INSTITUTO POLITÉCNICO NACIONAL

Unidad Profesional Interdisciplinaria de Ingeniería
Campus Zacatecas.

Materia:

Análisis y Diseño de Algoritmos

Práctica 03: Implementación y Evaluación del Algoritmo de Dijkstra.

Docente:

M. en C. Erika Sánchez-Femat

Nombre del alumno:

Dalia Naomi García Macías

Fecha de entrega:

1 de Diciembre de 2023

Índice

1. Introducción **3**

2. Desarrollo de la practica **3**

2.1. Definición del problema 3

2.2. Descripcion del algoritmo 3

2.3. Implementacion del algoritmo 3

2.4. Pruebas del funcionamiento 5

3. Análisis del algoritmo **7**

3.1. Medición del tiempo de ejecución 7

3.2. Cálculo de la complejidad 8

4. Codigo completo **8**

5. Resultados (Capturas) **11**

6. Conclusiones **15**

7. Referencias **15**

1. Introducción

Imagina que estás planeando un viaje por carretera y deseas encontrar la ruta más corta desde tu casa hasta un destino. Para tomar decisiones informadas sobre qué caminos tomar, podrías considerar la distancia entre las ciudades y elegir la ruta que minimice esa distancia.

El algoritmo de Dijkstra funciona de manera similar pero para resolver problemas de optimización en redes, como encontrar la ruta más corta entre nodos en un grafo. En un grafo, los nodos representan ubicaciones y las aristas entre ellos representan las conexiones. Cada arista tiene un "peso" que indica la distancia o el costo asociado con moverse de un nodo a otro.

La idea básica del algoritmo es explorar todas las posibles rutas desde un punto de inicio hasta todos los demás puntos, registrando la distancia más corta conocida en cada paso. Comienza desde el nodo de inicio y se mueve a los nodos vecinos, actualizando las distancias más cortas. Este proceso continúa hasta que todos los nodos han sido visitados.

2. Desarrollo de la practica

2.1. Definición del problema

El problema de caminos mínimos en grafos ponderados se refiere a la búsqueda del camino más corto entre dos nodos específicos en un grafo donde cada arista tiene un peso o costo asociado. Este problema tiene aplicaciones en diversos campos, como redes de comunicación, logística, planificación de rutas, diseño de circuitos, entre otros.

2.2. Descripcion del algoritmo

Para la resolucion de este problema nosotros usaremos el algoritmo de dijkstra. Este algoritmo es de tipo voraz (greedy) y se aplica principalmente a grafos dirigidos y no dirigidos. El algoritmo funciona de manera voraz al seleccionar en cada paso el nodo con la distancia mínima conocida en ese momento. A medida que avanza, actualiza las distancias a los nodos vecinos si encuentra caminos más cortos. La garantía de corrección del algoritmo se basa en la propiedad de optimalidad de los subcaminos más cortos: si el camino más corto desde el nodo inicial hasta un nodo intermedio es conocido, el camino más corto desde el nodo inicial hasta cualquier nodo conectado a ese nodo intermedio también es conocido.

2.3. Implementacion del algoritmo

Para desarrollar este código, establecimos una variable llamada "cola de prioridad". Su función es almacenar todos los nodos que aún no han sido visitados. Al recorrer el grafo, esta cola compara las distancias actuales con las de los nodos vecinos. Luego, selecciona la distancia mínima y la imprime. Finalmente, cuando la cola de prioridad está vacía, el algoritmo concluye e imprime los caminos más cortos.

Ahora, en cuanto al código, primero creamos una clase que contiene tres métodos. El primero crea un diccionario llamado "vertices", donde se generan tuplas representando las conexiones desde un vértice dado a otro, y se almacenan sus respectivos pesos. El segundo método se encarga de agregar todos los vértices que hemos definido en el grafo, inicializándolos con una lista vacía de conexiones. Finalmente, el tercer método agrega las aristas definidas en el grafo, creando tuplas que indican desde qué vértice hasta qué vértice va la arista y su peso.

