Análisis de Redes: Representación y Cálculo de Caminos Más Cortos Usando Grafos

Luis Solera Alfaro

Vannessa Espinoza Nieves

Ernesto Alvarez Meza

BMA06 ÁLGEBRA LINEAL

Luis Mejias Molina

Tercer cuatrimestre

Resumen

El objetivo del trabajo de investigación es desarrollar un programa que represente redes informáticas mediante matrices de adyacencia y analizar propiedades específicas para determinar los caminos más cortos entre un origen y un destino. Utilizando la librería de terceros "NetworkX" se construye un grafo que incluye aristas con pesos, como el ancho de banda entre dos puntos, y nodos, los cuales son equipos informáticos de redes de datos, como los enrutadores. Esto permite representar diversas relaciones en redes de computadoras. Posteriormente, se implementa el algoritmo de Dijkstra para calcular el camino más corto entre dos nodos, demostrando así su eficacia para encontrar soluciones óptimas en grafos ponderados. Los resultados demuestran que el programa es capaz de identificar rutas eficientes, siendo fundamental en aplicaciones como la planificación de redes de transporte de datos y la optimización de rutas en sistemas de información. Además, se exploraron extensiones del proyecto, como la interactividad y la visualización gráfica, para mejorar la lectura de los datos. Las conclusiones resaltan la importancia de los grafos en el análisis de redes y sugieren que la combinación de herramientas matemáticas y computacionales puede proporcionar soluciones efectivas a problemas complejos en diversas disciplinas. Este enfoque no sólo demuestra la utilidad de los algoritmos en la práctica, sino que también abre la puerta a investigaciones futuras en el campo del análisis de redes.

Introducción

Análisis de Redes: Representación y Cálculo de Caminos Más Cortos Usando Grafos

¿Cómo Se Aplican las Matrices a la Carrera de Ingeniería Telemática?

Las matrices se aplican en gran variedad de campos y en el área de la ingeniería telemática se pueden utilizar para el análisis y gestión de datos de redes de computadoras. En esta investigación el enfoque estará en la representación de redes informáticas mediante matrices de adyacencia y en el cálculo del camino más corto entre dos objetos.

Objetivo General

Desarrollar un programa con el lenguaje de programación Python que represente redes informáticas utilizando grafos y matrices de adyacencia, el cual permita analizar propiedades para determinar los caminos más cortos entre un origen y destino mediante el uso del algoritmo de Dijkstra.

Objetivos Específicos

Construir grafos. Utilizando la librería de terceros “NetworkX” se representara redes de datos, incluyendo nodos y aristas con pesos.

Efectuar el algoritmo de Dijkstra. Este algoritmo será utilizado para calcular el camino mas corto entre dos nodos específicos.

Analizar los resultados de las pruebas. Mediante grafos y un archivo CSV se podrá visualizar el mejor y peor camino, esto con el fin de medir la eficacia del algoritmo y determinar si se utilizó el camino más corto.

Implementar herramientas de visualización. Por medio del uso de la libraría de terceros “Matplotlib” se podrá graficar el camino preferido entre los dos nodos. Esto brindará una experiencia más amigable al usuario.

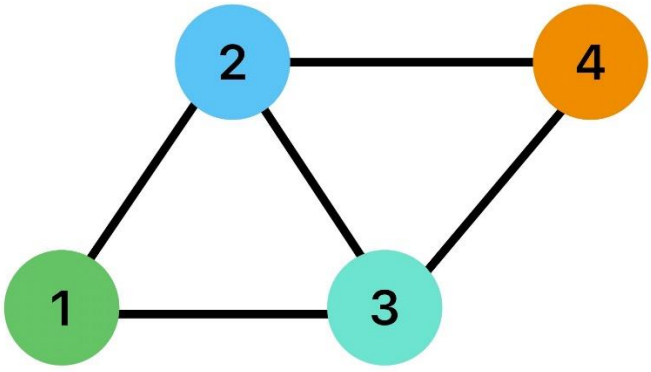
Justificación

El análisis de redes de computadoras mediante el uso de grafos y matrices es de suma relevancia dentro del campo de la ingeniería telemática ya que esta permite entender y optimizar las interacciones entre complejos sistemas informáticos. En la actualidad, prácticamente todas las personas están interconectadas de una u otra forma, por lo cual la capacidad de analizar y gestionar las redes informáticas es esencial para garantizar una comunicación eficaz.

