PUCE Virtual

CLASE

6

ESTRUCTURA DE DATOS (PROGRAMACIÓN III)

Operaciones en grafos y algoritmos avanzados

QS WORLD UNIVERSITY RANKINGS

Educación de calidad

INTRODUCCIÓN DE LA CLASE

GRADO / POSGRADO / TECNOLOGÍAS

Estudia a tu tiempo
son interrupciones

EDUCACIÓN EN LÍNEA DE CALIDAD

En esta semana se profundiza en el estudio de los grafos mediante el análisis de sus operaciones fundamentales y la aplicación de algoritmos avanzados. Los grafos no solo permiten representar relaciones entre elementos dentro de un sistema, sino que también facilitan el estudio de su comportamiento a través de recorridos y cálculos de rutas. En este contexto, se abordarán algoritmos esenciales como la búsqueda en profundidad (DFS) y la búsqueda en anchura (BFS), los cuales permiten explorar la estructura de una red y comprender cómo se conectan sus diferentes componentes.

Además, se introducirán algoritmos avanzados como el de Dijkstra, orientados a la identificación de caminos mínimos en grafos ponderados. Estos conceptos tienen una aplicación directa en el ámbito de la ciberseguridad, donde permiten analizar rutas de ataque, estudiar la propagación de amenazas y detectar puntos críticos dentro de una infraestructura tecnológica. A través de estos conocimientos, los estudiantes podrán desarrollar habilidades para modelar, analizar y optimizar redes, fortaleciendo así la capacidad de diseñar estrategias de seguridad más eficientes y basadas en el análisis estructural de los sistemas.

Clase 6: Seleccionar las estructuras de datos más adecuadas según los requerimientos y características de distintos escenarios de ciberseguridad.

6. Operaciones en grafos y algoritmos avanzados

Los grafos no solo permiten representar relaciones entre elementos dentro de un sistema, sino también realizar diferentes operaciones que facilitan el análisis de redes complejas. Estas operaciones incluyen recorridos dentro del grafo, búsqueda de rutas entre nodos, identificación de caminos óptimos y análisis de estructuras críticas dentro de una red.

Estas técnicas son fundamentales para analizar el comportamiento de redes informáticas, detectar vulnerabilidades y comprender cómo podría propagarse un ataque dentro de una infraestructura tecnológica. Los algoritmos de grafos permiten estudiar rutas de comunicación, identificar nodos críticos y determinar caminos mínimos entre sistemas conectados.

6.1. Recorridos en grafos

Los recorridos en grafos son algoritmos diseñados para visitar sistemáticamente todos los nodos o vértices de un grafo siguiendo determinadas reglas de exploración. Su objetivo principal es analizar la estructura del grafo y comprender las relaciones existentes entre los diferentes elementos que lo conforman. Estos algoritmos son esenciales dentro del estudio de la teoría de grafos y tienen aplicaciones en diversas áreas de la informática, como la inteligencia artificial, el análisis de redes y la optimización de rutas.

Los recorridos permiten modelar y analizar la forma en que un atacante podría desplazarse dentro de una infraestructura tecnológica. Mediante estos métodos es posible identificar rutas potenciales hacia sistemas críticos, detectar puntos vulnerables dentro de la red o estudiar la propagación de malware entre dispositivos interconectados. Los dos métodos de recorrido más utilizados son la Búsqueda en Profundidad (DFS) y la Búsqueda en Anchura (BFS), los cuales exploran los nodos del grafo siguiendo estrategias distintas pero complementarias. (Cormen et al., 2016).

6.1.1. Búsqueda en profundidad (DFS)

Algoritmo de recorrido de grafos que explora los nodos siguiendo cada rama del grafo hasta llegar lo más lejos posible antes de retroceder. Prioriza la exploración profunda de los caminos disponibles, lo que permite analizar detalladamente cada posible ruta dentro de la estructura del grafo. DFS permite recorrer sistemáticamente todos los vértices de un grafo explorando cada camino antes de considerar alternativas. Se inicia en la raíz y continúa avanzando hacia uno de sus nodos vecinos que aún no haya sido visitado. Este proceso se repite sucesivamente hasta que se alcanza un nodo desde el cual ya no es posible continuar avanzando. En ese momento, el algoritmo retrocede al nodo anterior para explorar otras ramas que aún no hayan sido recorridas. La Figura 1 muestra el paso a paso del funcionamiento del algoritmo de búsqueda en profundidad.