Listing 1: Código de la clase y sus metodos

```
#Definimos una clase que representara un grafo
class Graph:
    #Tiene un diccionario llamado vertices que manda cada vertice a una lista de tuplasque representa las conexiones desde ese vertice a otro y sus pesos
    def __init__(self):
        self.vertices = {}
    #Este metodo agregara vertices al grafo
    def add_vertex(self, vertex):
        #Y los inicializara con una lista vacia de conexiones
        self.vertices[vertex] = []
    #Este meyodo se encarga de agregar aritas al grafo
    def add_edge(self, from_vertex, to_vertex, weight):
        #Agrega tuplas que indica de que vertice a que vertice va la arista y su peso
        self.vertices[from_vertex].append((to_vertex, weight))
```

Luego de haber creado la clase encargada de la construcción del grafo, desarrollamos una función que ejecutará todo el proceso de Dijkstra. Esta función recibe como parámetros un grafo y un vértice de inicio. Posee un diccionario que recorre cada vértice, registrando la distancia mínima desde el inicio. Este diccionario también se inicializa en 0. Además, cuenta con una lista llamada 'cola de prioridad', encargada de almacenar todos los vértices que aún no han sido recorridos.

Dentro de esta función, recorremos el grafo mientras nuestra cola de prioridad no esté vacía. La condición de no estar vacía indica que aún hay vértices por explorar. A medida que avanza, se va visitando cada vértice. Si la distancia nueva es menor que la anterior, procedemos al siguiente bucle, donde ahora las distancias se comparan entre los vértices vecinos al actual. Si se encuentra una menor distancia entre un vértice y otro, se guarda en la cola de prioridad.

En resumen, esta función implementa el algoritmo de Dijkstra para encontrar los caminos más cortos en un grafo ponderado con pesos no negativos. Después de que la cola de prioridad esté vacía y todas las distancias hayan sido actualizadas, la función devuelve el diccionario 'distances' que contiene las distancias mínimas desde el vértice de inicio a todos los demás vértices del grafo.

Listing 2: Función dijkstra

```
#Esta funcion toma un grafo y un vertice de inicio como entrada
def dijkstra(graph, start_vertex):
    #Inicializa un diccionario que recorrera cada vertice a la distamcia minima desde e del inicio
    distances = {vertex: float('infinity') for vertex in graph.vertices}
    #Todas las distancias se inicializan en infinito excepto la del inicio
    distances[start_vertex] = 0
    #Inicializamos una cola de prioridad como una lista de tuplasdonde cada tupla tiene la distancia actual y el vertice
```

```
        correspondiente
    priority_queue = [(0, start_vertex)]

    #En el bucle principla mientras la variable anterior este vacia, extraemos el vertice con la distancia minima actual desde la variable anterior
    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)
        #Si la distancia actual es mayor que la distancia conocida desde el v rtice de inicio, se ignora y se pasa al siguiente ciclo del bucle
        if current_distance > distances[current_vertex]:continue
        #Itera sobre los vertices vecinos qu tenga el actual y calcula la distancia acumulada desde el vertice del inicio
        for neighbor, weight in graph.vertices[current_vertex]:
            distance = current_distance + weight
            #Si la nueva distancia es menor que la distancia que llevamso desde el vertice de inciio hasta el vecino
            if distance < distances[neighbor]:
                #Actualizaremos la distancia conocida
                distances[neighbor] = distance
                #Y agregaremos el vecino a la cola de prioridad con la nueva distancia
                heapq.heappush(priority_queue, (distance, neighbor))

    #Luego de que la cola de prioridad este vacia y todas las distancias hayan sido actualizadas, se devolvera el diccionario distancias
    return distances
```

2.4. Pruebas del funcionamiento

Y finalmente para poder probar su funcionamiento, se crearon cinco grafos, de los cuales muestra el peso minimo para ir de cierto vertice a cualquier otro.

