UCUENCA

UNIVERSIDAD DE CUENCA

FACULTAD DE INGENIERÍA

COMPUTACIÓN

Trabajo Final - Algoritmos de Redes de Computadores

Autor:

Mendoza Bryan

Asignatura: Redes de Computadoras

Docente: Ing. Raúl Ortiz

Fecha de entrega: 28/07/2025

Árbol generador PRIM

PI1.1 – Identificación del problema

El algoritmo de Prim resuelve el problema de encontrar un árbol de expansión mínima en un grafo no dirigido y ponderado, es decir, un árbol que recorra todos los nodos con el menor costo posible y sin generar bucles. Esto es un subproblema común en redes de computadores para el diseño eficiente de topologías de red. Su objetivo es conectar todos los routers (nodos) con el costo total mínimo, evitando ciclos.

Requisitos:

- El grafo debe ser no dirigido.
- Cada arista debe tener un peso.

Limitaciones:

- No funciona correctamente si el grafo no es conexo.
- Se toma un nodo arbitrario como punto de inicio, lo cual no afecta el resultado, pero puede cambiar el orden de las aristas seleccionadas.

PI1.2 – Descomposición del problema

Para implementar el algoritmo, se utilizaron las siguientes técnicas y estructuras:

- **Estructuras de datos:**
  - Set para llevar registro de nodos visitados.
  - Array para almacenar aristas candidatas y el resultado final (camino).
- **Técnica Greedy (voraz):** En cada iteración se selecciona la arista de peso mínimo que conecta un nodo visitado con uno no visitado.
- **Modularización:** El código se divide en dos funciones principales:
  - prim(graph): lógica central del algoritmo, el cual recorre todos los nodos del grafo hasta haber visitado todos.
  - agregarAristas(): identifica y agrega aristas candidatas desde el nodo actual.

PI1.3 – Evaluación y factores relevantes

Soluciones alternativas:

- **Algoritmo de Kruskal:** También construye un árbol de expansión mínima, pero ordenando todas las aristas desde el inicio y usando estructuras de conjuntos disjuntos.

Se eligió Prim porque es más eficiente para grafos con muchas aristas. A diferencia de Kruskal, Prim parte desde un nodo y expande hacia el más cercano no visitado, lo que lo hace más rápido en este tipo de grafos. Además, su lógica se adapta fácilmente a simulaciones de redes, como en el modelado de redes troncales óptimas. El algoritmo fue trabajado durante el curso, lo que facilita su implementación.
Otra ventaja es que no requiere analizar todas las aristas; su estructura progresiva permite mantener el código claro, modular y con buena legibilidad.

Algoritmo Dijkstra

PI1.1 – Identificación del problema

El algoritmo de Dijkstra resuelve el problema de encontrar la ruta más corta desde un nodo origen a todos los demás nodos en un grafo ponderado. Este algoritmo es fundamental en redes de computadores para el enrutamiento eficiente de paquetes, minimizando el costo o la distancia total del recorrido entre routers.

Requisitos:

- El grafo debe tener pesos no negativos en las aristas.
- Debe indicarse un nodo origen existente.

Limitaciones:

- En caso de ser una red extremadamente grande, la eficiencia del algoritmo puede verse comprometida si no se implementa con una estructura adecuada como una cola de prioridad basada en un heap, lo cual optimiza el tiempo de acceso al nodo con menor coste.
- El grafo debe ser conexo para obtener resultados útiles.

PI1.2 – Descomposición del problema

El problema fue descompuesto y estructurado en varias funciones para una implementación modular:

- **Estructuras utilizadas:**
  - distancias: objeto que almacena la distancia más corta conocida desde el origen a cada nodo.
  - anteriores: objeto que almacena el nodo anterior en el camino más corto.
  - heap: array que actúa como una cola de prioridad para elegir el siguiente nodo con la distancia mínima.
  - iteraciones: registro del estado del algoritmo en cada paso, útil para posteriormente mostrar la tabla correspondiente.
- **Funciones:**
  - reconstruirCamino(): permite obtener la ruta desde el nodo origen a un destino.
  - imprimirTablaDijkstra(): genera una tabla paso a paso con distancias y predecesores.
- **Técnica utilizada:**
  - El enfoque es voraz (greedy), ya que en cada iteración se expande el nodo más cercano aún no visitado.
  - Se simula una cola de prioridad ordenando el heap manualmente con sort().