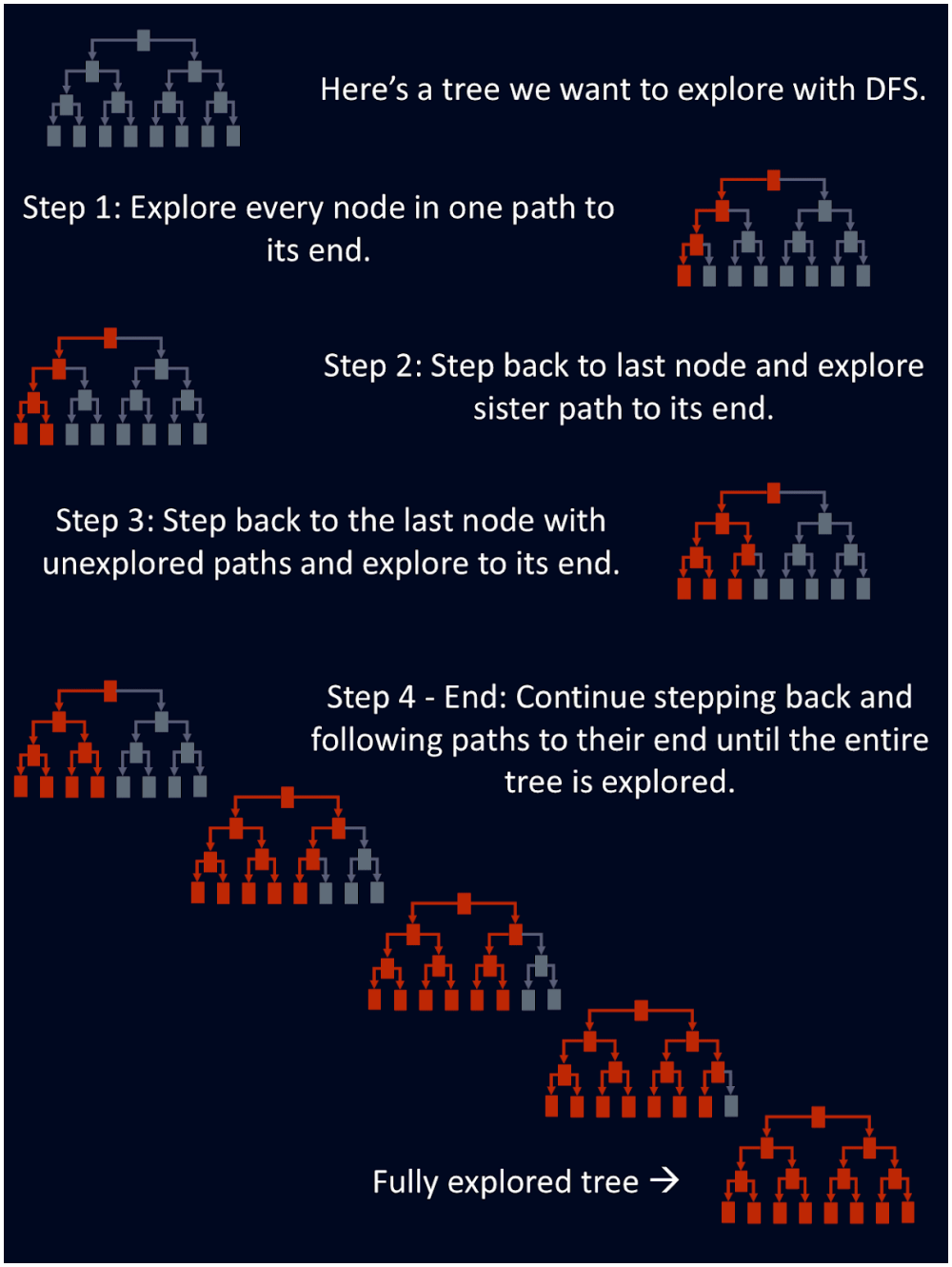


Figura 1. Explicación paso a paso de la búsqueda en profundidad. Tomado de: DataCamp. https://www.datacamp.com/es/tutorial/depth-first-search-in-python

Este tipo de recorrido es útil para analizar rutas completas dentro de una red, detectar ciclos y explorar estructuras jerárquicas. DFS puede emplearse para simular cómo un atacante podría desplazarse dentro de una red informática comprometiendo diferentes sistemas de manera progresiva, lo que permite identificar posibles caminos de intrusión y puntos vulnerables en la infraestructura tecnológica (Cormen et al., 2016).

Ejemplo en Python:

```
grafo = {
  "Internet": ["Firewall"],
```

```
  "Firewall": ["ServidorWeb"],
  "ServidorWeb": ["BaseDatos"],
  "BaseDatos": []
}

def dfs(nodo, visitados):
  if nodo not in visitados:
    print(nodo)
    visitados.add(nodo)
    for vecino in grafo[nodo]:
      dfs(vecino, visitados)

dfs("Internet", set())
```

Búsqueda en profundidad en Python: Recorrer grafos y árboles, de DataCamp. Página Web tutorial en español donde explica conceptualmente y con ejemplos y código en Python sobre la búsqueda en profundidad, además incluye enlaces a otros recursos complementarios. Enlace: https://www.datacamp.com/es/tutorial/depth-first-search-in-python

6.1.2. Búsqueda en anchura (BFS)

Algoritmo de recorrido de grafos que explora los nodos de manera progresiva por niveles. A diferencia de otros métodos que profundizan en una sola ruta, BFS visita primero todos los nodos vecinos del nodo inicial antes de avanzar hacia niveles más profundos del grafo. Este enfoque permite analizar la estructura de la red de forma ordenada, comenzando desde el nodo de origen y expandiéndose gradualmente hacia los demás nodos conectados.

Funciona utilizando una cola para almacenar los nodos que están pendientes de ser explorados. Inicialmente se coloca el nodo de partida en la cola y, a medida que se visitan los nodos, sus vecinos no visitados se agregan a la cola para ser procesados posteriormente. Esto garantiza que los nodos se recorran en orden de proximidad al nodo inicial. Útil para analizar grafos cuando se requiere identificar caminos mínimos o explorar estructuras organizadas por niveles. (Cormen et al., 2016)

En la figura 2 se presenta un ejemplo de búsqueda en anchura junto con su cola.

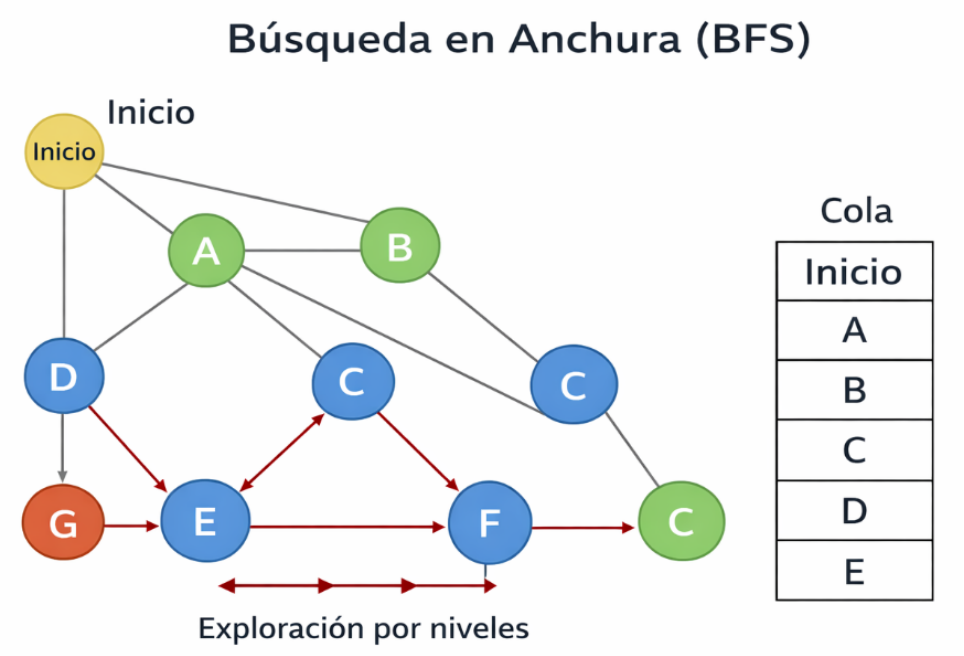


Figura 2. Ejemplo de Búsqueda en Anchura. Creación de autor Patricio Coba

Entre sus principales aplicaciones están calcular la distancia mínima entre nodos, analizar redes por niveles y estudiar la propagación de información o malware en sistemas interconectados. BFS puede emplearse para modelar cómo un ataque informático o un malware podría propagarse dentro de una red desde un punto inicial, permitiendo identificar rápidamente qué sistemas podrían verse afectados primero y cuáles serían los siguientes en la cadena de propagación (Cormen et al., 2016).

