Práctica 02

Hiram Caleb Gutiérrez Olague

Análisis y diseño de algoritmos
1ro de diciembre de 2023

1 Introducción

La resolución del problema de encontrar los caminos mínimos en un grafo ponderado es esencial en diversos contextos, desde redes de transporte hasta sistemas de información. El algoritmo de Dijkstra es una herramienta valiosa en este sentido, y en esta práctica, se explorará su funcionamiento y se implementará en Python.

2 Algoritmo

2.1 Definición del Problema

El problema de caminos mínimos en grafos ponderados busca encontrar la ruta más corta desde un nodo de origen a todos los demás nodos en el grafo, considerando los pesos asociados a las aristas. Este problema es central en la teoría de grafos y tiene aplicaciones prácticas en la planificación de rutas en redes de transporte, optimización de rutas de paquetes en redes de comunicación y en la determinación de la distancia más corta entre ubicaciones en mapas.

2.2 Descripción del Algoritmo de Dijkstra

El algoritmo de Dijkstra, desarrollado por Edsger Dijkstra en 1956, es un algoritmo voraz que resuelve el problema de caminos mínimos en grafos dirigidos con pesos no negativos. A continuación se detalla su funcionamiento:

2.2.1 Inicialización

Se asigna una distancia inicial de 0 al nodo de inicio y de infinito a todos los demás nodos. Se mantiene una cola de prioridad (min-heap) que contiene los nodos no visitados, ordenados por sus distancias actuales.

2.2.2 Iteración

Mientras la cola de prioridad no esté vacía, se extrae el nodo con la distancia mínima. Se actualizan las distancias de los nodos adyacentes si se encuentra una ruta más corta a través del nodo actual.

2.2.3 Finalización

Una vez que todos los nodos han sido visitados o la cola de prioridad está vacía, el algoritmo termina.

2.2.4 Resultado

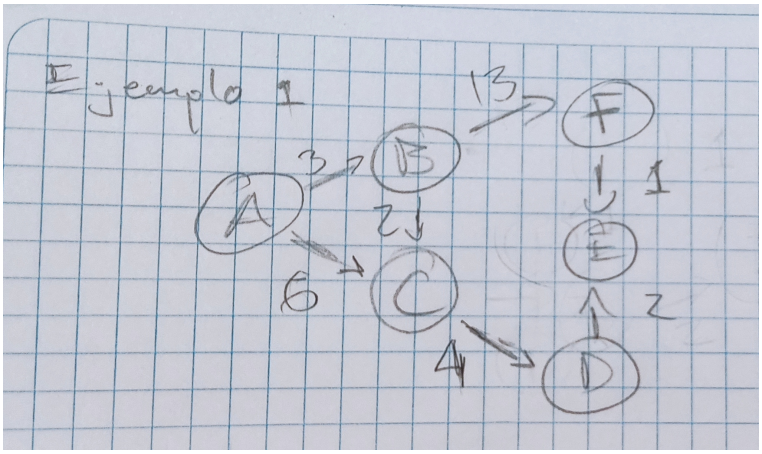
Al finalizar, las distancias mínimas desde el nodo de inicio a todos los demás nodos se han calculado y se pueden utilizar para determinar las rutas más cortas.

3 Desarrollo

3.1 Construcción del grafo y aplicación del algoritmo de Dijkstra

3.1.1 Grafo

Primero vamos a inventar un grafo, creando los vertices y aristas correspondientes:



Podemos observar la ponderación y dirección de los valores dentro de nuestro grafo, dicho sea de paso que la direccion dentro de nuestra codificación es sumamente relevante.

3.1.2 Algoritmo de Dijkstra en nuestro grafo

Ya tenemos nuestro grafo, ahora podemos decir que queremos la distancia de menor valor posible desde nuestro vertice A hasta todos los otros vertices.