PI1.3 – Evaluación y factores relevantes

Soluciones alternativas:

- **Bellman-Ford**: permite pesos negativos, pero es menos eficiente.
- **A***: útil si se quiere encontrar una ruta óptima entre dos nodos con heurística.

Se eligió el algoritmo Dijkstra por su eficiencia para encontrar el camino más corto en grafos y su amplia adopción en sistemas de enrutamiento como OSPF (Open Shortest Path First).
Además muestra paso a paso cada iteración, facilitando la comprensión detallada del algoritmo.

Vector - Distancia

PI1.1 – Identificación del problema

El algoritmo de Vector Distancia busca resolver el problema de encontrar las rutas más cortas entre todos los nodos de una red, cuando cada nodo sólo conoce a sus vecinos directos y la información es intercambiada entre nodos de forma iterativa, de manera que en cada iteración se va propagando la información con los nodos que no son directamente vecinos.
Este modelo es fundamental para protocolos como RIP (Routing Information Protocol), donde los nodos intercambian periódicamente sus tablas de enrutamiento.

Requisitos:

- Cada nodo debe tener conocimiento inicial solo de sus vecinos y el costo a ellos.
- Grafo representado como lista de adyacencia.

Limitaciones:

- Requiere múltiples iteraciones para converger a la solución.
- Los pesos deben ser no negativos.
- Puede converger lentamente si hay cambios en la red (caída de ruteadores o enlaces).

PI1.2 – Descomposición del problema

El problema fue descompuesto y estructurado en varias funciones para una implementación modular:

- **Inicialización:** Cada nodo construye su tabla de distancias inicial en donde se incluye únicamente las distancias a sí mismo y con sus vecinos directos.
- **Intercambio iterativo:** En cada iteración, los nodos actualizan sus tablas con la información recibida de sus vecinos. El proceso continúa hasta que no hay más cambios en las tablas.
- **Funciones clave:**
    - vectorDistancia(graph): lógica central del algoritmo donde inicializa las tablas y posteriormente va recorriendo los nodos para actualizarlas en cada iteración.
    - generarTablasNodo(): construye dos tablas por nodo en cada iteración, la Tabla 1 muestra las distancias que reporta cada vecino a cada destino y la Tabla 2 muestra la mejor ruta elegida (costo y desde donde se conecta).
- **Técnica utilizada:**
    - Este algoritmo es distribuido y cooperativo, ya que cada nodo decide localmente pero influenciado por los demás nodos.
    - Utiliza el principio de Bellman-Ford, pero aplicado de forma distribuida.

PI1.3 – Evaluación y factores relevantes

Soluciones alternativas:

- **Estado del Enlace:** Cada nodo identifica sus vecinos directos y difunde esta información a toda la red. A partir de estos datos, cada nodo reconstruye la topología completa.
- **A*** (con heurística), útil para rutas dirigidas entre pares concretos.

Se eligió el enfoque de Vector Distancia por su capacidad para simular el funcionamiento de protocolos como RIP, en los que la información se transmite de manera parcial y distribuida entre nodos. Su

implementación permite observar cómo evoluciona la red y cómo se construyen las tablas de enrutamiento desde datos locales. Además ofrece una visualización iterativa clara y refleja el comportamiento cooperativo real entre nodos. Sin embargo, su eficiencia se ve comprometida en redes grandes o dinámicas, ya que requiere múltiples iteraciones.

Estado del enlace

PI1.1 – Identificación del problema

El algoritmo de Estado del Enlace resuelve el problema de construcción de la topología completa de una red a partir de la información local de cada nodo. Esto es esencial para protocolos como OSPF (Open Shortest Path First), donde cada nodo recopila y difunde información sobre sus conexiones (enlaces), para luego calcular las rutas más cortas.
En resumen, su objetivo es construir una representación global y consistente de la red desde las percepciones individuales de cada nodo.