Ejemplo en Python:

```
from collections import deque

def bfs(grafo, nodo_inicio):
    visitados = set()
    cola = deque([nodo_inicio])

    visitados.add(nodo_inicio)

    while cola:
        nodo_actual = cola.popleft()
        print(f"Visitando nodo: {nodo_actual}")

        for vecino in grafo[nodo_actual]:
            if vecino not in visitados:
                visitados.add(vecino)
                cola.append(vecino)

mi_grafo = {
```

```
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

bfs(mi_grafo, 'A')
```

6.1.3. Comparación DFS vs BFS

DFS y BFS permiten recorrer un grafo visitando todos sus nodos, pero difieren en la forma en que exploran la estructura del grafo y en las aplicaciones para las que resultan más adecuados. Estos algoritmos representan dos estrategias distintas para explorar grafos: una que profundiza en cada camino antes de retroceder y otra que avanza nivel por nivel. (Bhargava, 2017)

DFS explora el grafo siguiendo una rama hasta llegar al punto más profundo posible antes de regresar para explorar otras rutas. Generalmente se implementa mediante recursión o utilizando una pila, lo que la hace útil para analizar rutas complejas, detectar ciclos o explorar completamente todas las posibles conexiones entre nodos. BFS recorre el grafo por niveles, visitando primero todos los nodos vecinos antes de avanzar a niveles más profundos, este algoritmo utiliza una cola para gestionar los nodos pendientes de explorar, lo que permite encontrar de manera eficiente la distancia mínima entre nodos en un grafo no ponderado. La Tabla 1 muestra la comparación entre los algoritmos DFS y BFS.

Tabla 1. Comparación entre algoritmos DFS y BFS. Creación de autor Patricio Coba

Característica	DFS	BFS
Tipo de exploración	Profundidad	Por niveles
Estructura utilizada	Pila o recursión	Cola
Uso común	Exploración completa	Distancia mínima
Aplicaciones	Análisis de rutas complejas	Propagación de información

En redes, BFS suele emplearse para encontrar rutas más cortas entre nodos, por ejemplo, en protocolos de enrutamiento o análisis de conectividad. DFS es más útil cuando se requiere explorar completamente todas las rutas posibles dentro de un sistema, lo que puede ayudar a comprender la estructura completa de una red o detectar posibles caminos de acceso dentro de un análisis de seguridad informática (Bhargava, 2017).

6.1.4. Uso de recorridos en análisis de propagación de ataques

Los algoritmos de recorrido permiten analizar cómo un ataque podría propagarse dentro de una infraestructura tecnológica. Por ejemplo, si un atacante compromete una computadora dentro de una red, podría intentar acceder a otros sistemas conectados.

La Figura 3 muestra un ejemplo de cómo los grafos permiten representar relaciones entre componentes de una red.

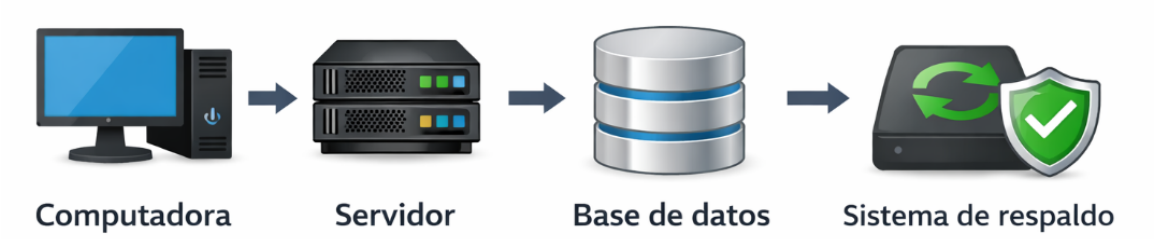


Figura 3. Representación de relaciones de grafos. Creación de autor Patricio Coba

Mediante algoritmos de recorrido es posible:

- Simular ataques
- Analizar rutas de propagación de malware
- Identificar puntos críticos de acceso
- Diseñar estrategias de defensa

Estas técnicas también se utilizan en herramientas de análisis de seguridad para evaluar la superficie de ataque de una organización.

6.1.5. Recursividad implícita en recorridos

Muchos algoritmos de recorrido de grafos utilizan recursividad, especialmente en el caso de DFS. La recursividad es una técnica de programación en la que una función se llama a sí misma para resolver un problema dividiéndolo en subproblemas más pequeños. En grafos, este enfoque permite que el algoritmo visite un nodo y, posteriormente, llame nuevamente a la misma función para explorar cada uno de sus nodos vecinos que aún no han sido visitados. La Figura 4 muestra un ejemplo de código de recursividad para calcular el factorial de un número.

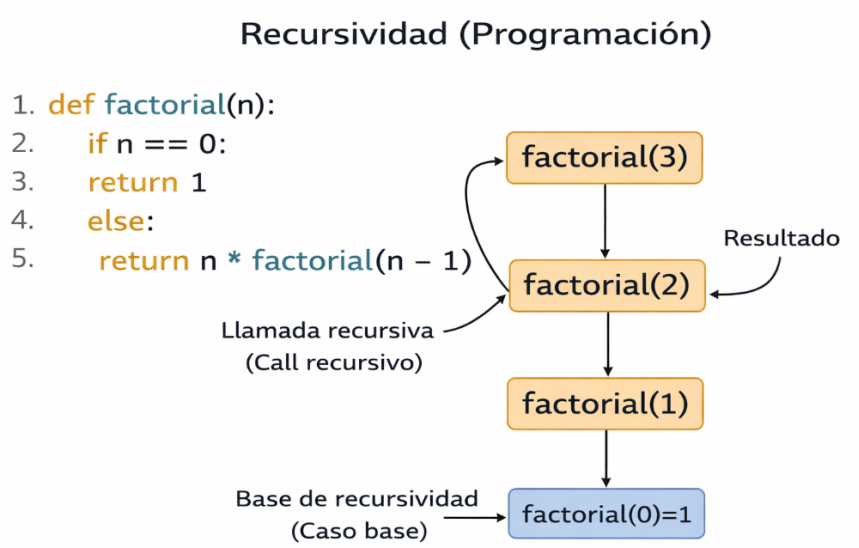


Figura 4. Ejemplo de recursividad. Creación de autor Patricio Coba

Este mecanismo facilita avanzar de forma natural por cada rama del grafo hasta llegar a un punto donde ya no existan más nodos por explorar, momento en el cual la función retorna al nivel anterior para continuar con otras rutas. Gracias a esto, la recursividad simplifica la implementación de algoritmos como DFS, haciendo que el código sea más claro, organizado y fácil de comprender.

