió a Vecinos:**
  Luego, se recorre cada cola de vecinos (`colasPorVecino`) del router. Para cada par (ID de vecino y cola de paquetes):
  - Se identifica el router correspondiente al vecino en el vector de routers.
  - Mientras la cola de ese vecino tenga paquetes, se transfieren a la cola intercalada del router vecino, imprimiendo un mensaje que indica el envío del paquete.
- **Recalculo Periódico de Rutas:**
  Cada 2 ciclos, se llama al método `recalcularRutas(routers)` para actualizar las rutas óptimas basándose en la nueva carga de la red.
- **Verificación de Paquetes Pendientes:**
  Se verifica si quedan paquetes en alguna de las colas intercaladas o en las colas por vecino. Si al menos uno de los routers tiene paquetes pendientes, el ciclo continúa; de lo contrario, se termina el procesamiento.

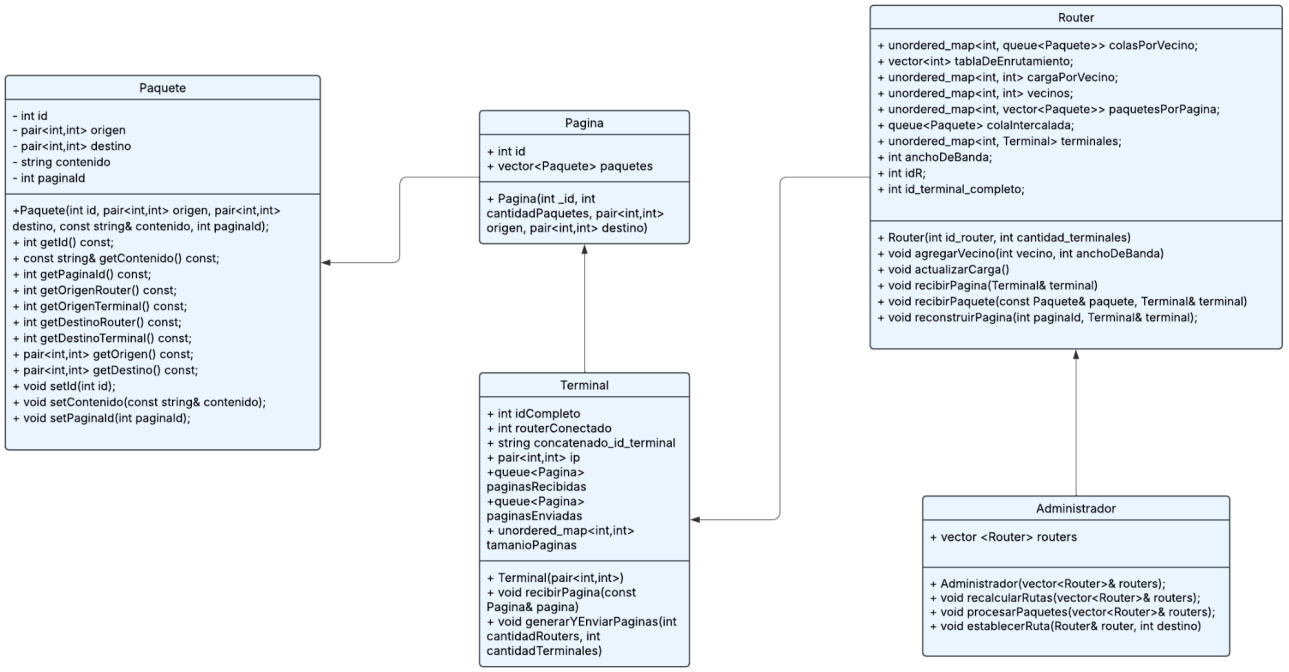
- **Finalización del Proceso:**
  Una vez que ya no quedan paquetes en ninguna de las colas, se imprime un mensaje indicando que la simulación ha sido completada.

- **Observaciones:**
  Este método simula el flujo de paquetes en la red y es fundamental para probar la eficiencia de los algoritmos de enrutamiento y la gestión de congestión. La estructura de ciclos y verificaciones asegura que se procese toda la información antes de finalizar la simulación.