Requisitos:

- Cada nodo debe tener su lista de enlaces con sus vecinos (destino y peso).
- No repetir un enlace con pesos diferentes.

Limitaciones:

- La función presentada no calcula las rutas, solo reconstruye el grafo a partir de las tablas locales de cada nodo.
- La posición de los nodos es aleatoria para la visualización, no representa su ubicación lógica en la red.
- No contempla fallos de enlaces.

PI1.2 – Descomposición del problema

La implementación se centra en construir la topología de red compartida usando los datos locales de cada nodo (tabla de enlaces por nodo).

- **Estructuras utilizadas:**
    - Set: para evitar nodos o aristas duplicadas.
    - nodes: arreglo de nodos con propiedades para renderizado visual.
    - edges: arreglo de enlaces con etiquetas de peso.
- **Lógica:**
    - Recorre cada nodo y su lista de enlaces (adyacencias).
    - Normaliza las aristas (A-B y B-A se consideran iguales).
    - Agrega nodos y enlaces al conjunto global si aún no están.

PI1.3 – Evaluación y factores relevantes

Soluciones alternativas:

- **Vector Distancia:** los nodos solo conocen a sus vecinos y aprenden rutas mediante intercambio.

El modelo de Estado del Enlace es aplicado en protocolos como OSPF, y permite estructurar la topología de la red a partir de la reconstrucción de caminos, basándose en las tablas de cada nodo, las cuales contienen los enlaces con sus vecinos correspondiente, logrando una convergencia más eficiente en comparación con el enfoque Vector Distancia y sin bucles de enrutamiento. Su precisión facilita la simulación de cambios y la representación gráfica completa. Sin embargo, en redes muy dinámicas requiere sincronización constante y un mayor uso de memoria.

Código de Hamming

PI1.1 – Identificación del problema

El Código de Hamming es un algoritmo de detección y corrección de errores ampliamente utilizado en el envío de paquetes, telecomunicaciones y almacenamiento digital. Su función principal es detectar y corregir un error de 1 bit en una trama de datos binarios sin necesidad de retransmisión.

Objetivo del algoritmo:

- Determinar el número de bits de paridad requeridos.
- Codificar la trama original agregando los bits de paridad.
- Detectar si un bit fue alterado durante la transmisión.
- Corregir el error y recuperar la trama original.

Limitaciones:

- Solo corrige un error por trama.
- No detecta ni corrige errores múltiples.
- No verifica errores en los bits de paridad.

PI1.2 – Descomposición del problema

El algoritmo está dividido en funciones independientes que representan fases concretas:

- calcularBitsRedundantes(m): Calcula cuántos bits de paridad se necesitan según el número de bits de datos.
- generarMultiplos2(): Determina las posiciones de los bits de paridad (potencias de 2).
- llenarPrimeraFila(): Inserta los bits de datos en la matriz, dejando libres las posiciones de paridad.
- llenarFilas(): Llena las filas de la matriz con base en el valor binario de las posiciones, para definir qué bits afectan a cada bit de paridad.
- contarUnosPar(): Verifica si el número de unos en una fila es par.
- completarBitsParidad(): Establece los valores de los bits de paridad en función de la paridad calculada.
- subirBitsParidad(): Inserta los bits de paridad en sus posiciones correspondientes en la trama final.
- danarTrama(): Simula un error modificando un bit específico de la trama.
- detectarError(): Compara los bits de paridad del emisor y el receptor para encontrar la posición del bit erróneo.

- corregirTrama(): Corrige la trama invirtiendo el bit dañado.
- obtenerDato(): Extrae los bits de datos, eliminando los bits de paridad.
- crearMatriz(): Genera una matriz vacía que sirve de base para el cálculo de paridad.