Ejemplo simple:

```
def dfs(nodo, visitados):
  visitados.add(nodo)
  print(nodo)

for vecino in grafo[nodo]:
  if vecino not in visitados:
    dfs(vecino, visitados)
```

6.2. Algoritmos avanzados de grafos

Además de los recorridos básicos, existen otros que permiten analizar grafos de manera más profunda.
Estos algoritmos permiten:

- encontrar rutas óptimas
- analizar costos de conexión
- identificar caminos críticos
- optimizar infraestructuras de red

Uno de los problemas más importantes en grafos es el problema de caminos mínimos.

6.2.1. Caminos mínimos

El problema de caminos mínimos consiste en encontrar la ruta más corta entre dos nodos dentro de un grafo. Este permite determinar la mejor forma de desplazarse entre distintos

puntos dentro de una red. Los algoritmos de caminos mínimos buscan minimizar el costo total de un recorrido considerando los pesos asociados a las aristas del grafo (Cormen et al., 2016).

En grafos ponderados, cada conexión o arista posee un valor numérico denominado peso, el cual puede representar diferentes variables dependiendo del problema que se esté analizando, tales como:

- distancia
- tiempo
- costo
- nivel de riesgo

La Figura 6 muestra un ejemplo de recorridos mínimos.

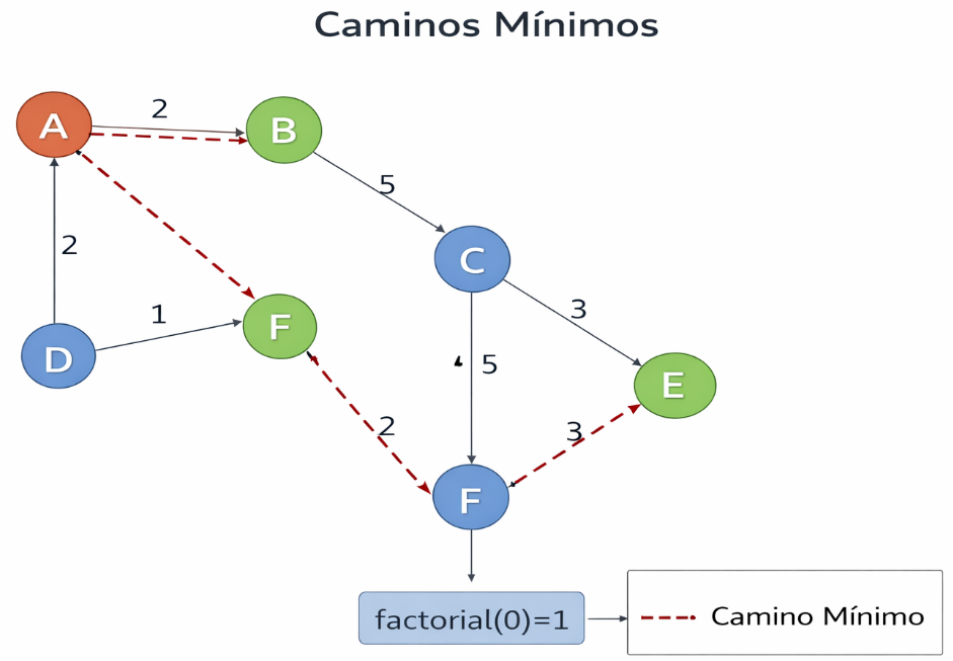


Figura 5. Ejemplo de recorridos mínimos. Creación de autor Patricio Coba

En redes, este problema puede aplicarse para identificar la ruta más eficiente para transmitir información entre dispositivos conectados. Puede utilizarse para determinar la ruta más rápida para enviar datos, el camino con menor latencia dentro de una red o el trayecto más probable de propagación de un ataque o malware dentro de una infraestructura tecnológica. Estos análisis permiten optimizar el funcionamiento de la red y mejorar las estrategias de seguridad informática.

6.2.2. Algoritmo de Dijkstra

Uno de los métodos más conocidos y utilizados para encontrar el camino mínimo entre nodos en un grafo ponderado, siempre que los pesos de las aristas sean no negativos, Este

algoritmo permite determinar la distancia mínima desde un nodo origen hacia todos los demás nodos del grafo (Cormen et al., 2016).

Dijkstra consiste en seleccionar en cada paso el nodo cuya distancia mínima desde el nodo inicial ya es conocida. Una vez seleccionado, se actualiza las distancias hacia sus nodos vecinos, verificando si existe un camino más corto a través de ese nodo. Si se encuentra una ruta más eficiente, se actualiza el valor de la distancia almacenada. Este proceso se repite de manera repetitiva hasta que todos los nodos han sido evaluados o hasta que se ha encontrado la ruta más corta hacia el nodo destino. Debido a su eficiencia, Dijkstra es ampliamente utilizado en sistemas de navegación, protocolos de enrutamiento de redes y análisis de infraestructuras informáticas.

La Figura 6 muestra un ejemplo de algoritmo de Dijkstra con la distancia mínima desde la raíz.