Listing 3: Pruebas del funcionamiento

```
#Caso de prueba 1
grafo = Graph()
grafo.add_vertex("A")
grafo.add_vertex("B")
grafo.add_vertex("C")
grafo.add_vertex("D")
grafo.add_edge("A", "B", 2)
grafo.add_edge("A", "C", 1)
grafo.add_edge("B", "D", 3)
grafo.add_edge("C", "D", 4)

#Luego llamamos a la funcion dijkstra con el grafo y el vertice de inicio a
```

```python
start_vertex = "A"
resultado = dijkstra(grafo, start_vertex)
#Al final imprimimos el resultado
resultado, tiempo = Tiempo_ejecucion(dijkstra, grafo, start_vertex)
print("\n-> Ejemplo #1: ")
print(f"   Caminos m nimos desde {start_vertex}: {resultado}")
print(f"   Tiempo de ejecuci n: {tiempo} segundos")

#Caso de prueba 2
grafo2 = Graph()
grafo2.add_vertex("M")
grafo2.add_vertex("N")
grafo2.add_vertex("O")
grafo2.add_vertex("P")
grafo2.add_edge("M", "N", 5)
grafo2.add_edge("M", "O", 6)
grafo2.add_edge("M", "P", 3)
grafo2.add_edge("N", "P", 12)
grafo2.add_edge("O", "P", 9)

start_vertex2 = "M"
resultado2, tiempo2 = Tiempo_ejecucion(dijkstra, grafo2, start_vertex2)
print("\n-> Ejemplo #3: ")
print(f"   Caminos m nimos desde {start_vertex2}: {resultado2}")
print(f"   Tiempo de ejecuci n: {tiempo2} segundos")

# Caso de prueba 3
grafo3 = Graph()
grafo3.add_vertex("A")
grafo3.add_vertex("B")
grafo3.add_vertex("C")
grafo3.add_vertex("D")
grafo3.add_vertex("E")
grafo3.add_vertex("F")
grafo3.add_vertex("G")
grafo3.add_edge("A", "B", 2)
grafo3.add_edge("B", "C", 1)
grafo3.add_edge("C", "D", 3)
grafo3.add_edge("A", "E", 4)
grafo3.add_edge("E", "F", 5)
grafo3.add_edge("F", "G", 2)
grafo3.add_edge("C", "G", 2)

start_vertex3 = "A"
resultado3, tiempo3 = Tiempo_ejecucion(dijkstra, grafo3, start_vertex3)
print("\n-> Ejemplo #4: ")
print(f"   Caminos m nimos desde {start_vertex3}: {resultado3}")
print(f"   Tiempo de ejecuci n: {tiempo3} segundos")

#Caso de prueba 4
```

```
grafo4 = Graph()
grafo4.add_vertex("X")
grafo4.add_vertex("Y")
grafo4.add_vertex("Z")
grafo4.add_vertex("W")
grafo4.add_vertex("V")
grafo4.add_edge("X", "Y", 4)
grafo4.add_edge("Y", "Z", 1)
grafo4.add_edge("Y", "V", 3)
grafo4.add_edge("X", "W", 2)
grafo4.add_edge("W", "V", 5)

start_vertex4 = "X"
resultado4, tiempo4 = Tiempo_ejecucion(dijkstra, grafo4, start_vertex4)
print("\n-> Ejemplo #5: ")
print(f"   Caminos m nimos desde {start_vertex4}: {resultado4}")
print(f"   Tiempo de ejecuci n: {tiempo4} segundos")

#Caso de prueba 5
grafo5 = Graph()
grafo5.add_vertex("1")
grafo5.add_vertex("2")
grafo5.add_vertex("3")
grafo5.add_vertex("4")
grafo5.add_vertex("5")
grafo5.add_edge("1", "2", 2)
grafo5.add_edge("2", "3", 1)
grafo5.add_edge("2", "5", 1)
grafo5.add_edge("1", "4", 3)
grafo5.add_edge("4", "5", 4)

start_vertex5 = "1"
resultado5, tiempo5 = Tiempo_ejecucion(dijkstra, grafo5, start_vertex5)
print("\n-> Ejemplo #6: ")
print(f"   Caminos m nimos desde {start_vertex5}: {resultado5}")
print(f"   Tiempo de ejecuci n: {tiempo5} segundos")
```