El flujo completo se gestiona en la función principal codigoHamming(), que realiza:

1. Codificación de la trama.
2. Simulación del daño (opcional).
3. Reconstrucción de la trama en el receptor.
4. Detección y corrección del error.
5. Recuperación del dato original.

PI1.3 – Evaluación y factores relevantes

Soluciones alternativas:

- **CRC (Cyclic Redundancy Check):** detecta errores múltiples pero no los corrige.
- **Reenvío automático (ARQ):** depende de confirmaciones y retransmisiones.

El Código de Hamming es una solución ligera, eficiente y fácil de implementar, ideal para sistemas embebidos donde se requiere transmisión confiable. Su capacidad de corrección directa sin comunicación adicional lo hace práctico, aunque limitada a errores de un solo bit y no contempla fallos en bits de paridad sin extensiones. La implementación modular facilita la depuración, la visualización y futuras mejoras, integrando de forma clara la simulación de errores, el cálculo de bits redundantes y la recuperación de la trama original.

Cyclic Redundancy Check (CRC)

PI1.1 - Identificación del problema

La transmisión de datos digitales está sujeta a errores causados por ruido o interferencias en el canal de comunicación. Para garantizar la integridad de los datos recibidos, es necesario un mecanismo eficiente de detección de errores. CRC permite detectar errores en la transmisión de un mensaje binario sin incurrir en una sobrecarga significativa.

Requisitos:

- Mensaje y generador en binario.
- La longitud del mensaje debe ser mayor a la del generador.

Limitaciones:

- El algoritmo no corrige errores, solo detecta si ocurren.
- La capacidad de detección depende del polinomio generador elegido.
- El proceso asume que la trama puede ser representada como un arreglo binario.

PI1.2 - Descomposición del problema

El problema se puede dividir en las siguientes etapas bien definidas:

- **Extensión del mensaje:** Agrega al mensaje original una cantidad de bits en cero igual al grado del polinomio generador menos uno, para preparar la división binaria.
- **División binaria mediante XOR:** Realizar la división polinómica entre el mensaje extendido y el polinomio generador, usando operaciones XOR bit a bit, que simulan la división sin acarreo en aritmética binaria.
- **Cálculo del residuo (CRC):** El residuo obtenido tras la división es la secuencia de bits de control, llamada CRC, que se adjunta al mensaje original para formar la trama.
- **Verificación en el receptor:** El receptor vuelve a realizar la división entre la trama recibida (mensaje + CRC) y el generador. Si el residuo es cero, se confirma que no hay errores; de lo contrario, se detecta una corrupción.
- **Registro de pasos:** Se almacenan los pasos intermedios de la división para facilitar la comprensión y depuración del proceso.

Esta división permite estructurar la solución de manera modular y clara, facilitando su implementación y pruebas.

PI1.3 - Evaluación y factores relevantes

Soluciones alternativas:

- Uso de otros métodos de detección/corrección de errores, como códigos de Hamming (corrección de errores simples) o checksum.

CRC (Cyclic Redundancy Check) es un método muy usado para detectar errores en transmisiones digitales, ya que ofrece mayor precisión que técnicas más simples como el checksum. Mientras que el checksum puede pasar por alto ciertos errores, CRC tiene una capacidad mucho mejor para detectar errores comunes, incluyendo cambios en múltiples bits.
Esta implementación se basa en divisiones usando la operación XOR, lo cual es eficiente y fácil de programar. Se utilizan arreglos binarios para representar los datos, facilitando su manipulación y comprensión. Además, guardar los pasos intermedios permite revisar con detalle cómo se genera y verifica el código.

Bibliografía

Algoritmo de Kruskal. (n.d.). Retrieved July 27, 2025, from https://www.programiz.com/dsa/kruskal-algorithm

Bellman–Ford Algorithm - GeeksforGeeks. (n.d.). Retrieved July 27, 2025, from https://www.geeksforgeeks.org/dsa/bellman-ford-algorithm-dp-23/

El Algoritmo A: Guía completa | DataCamp*. (n.d.). Retrieved July 27, 2025, from https://www.datacamp.com/es/tutorial/a-star-algorithm