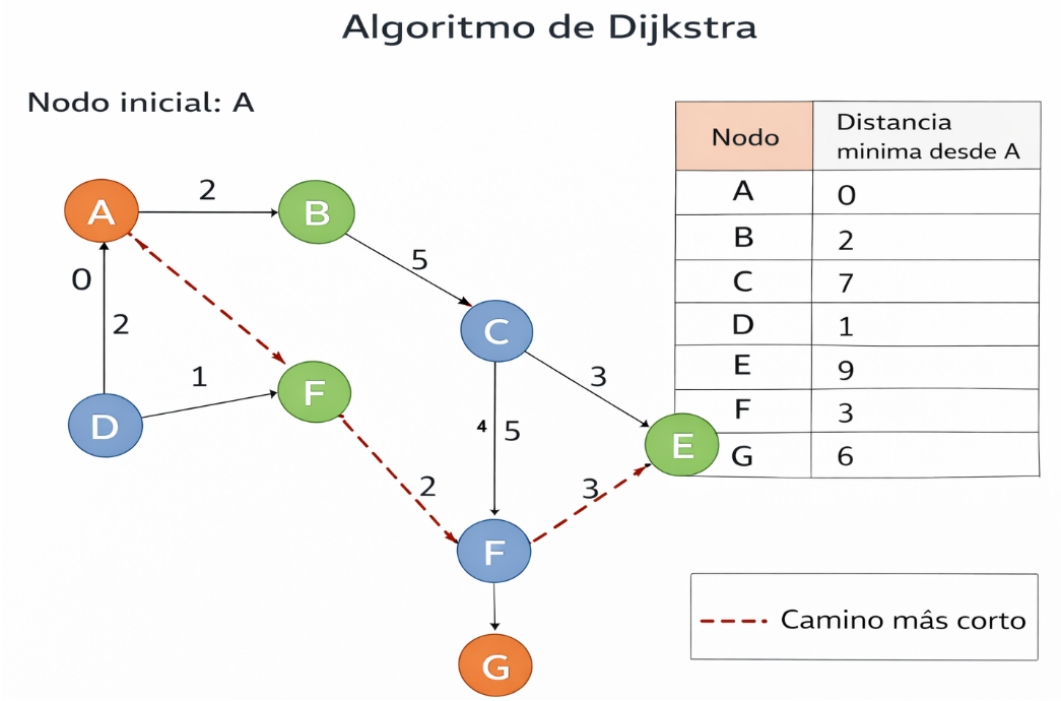


Figura 6. Ejemplo del algoritmo Dijkstra. Creación de autor Patricio Coba

Ejemplo en Python:

```
import heapq

grafo = {
 "Internet": [("Firewall",3)],
 "Firewall": [("ServidorWeb",2)],
 "ServidorWeb": [("ServidorApp",2)],
 "ServidorApp": [("BaseDatos",4)],
 "BaseDatos": []
}
```

```
def dijkstra(inicio):
 distancias = {nodo: float("inf") for nodo in grafo}
 distancias[inicio] = 0
 cola = [(0,inicio)]

 while cola:
  distancia_actual, nodo = heapq.heappop(cola)

  for vecino, peso in grafo[nodo]:
   distancia = distancia_actual + peso
   if distancia < distancias[vecino]:
    distancias[vecino] = distancia
    heapq.heappush(cola,(distancia,vecino))

 return distancias

print(dijkstra("Internet"))
```

Algoritmo de la ruta más corta de Dijkstra - Introducción gráfica y detallada. Página web en español, tutorial completo de FreeCodeCamp con explicaciones detalladas y gráficos para entender el algoritmo. Enlace: https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/

6.2.3. Análisis de rutas críticas

Consiste en identificar los caminos más importantes o influyentes dentro de una red, sistema o estructura representada mediante grafos. Estas rutas corresponden a secuencias de nodos y conexiones que tienen un impacto significativo en el funcionamiento del sistema, ya que concentran flujos de información, dependencias operativas o puntos clave de acceso. El análisis de caminos dentro de grafos permite comprender cómo se relacionan los componentes de una red y detectar rutas que influyen de manera determinante en su comportamiento (Kleinberg & Tardos, 2006).

El análisis de rutas críticas resulta útil para identificar posibles vectores de ataque o dependencias que podrían comprometer la estabilidad de un sistema. Estas rutas pueden representar:

- Caminos de acceso a sistemas sensibles.
- Dependencias entre servicios, donde la falla o compromiso de un componente afecta a otros.
- Posibles rutas de ataque.

La identificación de estas rutas permite a los especialistas priorizar las medidas de protección, concentrando los controles y mecanismos de defensa en los puntos más vulnerables o estratégicos de la red.

6.2.4. Optimización de recorridos

La optimización de recorridos busca mejorar la eficiencia al explorar un grafo.
Esto puede implicar:

- reducir el tiempo de búsqueda
- minimizar el costo de una ruta
- limitar la exploración de nodos innecesarios

Estas optimizaciones son importantes en sistemas que analizan grandes redes, como:

- sistemas de monitoreo de tráfico
- análisis de redes sociales
- plataformas de ciberseguridad

6.2.5. Aplicaciones en análisis de infraestructuras comprometidas

Hay múltiples aplicaciones en ciberseguridad, ya que permiten representar y estudiar las relaciones entre los distintos dispositivos, servicios y sistemas que forman parte de una infraestructura tecnológica.

Mediante el uso de grafos, los analistas pueden modelar una red como un conjunto de nodos conectados por aristas que representan las comunicaciones o dependencias entre ellos. Los algoritmos de grafos permiten analizar estructuras complejas de redes e identificar patrones de conexión relevantes para la toma de decisiones (Kleinberg & Tardos, 2006).

Entre las aplicaciones más importantes se encuentran:

- Análisis de movimiento lateral de atacantes, que permite estudiar cómo un intruso podría desplazarse entre distintos sistemas dentro de una red.
- Detección de rutas de propagación de malware, identificando los posibles caminos que podría seguir para infectar otros dispositivos.
- Identificación de nodos críticos dentro de una red.
- Análisis de vulnerabilidades en infraestructuras tecnológicas, permitiendo detectar puntos débiles o rutas potenciales de explotación.

Así, los especialistas pueden comprender mejor el comportamiento de una red, anticipar posibles escenarios de ataque y diseñar estrategias más efectivas para proteger los sistemas informáticos.

Definición de los términos citados en la Semana 6.

Recorridos en grafos. El recorrido en grafos es un proceso sistemático mediante el cual se visitan todos los vértices (nodos) de un grafo siguiendo ciertas reglas, con el objetivo de explorar su estructura, encontrar caminos o analizar relaciones entre sus elementos. Los métodos más comunes de recorrido son la búsqueda en profundidad (DFS) y la búsqueda en anchura (BFS), que difieren en la forma en que exploran los nodos del grafo.