- De A a B: la ruta con menos valor dentro de nuestro grafo es $A \to B$ con un valor de 3.
- De A a C: la ruta con menos valor dentro de nuestro grafo es $A \to B \to C$ con un valor de 5, a pesar de que exista la ruta $A \to C$ no es la ruta que tiene menos valor dentro de nuestro grafo.
- De A a D: la ruta con menos valor (y única) dentro de nuestro grafo es $A \to B \to C \to D$ con un valor de 9.
- De A a E: la ruta con menos valor dentro de nuestro grafo es $A \to B \to C \to D \to E$ con un valor de 11.
- De A a E: la ruta con menos valor dentro de nuestro grafo es $A \to B \to F$ con un valor de 16, el valor de nuestra ruta podría haber sido 12 pero la dirección que tienen las aristas importa, por lo que no se podía pasar del vertice E al vertice F.

3.2 Implementación en python

3.2.1 Implementación del grafo

Para la creación del grafo simplemente declaramos nuestros vertices junto con las aristas que le corresponden a cada una de ellas.

```
grafo = Grafo()
grafo.agregar_vertice("A")
grafo.agregar_vertice("B")
grafo.agregar_vertice("C")
grafo.agregar_vertice("D")
grafo.agregar_vertice("E")
grafo.agregar_vertice("F")
grafo.agregar_arista("A", "B", 3)
grafo.agregar_arista("A", "C", 6)
grafo.agregar_arista("B", "C", 2)
grafo.agregar_arista("C", "D", 4)
grafo.agregar_arista("C", "E", 8)
grafo.agregar_arista("B", "F", 13)
grafo.agregar_arista("F", "E", 1)
```

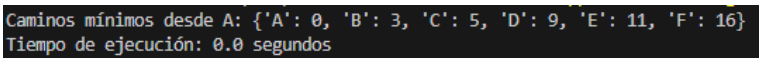
grafo.agregar_arista(”D”, ”E”, 2)

3.2.2 Prueba del Algoritmo de Dijkstra en nuestro grafo

Ya teniendo el grafo construido, debemos declarar cual va a ser nuestro vertice inicial, en este caso va a ser el vertice A.
vertice_inicio = ”A”
resultado = dijkstra(grafo, vertice_inicio)
print(f’Caminos mínimos desde vertice_inicio: resultado”)

3.2.3 Medición del Tiempo dentro del codigo

vertice_inicio = ”A”
inicio_tiempo = time.time()
resultado = dijkstra(grafo, vertice_inicio)
tiempo = time.time() - inicio_tiempo
print(f’Caminos mínimos desde vertice_inicio: resultado”)
print(f’Tiempo de ejecución: tiempo segundos”)
Simplemente se implementaron nuestras variables inicio_tiempo y tiempo para poder medir los segundos que se tardó en ejecutar el codigo.



3.2.4 Cálculo de Complejidad

Tiempo: O((V + E) * log(V)), donde V es el número de vértices y E es el número de aristas en el grafo. La complejidad se debe a la operación de extracción y la inserción en la cola de prioridad para cada vértice adyacente, y se multiplica por el logaritmo del número de vértices debido a la implementación de la cola de prioridad utilizando un montículo binario.

Espacio: O(V), donde V es el número de vértices en el grafo. Esto se debe al hecho de que se mantiene un diccionario (distancias) para almacenar las distancias mínimas desde el vértice de inicio a cada uno de los demás vértices.

En resumen, el algoritmo de Dijkstra tiene un rendimiento muy eficiente para grafos densos (muchas aristas) o dispersos (pocas aristas), pero el factor logarítmico en la cola de prioridad hace que el tiempo de ejecución sea sensible al número de vértices.

4 Conclusiones

La implementación del algoritmo de Dijkstra es una forma efectiva de encontrar los caminos mínimos en un grafo ponderado, y gracias a su eficiencia, es una opción sólida para resolver problemas de rutas y redes en aplicaciones del mundo real.

5 Referencias

Rey Vela, N. E. (2018). Implementación del algoritmo pulso en Python.
Vela, N. E. R. (2018). Implementación del algoritmo pulso en Python (Doctoral dissertation, Uniandes).