3. Análisis del algoritmo

3.1. Medición del tiempo de ejecución

Comtamos de igual modo con una función encargada de medir el tiempo de ejecucion del proceso

Listing 4: Funcion para medir el tiempo de ejecucion

```
def Tiempo_ejecucion(funcion, *args):
    inicio = time.time()
```

```
    resultado = funcion(*args)
    fin = time.time()
    tiempo_ejecucion = fin - inicio
    return resultado, tiempo_ejecucion
```

3.2. Cálculo de la complejidad

La complejidad en notación big O del algoritmo de Dijkstra es comúnmente expresada como $O((V + E)logV)$, donde V es el número de vértices y E es el número de aristas en el grafo.

4. Codigo completo

Listing 5: Código completo

```
#Libreria para crear arboles
import heapq
#Libreria para el tiempo
import time

#Definimos una clase que representara un grafo
class Graph:
    #Tiene un diccionario llamado vertices que manda cada vertice a
       una lista de tuplasque representa las conexiones desde ese
       vertice a otro y sus pesos
    def __init__(self):
        self.vertices = {}
    #Este metodo agregara vertices al grafo
    def add_vertex(self, vertex):
        #Y los inicializara con una lista vacia de conexiones
        self.vertices[vertex] = []
    #Este meyodo se encarga de agregar aritas al grafo
    def add_edge(self, from_vertex, to_vertex, weight):
        #Agrega tuplas que indica de que vertice a que vertice va la
           arista y su peso
        self.vertices[from_vertex].append((to_vertex, weight))

#Esta funcion toma un grafo y un vertice de inicio como entrada
def dijkstra(graph, start_vertex):
    #Inicializa un diccionario que recorrera cada vertice a la
       distamcia minima desde e del inicio
    distances = {vertex: float('infinity') for vertex in graph.
       vertices}
    #Todas las distancias se inicializan en infinito excepto la del
       inicio
    distances[start_vertex] = 0
    #Inicializamos una cola de prioridad como una lista de tuplasdonde
        cada tupla tiene la distancia actual y el vertice
       correspondiente
```

```
    priority_queue = [(0, start_vertex)]

    #En el bucle principla mientras la variable anterior este vacia, extraemos el vertice con la distancia minima actual desde la variable anterior
    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)
        #Si la distancia actual es mayor que la distancia conocida desde el v rtice de inicio, se ignora y se pasa al siguiente ciclo del bucle
        if current_distance > distances[current_vertex]:continue
        #Itera sobre los vertices vecinos qu tenga el actual y calcula la distancia acumulada desde el vertice del inicio
        for neighbor, weight in graph.vertices[current_vertex]:
            distance = current_distance + weight
            #Si la nueva distancia es menor que la distancia que llevamso desde el vertice de inciio hasta el vecino
            if distance < distances[neighbor]:
                #Actualizaremos la distancia conocida
                distances[neighbor] = distance
                #Y agregaremos el vecino a la cola de prioridad con la nueva distancia
                heapq.heappush(priority_queue, (distance, neighbor))