Algoritmo de Dijkstra. El algoritmo de Dijkstra es un método utilizado en teoría de grafos para encontrar el camino más corto desde un nodo origen hacia todos los demás nodos en un grafo ponderado, siempre que los pesos de las aristas sean no negativos. Funciona seleccionando iterativamente el nodo con la menor distancia acumulada conocida y actualizando las distancias de sus vecinos, garantizando así la obtención de las rutas más cortas. (Dijkstra, 1959)

Recorridos en grafos aplicados a la ciberseguridad. Infografía que contiene la definición de recorridos y los algoritmos DFS y BFS, y su comparación.

Recorridos en grafos aplicados a ciberseguridad

¿Qué es un recorrido en un grafo?

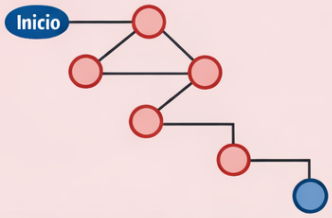
Un **recorrido** en un grafo es un algoritmo que permite visitar todos los nodos de una red siguiendo un conjunto de reglas.

Ciberseguridad:

Analizar cómo están conectados los **sistemas** y cómo puede propagarse un ataque en una infraestructura.

Búsqueda en profundidad (DFS)

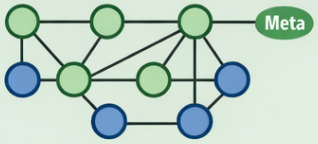
- Utiliza **recursividad o una pila**
- Explora **rutas completas** antes de probar otras
- Útil para estructuras complejas



- Simulación de rutas de ataque
- Análisis de **vulnerabilidades**
- Dependencias entre sistemas

Búsqueda en anchura (BFS)

- Utiliza una **cola**
- Análiza el grafo **por niveles**
- Encuentra **rutas minimas**



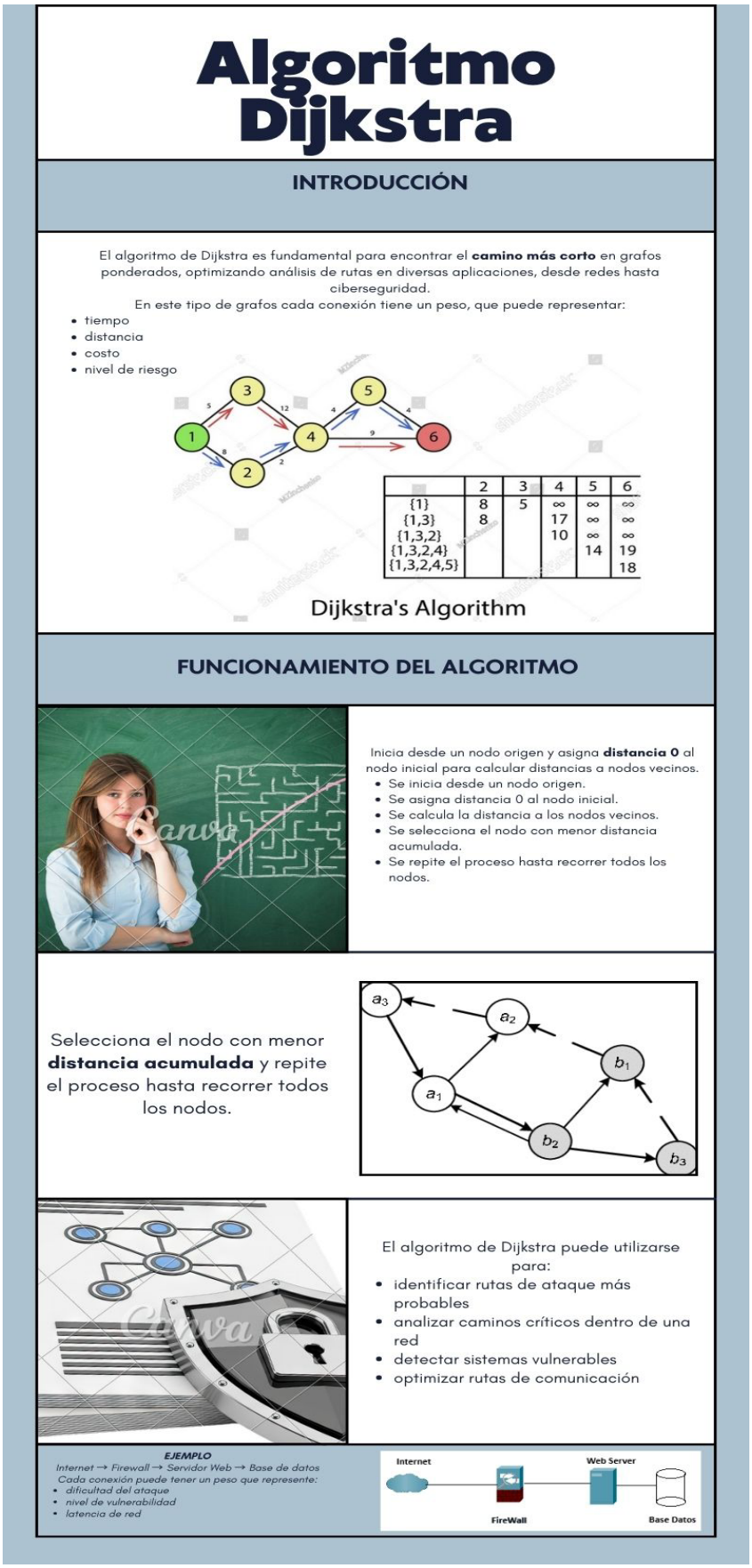
- Propagación de **malware**
- Identificación de **sistemas alcanzables**
- Detección de **movimiento lateral**

Ejemplo ▶ Internet → Firewall → Servidor → Base de Datos

Comparación DFS vs BFS

	DFS	BFS
Exploración	Profunda	Por niveles
Estructura	Pila o recursión	Cola
Uso principal	Explorar rutas	Encontrar caminos mínimos
Aplicación en seguridad	Análisis de rutas de ataque	Análisis de propagación

Algoritmo Dijkstra. Infografía con información general del algoritmo, su funcionamiento, utilidad y un ejemplo.



