PUCE Virtual

DOCUMENTO DE PROFUNDIZACIÓN

Estructura de Datos

Estudio de caso 2: Identificación de rutas críticas mediante Dijkstra

Estudio de caso 2: Identificación de rutas críticas mediante Dijkstra

Contexto

La empresa "SecureGrid Ops" gestiona una red de centros de datos interconectados por fibra óptica en cinco ciudades. Cada enlace (arista) entre ciudades tiene un "costo" asociado que representa la latencia (en milisegundos) y la tasa de error del canal. En un entorno de alta disponibilidad, la empresa necesita asegurar que los datos de control de procesos industriales viajen siempre por la ruta más rápida y estable.

Problema

Debido a un aumento en los ciberataques de denegación de servicio (DDoS) distribuidos, ciertos nodos de la red están experimentando congestión, lo que eleva la latencia de forma impredecible. El sistema de enrutamiento estático actual no es capaz de recalcular rutas óptimas en tiempo real, lo que provoca que los paquetes de seguridad lleguen con retraso, dejando las subestaciones vulnerables.

Solución Propuesta

Implementar un sistema de Identificación de Rutas Críticas basado en el Algoritmo de Dijkstra. Este sistema monitoreará constantemente los costos de los enlaces y reconfigurará las tablas de enrutamiento para garantizar que el tráfico crítico siempre tome el camino de menor "peso" (latencia acumulada).

Estrategia:

- **Modelado del Grafo:** Representar las ciudades como nodos y las conexiones como aristas con pesos.
- **Mantenimiento de Estados:** Utilizar una **cola de prioridad** (Priority Queue) para extraer siempre el nodo con la distancia mínima conocida.
- **Relajación de Aristas:** Actualizar las distancias de los vecinos si se encuentra un camino más corto a través del nodo actual.
- **Reconstrucción:** Almacenar los predecesores para trazar la ruta exacta.

Implementación

Utilizamos el módulo heapq para asegurar una complejidad eficiente de $O((E + V)LogV)$

```python
import heapq

def dijkstra(grafo, inicio, destino):
    # distancias[nodo] = [distancia_minima, nodo_previo]
    distancias = {nodo: float('inf') for nodo in grafo}
    distancias[inicio] = 0
    predecesores = {nodo: None for nodo in grafo}
```

```python
# Cola de prioridad: (distancia, nodo)
prioridad = [(0, inicio)]

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

    if nodo_actual == destino:
        break

    if distancia_actual > distancias[nodo_actual]:
        continue

    for vecino, peso in grafo[nodo_actual].items():
        camino = distancia_actual + peso

        if camino < distancias[vecino]:
            distancias[vecino] = camino
            predecesores[vecino] = nodo_actual
            heapq.heappush(prioridad, (camino, vecino))
```

```python
    # Reconstrucción de la ruta
    ruta = []
    actual = destino
    while actual:
        ruta.insert(0, actual)
        actual = predecesores[actual]

    return ruta, distancias[destino]

# Grafo de SecureGrid (Latencia en ms)
red_infraestructura = {
    'Sede_Central': {'Nodo_A': 5, 'Nodo_B': 2},
    'Nodo_A': {'Sede_Central': 5, 'Nodo_C': 4, 'Nodo_D': 2},
    'Nodo_B': {'Sede_Central': 2, 'Nodo_A': 8, 'Nodo_D': 7},
    'Nodo_C': {'Nodo_A': 4, 'Destino_Final': 3},
    'Nodo_D': {'Nodo_A': 2, 'Nodo_B': 7, 'Destino_Final': 1},
    'Destino_Final': {'Nodo_C': 3, 'Nodo_D': 1}
}

ruta_optima, latencia_total = dijkstra(red_infraestructura, 'Sede_Central', 'Destino_Final')
print(f"Ruta Crítica: {ruta_optima} | Latencia Total: {latencia_total}ms")
```

Resultados:

El algoritmo identificó que, aunque el camino directo por el Nodo B parece corto, la ruta Sede_Central -> Nodo_A -> Nodo_D -> Destino_Final ofrece una latencia menor (ms vs otras opciones de ms).

Impacto en Ciberseguridad:

- **Mitigación de DDoS:** Al detectar un aumento de "peso" (latencia) en un nodo bajo ataque, Dijkstra desvía automáticamente el tráfico por rutas limpias.
- **Integridad de Datos:** Reduce la probabilidad de pérdida de paquetes por congestión, manteniendo los túneles VPN estables.
- **Segmentación Dinámica:** Permite aislar nodos comprometidos simplemente asignándoles un peso "infinito" en el grafo.

Reflexión Técnica

Dijkstra es un algoritmo voraz (greedy). Su mayor fortaleza es la precisión en grafos con pesos positivos, pero su debilidad es que no puede manejar pesos negativos (donde se requeriría Bellman-Ford). En ciberseguridad, la clave no es solo encontrar el camino más corto, sino actualizar los pesos de las aristas en función de métricas de seguridad en tiempo real para que el algoritmo tome decisiones informadas sobre amenazas.