4. Método: `void Administrador::establecerRuta(Router& router, int destino)`

- **Propósito:**
  Determinar la mejor ruta para enviar un paquete desde un router hacia un destino específico, basándose en la carga y el ancho de banda disponible en las conexiones con los routers vecinos.
- **Funcionamiento:**
  - Se inicia verificando que el router tenga vecinos configurados. Si no los tiene, no se puede establecer una ruta.
  - Se actualiza la carga del router (llamando a `actualizarCarga()`) para obtener datos actuales de congestión.
  - Se inicializan variables para determinar la "mejor opción":
    - `mejorOpcion` se inicializa en -1.
    - `menorCargaRelativa` se inicializa con un valor muy alto para asegurarse de que cualquier carga calculada sea menor.
  - Se itera sobre cada vecino del router, accediendo a sus respectivos anchos de banda:
    - Se verifica que el vecino exista en la estructura `colasPorVecino`.
    - Se obtiene el tamaño (cantidad de paquetes en cola) y se calcula la carga relativa como el cociente entre la cantidad de paquetes y el ancho de banda. Si el ancho de banda es 0, se asigna un valor máximo para evitar división por cero.
    - Si la carga relativa calculada es menor o igual que la menor encontrada hasta el momento, se actualiza la variable `mejorOpcion` con el ID del vecino y se guarda la nueva carga relativa mínima.
  - Tras evaluar todos los vecinos:
    - Si se encontró una mejor opción (es decir, `mejorOpcion` es distinto de -1), se actualiza la tabla de enrutamiento del router:
      - Se verifica y, de ser necesario, se redimensiona la tabla de enrutamiento para asegurar que tenga capacidad para almacenar la ruta al destino.
      - Se asigna en la posición correspondiente de la tabla el ID del vecino seleccionado.
    - Si no se encontró una ruta válida, se imprime un mensaje de error.
  - Finalmente, se imprime un mensaje indicando cuál es la mejor opción (el vecino) para alcanzar el router destino.
- **Observaciones:**
  Este método es crucial para el algoritmo de enrutamiento, ya que permite adaptar dinámicamente las rutas en función de la congestión y el ancho de banda, asegurando que los paquetes se envíen por el camino más eficiente posible.

Diagrama de clases


Paquete
- int id
- pair<int,int> origen
- pair<int,int> destino
- string contenido
- int paginaId
+Paquete(int id, pair<int,int> origen, pair<int,int> destino, const string& contenido, int paginaId);
+ int getId() const;
+ const string& getContenido() const;
+ int getPaginaId() const;
+ int getOrigenRouter() const;
+ int getOrigenTerminal() const;
+ int getDestinoRouter() const;
+ int getDestinoTerminal() const;
+ pair<int,int> getOrigen() const;
+ pair<int,int> getDestino() const;
+ void setId(int id);
+ void setContenido(const string& contenido);
+ void setPaginaId(int paginaId);
Pagina
+ int id
+ vector<Paquete> paquetes
+ Pagina(int _id, int cantidadPaquetes, pair<int,int> origen, pair<int,int> destino)
Router
+ unordered_map<int, queue<Paquete>> colasPorVecino;
+ vector<int> tablaDeEnrutamiento;
+ unordered_map<int, int> cargaPorVecino;
+ unordered_map<int, int> vecinos;
+ unordered_map<int, vector<Paquete>> paquetesPorPagina;
+ queue<Paquete> colaIntercalada;
+ unordered_map<int, Terminal> terminales;
+ int anchoDeBanda;
+ int idR;
+ int id_terminal_completo;
+ Router(int id_router, int cantidad_terminales)
+ void agregarVecino(int vecino, int anchoDeBanda)
+ void actualizarCarga()
+ void recibirPagina(Terminal& terminal)
+ void recibirPaquete(const Paquete& paquete, Terminal& terminal)
+ void reconstruirPagina(int paginaId, Terminal& terminal);
Terminal
+ int idCompleto
+ int routerConectado
+ string concatenado_id_terminal
+ pair<int,int> ip
+queue<Pagina> paginasRecibidas
+queue<Pagina> paginasEnviadas
+ unordered_map<int,int> tamanioPaginas
+ Terminal(pair<int,int>)
+ void recibirPagina(const Pagina& pagina)
+ void generarYEnviarPaginas(int cantidadRouters, int cantidadTerminales)
Administrador
+ vector <Router> routers
+ Administrador(vector<Router>& routers);
+ void recalcularRutas(vector<Router>& routers);
+ void procesarPaquetes(vector<Router>& routers);
+ void establecerRuta(Router& router, int destino)