Laboratorio de Contenido 1: Recorridos en grafos aplicados al análisis de ataques

Explicación básica

Los algoritmos de recorrido en grafos permiten analizar cómo se conectan los diferentes sistemas dentro de una red informática. En ciberseguridad, estos algoritmos pueden utilizarse para estudiar cómo un atacante podría desplazarse dentro de una infraestructura tecnológica una vez que ha logrado comprometer un sistema inicial.

Mediante algoritmos como DFS o BFS es posible simular el comportamiento de un ataque y analizar qué sistemas podrían verse afectados.

Objetivos

- Comprender el concepto de recorrido en grafos.
- Identificar las diferencias entre DFS y BFS.
- Analizar posibles rutas de ataque dentro de una red.
- Representar una red informática mediante un grafo.
- Implementar algoritmos de recorrido utilizando Python.

Caso práctico

Una organización posee la siguiente infraestructura:

- Internet
- Firewall
- Servidor Web
- Servidor de Aplicaciones
- Base de Datos

El equipo de seguridad desea analizar cómo un atacante podría desplazarse dentro de la red si logra comprometer el servidor web.

Problema

Se requiere:

- Representar la infraestructura como un grafo.
- Aplicar un algoritmo de recorrido para analizar los sistemas accesibles.
- Simular el movimiento del atacante dentro de la red.

Aplicación paso a paso

Paso 1: Representar la red

Cada sistema será un nodo.

Internet → Firewall → Servidor Web → Servidor Aplicaciones → Base de Datos

Paso 2: Representar el grafo en Python

```
grafo = {
"Internet":["Firewall"],
"Firewall":["ServidorWeb"],
"ServidorWeb":["ServidorApp"],
"ServidorApp":["BaseDatos"],
"BaseDatos":[]
}
```

Paso 3: Aplicar DFS

```
def dfs(nodo, visitados):
    if nodo not in visitados:
        print(nodo)
        visitados.add(nodo)

        for vecino in grafo[nodo]:
            dfs(vecino, visitados)

dfs("Internet", set())
```

Conclusiones

Los algoritmos de recorrido permiten analizar la estructura de una red informática y estudiar cómo un ataque podría propagarse entre sistemas conectados. Este tipo de análisis es fundamental para identificar vulnerabilidades y diseñar estrategias de defensa más efectivas dentro de una infraestructura tecnológica.

Laboratorio de Contenido 3: Búsqueda en Anchura (BFS)

Explicación básica
El algoritmo BFS es un método de búsqueda para grafos y árboles. Su estrategia consiste en explorar todos los vecinos de un nodo antes de pasar al siguiente nivel de profundidad. Imagina que lanzas una piedra en un estanque: las ondas se expanden uniformemente desde el centro hacia afuera. BFS funciona igual, garantizando que, en grafos sin pesos, la primera vez que alcances un nodo, habrás llegado por el **camino más corto**.

Objetivos

- Comprender la estructura de datos tipo **Cola (FIFO)** y su rol en BFS.
- Implementar una búsqueda que encuentre la ruta mínima entre dos puntos.
- Analizar la complejidad temporal y espacial del algoritmo.

Caso práctico

Supongamos que trabajas en una red social. Quieres saber cuál es el "grado de separación" entre dos usuarios. Por ejemplo, ¿cuántos conocidos hay entre Ana y Juan? BFS es la herramienta perfecta para calcular esta distancia mínima de contactos.

Problema
Dado un mapa de conexiones (grafo), determinar si existe un camino entre un **Usuario A** (Inicio) y un **Usuario B** (Destino) y mostrar la ruta exacta.

Aplicación paso a paso
Para este laboratorio, utilizaremos un diccionario para representar el grafo y la clase deque para gestionar la cola de exploración.

Implementación en Python

Paso 1: Definir el Grafo

```
from collections import deque

# Red social: nombres y sus listas de amigos
red_social = {
   "Ana": ["Beto", "Carla"],
   "Beto": ["Ana", "David", "Elena"],
   "Carla": ["Ana", "Felipe"],
   "David": ["Beto"],
   "Elena": ["Beto", "Felipe"],
   "Felipe": ["Carla", "Elena"]
}
```

Paso 2: Implementar BFS para encontrar el camino

```
def buscar_camino_corto(grafo, inicio, destino):
    # La cola guarda tuplas: (nodo_actual, camino_recorrido)
    cola = deque([(inicio, [inicio])])
    visitados = {inicio}

    while cola:
        (nodo, camino) = cola.popleft()

        # Si encontramos el destino, retornamos el camino
        if nodo == destino:
            return camino

        # Explorar vecinos
        for vecino in grafo.get(nodo, []):
            if vecino not in visitados:
                visitados.add(vecino)
                # Creamos un nuevo camino sumando el vecino al actual
                cola.append((vecino, camino + [vecino]))

    return None # No hay conexión

# Ejecución
ruta = buscar_camino_corto(red_social, "Ana", "Felipe")
print(f"La ruta más corta es: {ruta}")
```

Conclusiones

- **Eficacia en Rutas Cortas:** BFS es óptimo para encontrar el camino mínimo en grafos donde todas las aristas tienen el mismo "costo".
- **Gestión de Memoria:** A diferencia de DFS (Búsqueda en Profundidad), BFS puede consumir mucha memoria en grafos con un factor de ramificación alto, ya que debe almacenar todos los nodos de un nivel en la cola.
- **Uso de la Cola:** La propiedad **FIFO** (First-In, First-Out) es lo que garantiza que exploremos por niveles.

Laboratorio de Contenido 3: Algoritmo de Dijkstra aplicado al análisis de rutas de ataque

Explicación básica
Los grafos ponderados permiten representar redes donde cada conexión tiene un valor asociado. En ciberseguridad, este valor puede representar el nivel de dificultad que tendría un atacante para comprometer un sistema.
El algoritmo de Dijkstra permite calcular el camino con menor costo dentro de este tipo de grafos.

Objetivos

- Comprender el concepto de caminos mínimos.
- Identificar cómo funcionan los grafos ponderados.
- Aplicar el algoritmo de Dijkstra.
- Analizar rutas de ataque dentro de una red informática.

Caso práctico
Una empresa desea analizar su infraestructura tecnológica.

Los sistemas identificados son:

- Internet
- Firewall
- Servidor Web
- Servidor Aplicaciones
- Base de Datos

Cada conexión tiene un valor que representa la dificultad del ataque.

Problema
El equipo de seguridad desea identificar cuál sería el camino más sencillo para que un atacante llegue a la base de datos.