Marco Teórico

¿Qué es un Grafo?

Un grafo es un esquema matemático que permite representar los enlaces entre diferentes entidades. Está formado por un conjunto de nodos, que representan las entidades, y ejes que representan las relaciones o conexiones entre entidades. Los grafos son utilizados en disciplinas como la informática, redes sociales, optimización de algoritmos, análisis de datos, entre otros. Ayudan a visualizar y solucionar problemas de conectividad, rutas esqueléticas y agrupamiento interrelacionados y a menudo complejos, por ejemplo, en una red de metro, cada estación sería un nodo y cada línea que conecta dos estaciones sería un eje. Con un grafo, puede encontrar la ruta más corta entre dos lugares, maximizar el tiempo de desplazamiento o encontrar conexiones óptimas.



Los nodos son los círculos de colores y los arcos son las líneas que los conectan.

¿Qué es una Matriz de Adyacencia?

Una matriz de adyacencia es otra forma de representar un grafo en programación. Es una matriz bidimensional donde cada celda indica si existe una conexión entre los dos nodos o no. Se utiliza para consultas rápidas cuando se necesita comprobar la presencia de un enlace. Por ejemplo, imagine un grafo con nodos A, B y C; A está conectado con B y B con C, pero A no está conectado con C. En este caso, la matriz de adyacencia será la siguiente:

Tabla 1

Matriz de adyacencia

Nodos	A	B	C
A	0	1	0
B	1	0	1
C	0	1	0

Nota: Esta tabla muestra los Nodos A,B y C en la columna 1 y fila 1. Donde los unos significan una conexión entre ellos y el 0 significa lo contrario.

La matriz de adyacencia se utiliza para resolver diversos problemas en computación, puesto que la representación es eficiente y fácil de resolver, se utiliza mucho para modelar sistemas complejos como redes de transporte, comunicación y redes de datos, permitiendo así la optimización.

¿Qué es Dijkstra y Dónde se Utiliza?

Dijkstra son las bases para la programación y la resolución de problemas. A través de ellos, se realiza un recorrido eficiente y ahorrador en recursos haciendo que los procesos sean eficientes. El algoritmo Dijkstra es como un GPS muy inteligente, que ayuda a encontrar la forma más fácil y eficiente en que una persona puede llegar de un punto a otro en la red; este

algoritmo es utilizado con frecuencia en redes informáticas, sistemas de navegación, estudios de redes sociales, diseños de circuitos integrados, videojuegos, organizadores de rutas, entre otros.

¿Cómo se usa una Matriz de Adyacencia en Python para Representar un Grafo?

En Python, una forma común de implementar una matriz de adyacencia es utilizando listas anidadas:

```
# Grafo no dirigido con 4 nodos
graph = [[0, 1, 1, 0],
         [1, 0, 1, 1],
         [1, 1, 0, 1],
         [0, 1, 1, 0]]
```

La adyacencia existe entre los nodos 0 y 1 y viceversa ya que graph[0][1] y graph[1][0] son 1. Sin embargo, “edge” no existe entre los nodos 0 y 3 ya que graph[0][3] y graph[3][0] tienen 0. Para ello, a continuación, podemos usar la matriz de adyacencia como entrada para la implementación de ciertos algoritmos de búsqueda de caminos:

```
import sys

def dijkstra(graph, src):
    row = len(graph)
    col = len(graph[0])

    # Inicializar distancias y nodos visitados
    dist = [sys.maxsize] * row
    dist[src] = 0
    sptSet = [False] * row

    for cout in range(row):
        # Encontrar el nodo con la distancia mínima desde el nodo de
            origen
        u = minDistance(dist, sptSet)

        # Marcar el nodo como visitado
        sptSet[u] = True

        # Actualizar las distancias de los nodos adyacentes
        for v in range(col):
            if graph[u][v] > 0 and sptSet[v] == False and \
                    dist[v] > dist[u] + graph[u][v]:
                dist[v] = dist[u] + graph[u][v]
```

```
24
25      # Imprimir las distancias más cortas desde el nodo de origen
26      print(dist)
27
28  # Función para encontrar el nodo con la distancia mínima
29  def minDistance(dist, sptSet):
30      # ... (Implementación de la función minDistance)
31      # ... (Busca el nodo con la distancia mínima entre los nodos no
            visitados)
```

Este algoritmo dice que, para iniciar, se inicializan las distancias a todos los nodos como infinitas, excepto al nodo de origen; después, se selecciona el nodo con la distancia mínima entre los no visitados; también se actualizan las distancias de los vecinos del nodo seleccionado si la nueva distancia es menor que la anterior; para finalizar se repiten los pasos 2 y 3 hasta que todos los nodos hayan sido visitados.