    #Luego de que la cola de prioridad este vacia y todas las distancias hayan sido actualizadas, se devolvera el diccionario distancias
    return distances

def Tiempo_ejecucion(funcion, *args):
    inicio = time.time()
    resultado = funcion(*args)
    fin = time.time()
    tiempo_ejecucion = fin - inicio
    return resultado, tiempo_ejecucion

print("\nAlgoritmo de Dijkstra")

#Caso de prueba 1
grafo = Graph()
grafo.add_vertex("A")
grafo.add_vertex("B")
grafo.add_vertex("C")
grafo.add_vertex("D")
grafo.add_edge("A", "B", 2)
grafo.add_edge("A", "C", 1)
grafo.add_edge("B", "D", 3)
grafo.add_edge("C", "D", 4)

#Luego llamamos a la funcion dijkstra con el grafo y el vertice de inicio a
```

```python
start_vertex = "A"
resultado = dijkstra(grafo, start_vertex)
#Al final imprimimos el resultado
resultado, tiempo = Tiempo_ejecucion(dijkstra, grafo, start_vertex)
print("\n-> Ejemplo #1: ")
print(f"   Caminos m nimos desde {start_vertex}: {resultado}")
print(f"   Tiempo de ejecuci n: {tiempo} segundos")

#Caso de prueba 2
grafo2 = Graph()
grafo2.add_vertex("M")
grafo2.add_vertex("N")
grafo2.add_vertex("O")
grafo2.add_vertex("P")
grafo2.add_edge("M", "N", 5)
grafo2.add_edge("M", "O", 6)
grafo2.add_edge("M", "P", 3)
grafo2.add_edge("N", "P", 12)
grafo2.add_edge("O", "P", 9)

start_vertex2 = "M"
resultado2, tiempo2 = Tiempo_ejecucion(dijkstra, grafo2, start_vertex2)
print("\n-> Ejemplo #3: ")
print(f"   Caminos m nimos desde {start_vertex2}: {resultado2}")
print(f"   Tiempo de ejecuci n: {tiempo2} segundos")

# Caso de prueba 3
grafo3 = Graph()
grafo3.add_vertex("A")
grafo3.add_vertex("B")
grafo3.add_vertex("C")
grafo3.add_vertex("D")
grafo3.add_vertex("E")
grafo3.add_vertex("F")
grafo3.add_vertex("G")
grafo3.add_edge("A", "B", 2)
grafo3.add_edge("B", "C", 1)
grafo3.add_edge("C", "D", 3)
grafo3.add_edge("A", "E", 4)
grafo3.add_edge("E", "F", 5)
grafo3.add_edge("F", "G", 2)
grafo3.add_edge("C", "G", 2)

start_vertex3 = "A"
resultado3, tiempo3 = Tiempo_ejecucion(dijkstra, grafo3, start_vertex3)
print("\n-> Ejemplo #4: ")
print(f"   Caminos m nimos desde {start_vertex3}: {resultado3}")
print(f"   Tiempo de ejecuci n: {tiempo3} segundos")

#Caso de prueba 4
```

```python
grafo4 = Graph()
grafo4.add_vertex("X")
grafo4.add_vertex("Y")
grafo4.add_vertex("Z")
grafo4.add_vertex("W")
grafo4.add_vertex("V")
grafo4.add_edge("X", "Y", 4)
grafo4.add_edge("Y", "Z", 1)
grafo4.add_edge("Y", "V", 3)
grafo4.add_edge("X", "W", 2)
grafo4.add_edge("W", "V", 5)

start_vertex4 = "X"
resultado4, tiempo4 = Tiempo_ejecucion(dijkstra, grafo4, start_vertex4)
print("\n-> Ejemplo #5: ")
print(f"   Caminos minimos desde {start_vertex4}: {resultado4}")
print(f"   Tiempo de ejecuci n: {tiempo4} segundos")

#Caso de prueba 5
grafo5 = Graph()
grafo5.add_vertex("1")
grafo5.add_vertex("2")
grafo5.add_vertex("3")
grafo5.add_vertex("4")
grafo5.add_vertex("5")
grafo5.add_edge("1", "2", 2)
grafo5.add_edge("2", "3", 1)
grafo5.add_edge("2", "5", 1)
grafo5.add_edge("1", "4", 3)
grafo5.add_edge("4", "5", 4)

start_vertex5 = "1"
resultado5, tiempo5 = Tiempo_ejecucion(dijkstra, grafo5, start_vertex5)
print("\n-> Ejemplo #6: ")
print(f"   Caminos minimos desde {start_vertex5}: {resultado5}")
print(f"   Tiempo de ejecucion: {tiempo5} segundos")
```