Aplicación paso a paso

Paso 1: Definir las conexiones
Internet → Firewall (3)
Firewall → Servidor Web (2)
Servidor Web → Servidor Aplicaciones (2)
Servidor Aplicaciones → Base de Datos (4)

Paso 2: Representar el grafo

```
grafo = {
"Internet":[("Firewall",3)],
"Firewall":[("ServidorWeb",2)],
"ServidorWeb":[("ServidorApp",2)],
"ServidorApp":[("BaseDatos",4)],
"BaseDatos":[]
}
```

Paso 3: Aplicar Dijkstra

```
import heapq

def dijkstra(inicio):
    distancias = {nodo:float("inf") for nodo in grafo}
    distancias[inicio] = 0

    cola = [(0,inicio)]

    while cola:
        distancia_actual, nodo = heapq.heappop(cola)

        for vecino,peso in grafo[nodo]:
            distancia = distancia_actual + peso

            if distancia < distancias[vecino]:
                distancias[vecino] = distancia
                heapq.heappush(cola,(distancia,vecino))

    return distancias

print(dijkstra("Internet"))
```

Conclusiones

El algoritmo de Dijkstra permite analizar rutas dentro de redes complejas y determinar cuáles son los caminos más eficientes o vulnerables dentro de una infraestructura tecnológica.

Este tipo de análisis es especialmente útil para identificar puntos críticos dentro de una red informática.

Estudio de caso 2: Simulación de Propagación de Malware (BFS)

Contexto

La empresa **"TechLogistics"** posee una infraestructura de red híbrida con 100 servidores interconectados. El departamento de seguridad quiere poner a prueba su estrategia de segmentación. Para ello, necesitan simular el "radio de explosión" (*blast radius*) de un nuevo tipo de ransomware que explota una vulnerabilidad en el protocolo SMB para saltar de un equipo a otro.

Problema

Si un solo terminal de ventas en una oficina remota es infectado, ¿cuántos saltos (*hops*) le toma al malware llegar al servidor de base de datos central? ¿Qué equipos se verán comprometidos en la primera, segunda y tercera hora si el malware tarda 1 hora en infectar un nodo vecino?

Solución Propuesta

Utilizar el algoritmo **BFS (Breadth-First Search)** para simular la propagación nivel por nivel. Dado que el malware se propaga a todos los vecinos disponibles simultáneamente, BFS representa perfectamente este crecimiento radial, a diferencia de DFS que solo seguiría una línea de infección.

Estrategia:

1. **Modelado:** Los nodos son dispositivos (PCs, Servidores, Switch) y las aristas son las conexiones físicas o lógicas.
2. **Estado de Infección:** Mantener una lista de nodos "sanos", "infectados" y "en proceso".
3. **Análisis por Capas:** Cada nivel del BFS representará una "Ola de Infección" (Tiempo).
4. **Detección de Cortafuegos:** Si una arista conecta con una VLAN protegida, se le asigna una probabilidad de bloqueo (en esta simulación básica, asumiremos paso libre para medir el peor escenario).

Implementación

```python
from collections import deque

def simular_propagacion(red, inicio):
    cola = deque([(inicio, 0)])  # (Nodo actual, Hora de infección)
    infectados = {inicio: 0}

    print(f"--- Inicio de Simulación: Paciente Cero [{inicio}] ---")

    while cola:
        nodo_actual, hora = cola.popleft()

        # Simulamos la propagación a vecinos no infectados
        for vecino in red.get(nodo_actual, []):
            if vecino not in infectados:
                infectados[vecino] = hora + 1
                cola.append((vecino, hora + 1))
                print(f"Hora {hora + 1}: El nodo [{vecino}] ha sido comprometido por [{nodo_actual}]")

    return infectados
```

```python
# Mapa de la Red Corporativa
red_empresa = {
    "PC_Ventas_01": ["Switch_Oficina", "PC_Ventas_02"],
    "PC_Ventas_02": ["PC_Ventas_01", "Switch_Oficina"],
    "Switch_Oficina": ["Router_Principal", "PC_Ventas_01", "PC_Ventas_02"],
    "Router_Principal": ["Switch_Oficina", "Firewall_Core"],
    "Firewall_Core": ["Router_Principal", "Servidor_AD", "Switch_DataCenter"],
    "Switch_DataCenter": ["Firewall_Core", "DB_Produccion", "Backup_Server"],
    "Servidor_AD": ["Firewall_Core"],
    "DB_Produccion": ["Switch_DataCenter"],
    "Backup_Server": ["Switch_DataCenter"]
}

# Ejecución de la simulación
log_infeccion = simular_propagacion(red_empresa, "PC_Ventas_01")
```

Resultados e Impacto en ciberseguridad

- **Identificación de Nodos Críticos:** La simulación muestra que el Firewall_Core es el punto de inflexión. Si este nodo cae, el malware accede al Data Center en solo 1 salto adicional.
- **Cálculo del Tiempo de Respuesta:** Si el equipo de IT tarda 3 horas en reaccionar, la simulación revela que para la **Hora 3**, el Firewall_Core ya habría sido alcanzado, comprometiendo toda la red.
- **Estrategia de Contención:** Gracias al BFS, se pueden identificar "puntos de estrangulamiento" (*choke points*) donde colocar sistemas de detección de intrusos (IDS) para romper la cadena de infección.

Reflexión Técnica

BFS es la herramienta estándar para el análisis de alcance en redes. Mientras que Dijkstra busca la ruta más rápida, BFS nos da la expansión total.

En un escenario real de ciberseguridad, esta simulación se refina convirtiéndola en un BFS Estocástico, donde la infección de un vecino no es segura (100%), sino que depende de una probabilidad basada en las vulnerabilidades detectadas en cada nodo.

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.

Educación de calidad

REFERENCIAS

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2016). Introducción a los algoritmos (3.ª ed.). MIT Press / McGraw-Hill Interamericana.

Bhargava, A. (2017). Grokking Algorithms: An illustrated guide for programmers and other curious people. Manning Publications.

Kleinberg, J., & Tardos, É. (2006). Algorithm design. Pearson.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1, 269–271. https://doi.org/10.1007/BF01386390

Educación de calidad

GRADO / POSGRADO / TECNOLOGÍAS

CONTENIDO DE PROFUNDIZACIÓN

PUCE Virtual

síguenos en:

www.pucevirtual.puce.edu.ec