De esta manera obtenemos una sencilla implementación, puesto que es fácil de crear y entender, es eficiente para verificar adyacencias, al denotar si dos nodos son adyacentes en tiempo constante, y es adecuada para grafos densos, cuando la mayoría de los pares de nodos están conectados, la matriz de adyacencia es eficiente en términos de espacio.

¿Qué es NetworkX?

NetworkX es una herramienta para crear redes en Python. Su librería nos ayuda a manipular, crear y analizar los grafos por medio de una estructura basada en nodos y aristas. Para aprovechar su funcionalidad en la creación de grafos, y lograr entender mejor la estructuración de NetworkX en Phyton, se muestra a continuación un ejemplo de un grafo:

Primero. Instalar e importar la librería de terceros como nx:

```
pip install networkx
import networkx as nx
```

Segundo. Creación de un grafo vacío:

```
G = nx.Graph()
```

Tercero. Se agregan los nodos individualmente o desde una lista:

```
G.add_node(1)
G.add_nodes_from([2, 3, 4])
```

Cuarto. Agregar aristas (conexiones) entre los nodos:

```
G.add_edge(1, 2)
G.add_edges_from([(2, 3), (3, 4)])
```

Quinto. (Opcional) también se puede añadir peso a las aristas con el código:

```
G.add_edge(1, 3, weight=2.5)
```

Sexto. Para visualizarlo se integra con matplotlib, el cual es explicado más adelante.

Séptimo. (Opcional) se puede acceder a los nodos (nodes) y aristas (edges) existentes para inspeccionarlos:

```
print(G.nodes)
print(G.edges)
```

Octavo. (Opcional) se les agregan atributos a los nodos:

```
G.nodes[1]['color'] = 'blue'
print(G.nodes.data())
```

Octavo. (Opcional) se les agregan atributos a las aristas:

```
G[1][2]['weight'] = 3.5
print(G.edges.data())
```

Noveno. Algoritmo de busaqueda de caminos (buscar el camino más corto)

```
path = nx.shortest_path(G, source=1, target=4, weight='weight')
print(path)
```

Implementación del Algoritmo de Dijkstra

¿Cómo funciona el algoritmo de Dijkstra para calcular el camino más corto?

Lo hace a partir de grafo ponderado, es decir un grafo en el que los arcos tienen valores asociados o pesos que representan costos, distancias, o tiempos (puede ser cualquier métrica relevante). Para encontrar la ruta más corta el algoritmo inicia asignando una distancia de cero al nodo origen y una distancia infinita a los demás nodos. Luego selecciona el nodo con la distancia

menor y se exploran sus vecinos, va actualizando las distancias seleccionando las de menor costo para llegar al nodo deseado a través del nodo actual. Este proceso lo hará tantas veces como sea necesario hasta que se exploren todos los nodos o se encuentre la distancia más corta al objetivo.

¿Qué pasa si un camino no está disponible?

Si un camino no está disponible en un grafo, significa que no existe una conexión directa entre ciertos nodos, o bien, que todos los caminos posibles hacia ese nodo en específico están bloqueados o es inalcanzable y de ahí se establecen 3 puntos importantes.

Distancia infinita. La distancia de los nodos inalcanzables se mantendrán en infinita desde el nodo inicial. Esto para dar a entender que no hay una ruta posible para ese nodo desde el punto de origen.

Ausencia de camino mínimo. A la hora de buscar un camino que no existe el algoritmo indicará que no hay solución o va a omitir el nodo en el resultado. Esto es de mucha ayuda en ciertas ramas como telecomunicaciones o ruteo de redes para avisar que hay puntos para reparar o mejorar.