5. Resultados (Capturas)

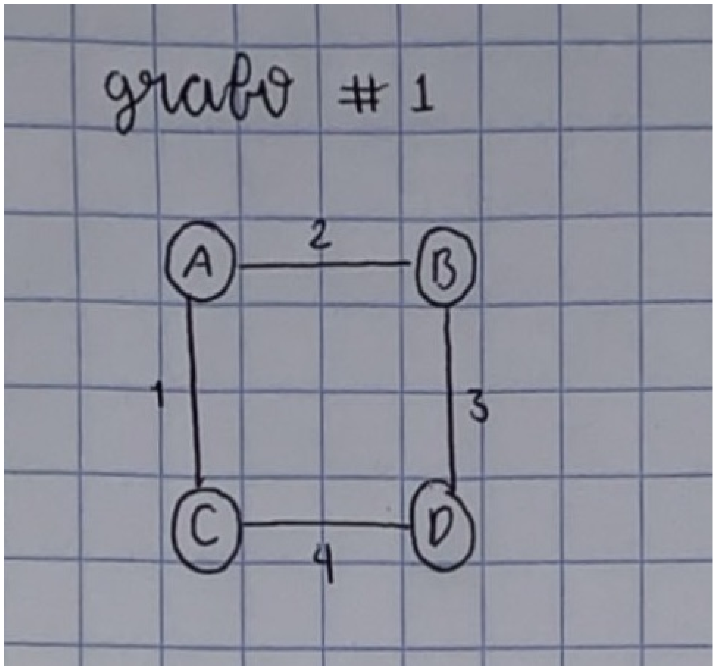

Figura 1: Primer grafo

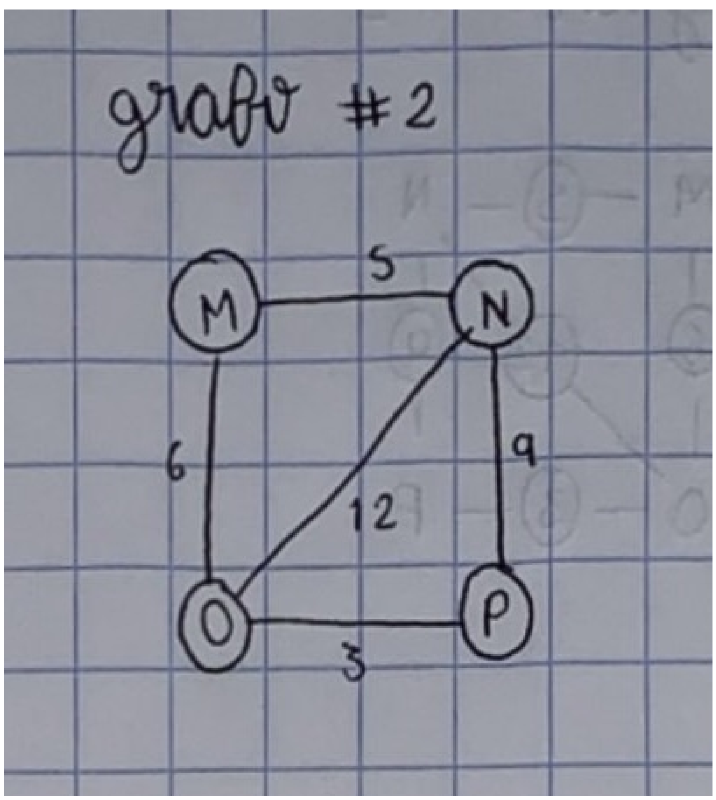

Figura 2: Segundo grafo

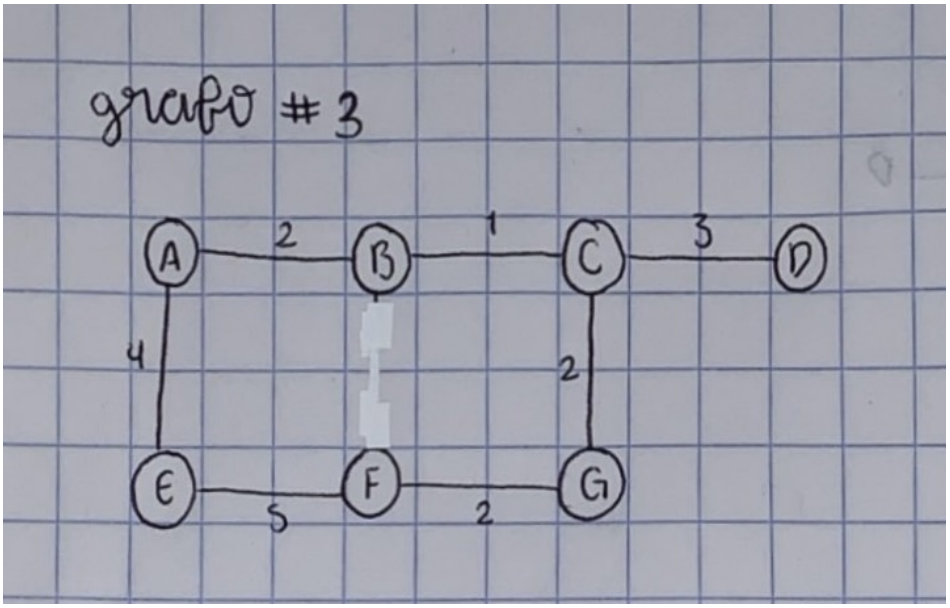

Figura 3: Tercer grafo

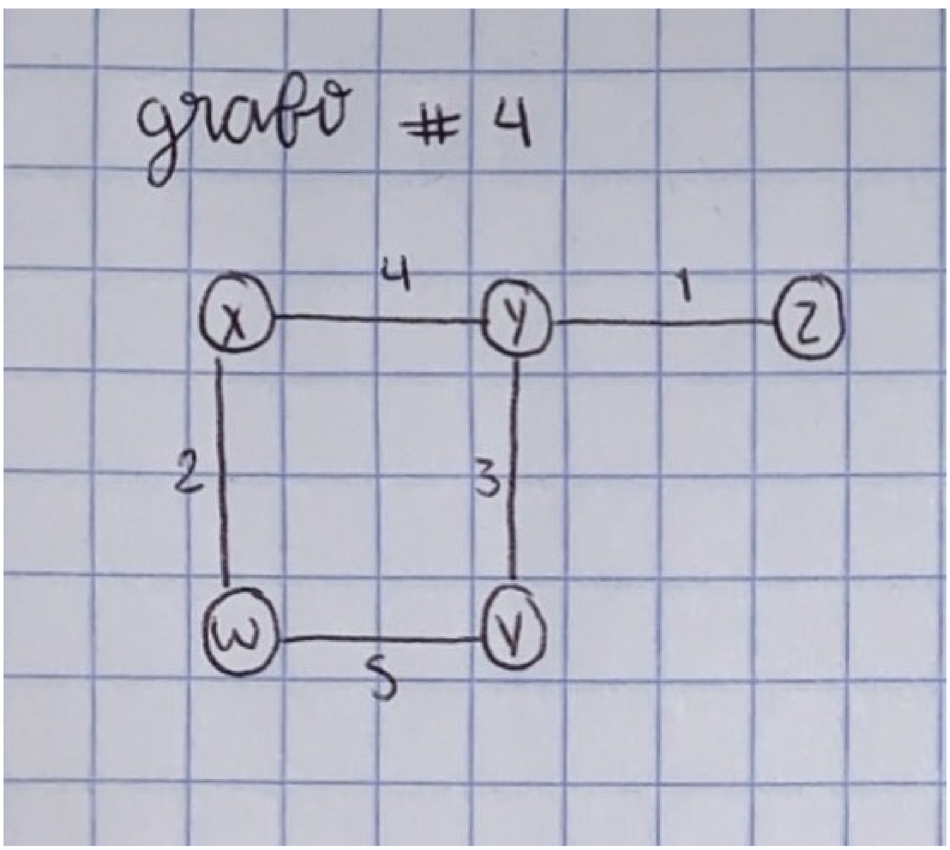

Figura 4: Cuarto grafo

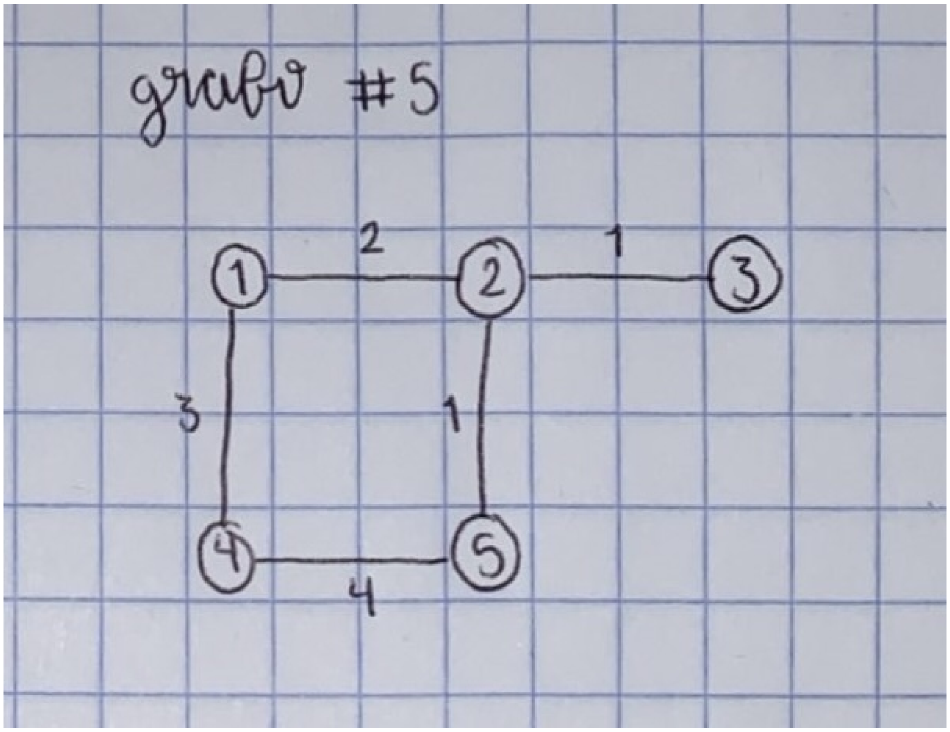

Figura 5: Quinto grafo

```
Algoritmo de Dijkstra

-> Ejemplo #1:
   Caminos mínimos desde A: {'A': 0, 'B': 2, 'C': 1, 'D': 5}
   Tiempo de ejecución: 3.0994415283203125e-06 segundos

-> Ejemplo #3:
   Caminos mínimos desde M: {'M': 0, 'N': 5, 'O': 6, 'P': 3}
   Tiempo de ejecución: 4.0531158447265625e-06 segundos

-> Ejemplo #4:
   Caminos mínimos desde A: {'A': 0, 'B': 2, 'C': 3, 'D': 6, 'E': 4, 'F': 9, 'G': 5}
   Tiempo de ejecución: 5.245208740234375e-06 segundos

-> Ejemplo #5:
   Caminos mínimos desde X: {'X': 0, 'Y': 4, 'Z': 5, 'W': 2, 'V': 7}
   Tiempo de ejecución: 4.0531158447265625e-06 segundos

-> Ejemplo #6:
   Caminos mínimos desde 1: {'1': 0, '2': 2, '3': 3, '4': 3, '5': 3}
   Tiempo de ejecución: 0.00010704994201660156 segundos
○ naomigarcia@Naomis-MacBook-Air Analisis y diseño de algoritmos %
```

Figura 6: Resultados del código

6. Conclusiones

El algoritmo de Dijkstra es como un planificador de rutas que te ayuda a encontrar el camino más corto entre puntos en un mapa. Utiliza una estrategia astuta para explorar las opciones de manera eficiente, asegurándose de elegir siempre el camino más corto. Gracias a su habilidad para manejar diferentes caminos y distancias, es ideal para encontrar la ruta más rápida entre lugares en un grafo, como una red de carreteras. Y gracias a la práctica donde implementamos este algoritmo, pudimos tener una vista mejorada sobre el manejo de arboles y nodos en python, asi como su manipulación y manjeo dentro del entorno de programación de python.

7. Referencias

- `https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/`
- `https://www.codingame.com/playgrounds/7656/los-caminos-mas-cortos-con-el-algoritmo-de-dijkstra/el-algoritmo-de-dijkstra`
- `http://atlas.uned.es/algoritmos/voraces/dijkstra.html`