Efecto en grafos no conexos. Puede suceder también en grafos grandes y complejos que estén aislados entre sí, y Dijkstra solo encuentra caminos que incluyan al nodo inicial. Si el nodo objetivo está en otro componente, Dijkstra no encontrará un camino hacia él.

¿Cómo se Ejecuta el Algoritmo de Dijkstra Utilizando NetworkX en Python?

Para poder usar Dijkstra en Python lo primero a realizar es crear un grafo, donde es necesario definir las adyacencias con sus respectivos pesos asignados a las aristas y luego utilizar la función nx.dijkstra_path de NetworkX la cual es una implementación optimizada del algoritmo de Dijkstra, la cual puede ser utilizada para calcular el camino más corto entre ambos nodos. A continuación, presentamos un ejemplo de uso:

Primero. La función nx.dijkstra_path recibe varios paramteros, entre ellos el grafo, origen, destino y el valor del peso. El retorno se guarda dentro de la variable camino:

```
camino = nx.dijkstra_path(G, source=srcNode, target=dstNode,
weight=weight)
```

Segundo. Podemos imprimir cual es el camino mas corto:

```
print(camino)
```

¿Cómo se puede analizar los resultados?

Para analizar los resultados y comprobar si realmente estamos usando el camino más corto entre un punto A y un punto B, se va a utilizar Dijkstra. De igual manera, basados en los resultados de las pruebas, se van a crear dos grafos y un archivo CSV. Esto con el fin de visualizar los datos de una forma más sencilla y manejarlos como una matriz. Primero se hace uso de la función de NetworkX all_simple_paths para encontrar todos los posibles caminos entre los dos puntos. Seguidamente, para cada camino descubierto, se calcula la longitud entre el origen y el destino con el uso del peso de las aristas. Posteriormente, se emplea la función de NetworkX dijkstra_path para determinar el camino más corto, la cual está diseñada para utilizar el algoritmo de Dijkstra. Una vez tenemos el camino más corto y todos los otros posibles caminos, estos se representan gráficamente y, con la ayuda de colores, podemos diferenciar cuál es el camino preferido.

El cálculo del peor camino se realiza usando la función incorporada de Python llamada max. Esta función encuentra el valor máximo; por ejemplo, entre los números 1, 2, 3, 4 y 5, max selecciona y guarda el número 5. Por lo tanto, se realiza una comparación entre el retorno de all_simple_paths, es decir, las longitudes de todos los caminos. A continuación, max procede a tomar el valor superior, que se clasifica como el camino menos deseado. Finalmente, dentro del

archivo CSV se escriben los datos en forma de matriz, donde se encontrarán todas las rutas disponibles, el peso y una indicación de cuál es el mejor y el peor camino.

¿Cómo se visualizan grafos en Python usando NetworkX y otras librerías como Matplotlib?

NetworkX permite construir y manipular grafos, mientras que Matplotlib se utiliza para visualizarlos de manera atractiva, donde se puede resaltar caminos específicos y es posible personalizarlo con estilos para una mejor comprensión. La implementación de ambas librerías permite el manejo y representación de datos de forma eficiente, primero es necesario la creación de un grafo, asignar los respectivos pesos, calcular el mejor camino y seguidamente modificar la representación graficas de los datos a preferencia, a continuación, un código de ejemplo:

Primero. Generar una distribución basada en fuerzas para que los nodos estén visualmente bien distribuidos, la función recibe el grafo:

```
pos = nx.spring_layout(G)
```

Segundo. Darle características visuales a los nodos utilizando la función nx.draw la cual recibe distinto valores entre ellos: se le habilitan las etiquetas (nombres de los nodos), tamaño del nodo, color del nodo, y tipo de letra.

```
nx.draw(G, pos, with_labels=True, node_color='lightblue',
node_size=500, font_weight =’bold’)
```

Tercero. Mostrar las etiquetas del peso de las aristas:

```
edge_labels = nx.get_edge_attributes(graph, 'weight')
nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels)
```

Cuarto. Resaltar el mejor camino:

```
path_edges = list(zip(camino, camino [1:]))
nx.draw_networkx_edges(graph, pos, edgelist=path_edges,
edge_color='red', width=2.5)
```

Quinto. Mostrar gráficamente la representación del grafo.

```
plt.show()
```