Reporte 03

Análisis y Diseño de Algoritmos

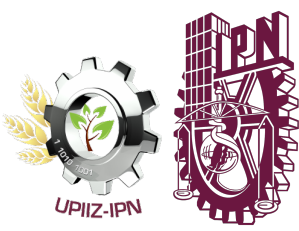
Grupo: 3CM1

Implementación y Evaluación del Algoritmo de Dijkstra

3202

Docente:
M. en C. Erika Sánchez-Femat
Alumna:
Melanie Aileen Roman Espitia

Fecha de Entrega: viernes 01 de diciembre de 2023 Zacatecas, México



Índice

1. Introducción **1**

2. Desarrollo **1**
2.1. Descripción del Algoritmo de Dijkstra 1
2.2. Implementación en Python 3
2.3. Análisis del Algoritmo 4

3. Conclusiones **8**

4. Referencias **9**

1. Introducción

La teoría de grafos y sus algoritmos asociados desempeñan un papel fundamental en la resolución de problemas relacionados con redes y rutas en diversos campos, desde logística hasta redes de computadoras. Uno de los algoritmos más destacados en este contexto es el algoritmo de Dijkstra, diseñado para encontrar los caminos más cortos entre dos puntos en un grafo ponderado. Este algoritmo, desarrollado por el científico de la computación Edsger Dijkstra en 1956, ha demostrado ser esencial en la optimización de rutas y la toma de decisiones basada en distancias mínimas.
Este informe se sumerge en la implementación y aplicación práctica del algoritmo de Dijkstra, centrándose en su representación mediante un grafo ponderado dirigido. La estructura del grafo se modela a través de una clase en Python, la cual se diseña para facilitar la adición de vértices y aristas con pesos. Además, se explora la implementación del algoritmo de Dijkstra como una función que encuentra los caminos mínimos desde un vértice de inicio a todos los demás vértices en el grafo.
Para poner a prueba la robustez y eficacia del algoritmo, se lleva a cabo una serie de pruebas utilizando grafos de ejemplo, en los cuales se añaden vértices y aristas con pesos específicos. Posteriormente, se ejecuta el algoritmo de Dijkstra sobre estos grafos para visualizar cómo determina los caminos mínimos desde un vértice de inicio dado.
Además, con el objetivo de evaluar el rendimiento del algoritmo en situaciones diversas, se introducirá la generación de grafos aleatorios. Estos grafos proporcionan un contexto más dinámico y permiten analizar la escalabilidad y eficiencia del algoritmo de Dijkstra en entornos variables.
Finalmente, se analizará la complejidad temporal del algoritmo, destacando cómo el número de vértices y aristas influye en el tiempo de ejecución. Este análisis contribuirá a una comprensión más profunda de la eficiencia del algoritmo de Dijkstra y su aplicabilidad en escenarios del mundo real.

2. Desarrollo

2.1. Descripción del Algoritmo de Dijkstra

1. Inicialización:
- Crear un conjunto de vértices no visitados y establecer la distancia inicial al vértice de inicio como 0, mientras que todas las demás distancias se inicializan como infinito.
- Marcar todos los vértices como no visitados.

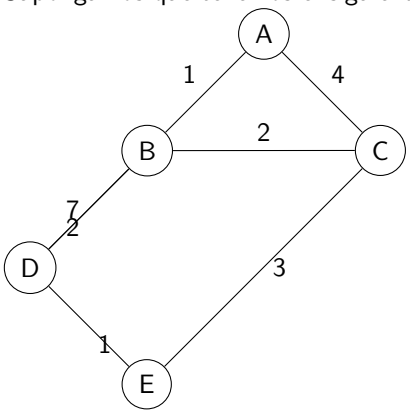
2. Exploración del Vértice de Inicio:
- Iniciar desde el vértice de inicio y actualizar las distancias a sus vecinos directos.
- Marcar el vértice de inicio como visitado.

3. Iteración hasta que todos los vértices estén visitados:
- Escoger el vértice no visitado con la distancia más pequeña.
- Actualizar las distancias a los vecinos del vértice escogido si se encuentra un camino más corto.
- Marcar el vértice escogido como visitado.

- Repetir el Paso 3 hasta que todos los vértices estén visitados.

4. Resultados:
- Al finalizar, se obtienen las distancias mínimas desde el vértice de inicio a todos los demás vértices.

EJEMPLO:
Supongamos que tenemos el siguiente grafo ponderado dirigido:



Vamos a calcular los caminos mínimos desde el vértice "A" utilizando el algoritmo de Dijkstra.

Paso 1: Inicialización
Inicializamos todas las distancias con infinito, excepto la distancia a sí mismo, que es 0.
Distancias: A: 0, B: ∞, C: ∞, D: ∞, E: ∞

Paso 2: Explorar vecinos de A
Vecinos de A: B (1), C (4)
Actualizamos las distancias a los vecinos si encontramos caminos más cortos.
Distancias actualizadas: A: 0, B: 1, C: 4, D: ∞, E: ∞

Paso 3: Explorar vecinos de B
Vecinos de B: A (1), C (2), D (7)
Actualizamos las distancias
Distancias actualizadas: A: 0, B: 1, C: 3, D: ∞, E: ∞

Paso 4: Explorar vecinos de C
Vecinos de C: A (4), B (2), E (3)
Actualizamos las distancias
Distancias actualizadas: A: 0, B: 1, C: 3, D: ∞, E: 6

Paso 5: Explorar vecinos de D
Vecinos de D: B (7), E (1)
Actualizamos las distancias
Distancias actualizadas: A: 0, B: 1, C: 3, D: 8, E: 4

Paso 6: Explorar vecinos de E

Vecinos de E: C (3)
Actualizamos las distancias
Distancias actualizadas: A: 0, B: 1, C: 3, D: 8, E: 4
En este punto, hemos encontrado los caminos mínimos desde A a todos los demás vértices. Los resultados son:
Caminos mínimos desde A: 'A': 0, 'B': 1, 'C': 3, 'D': 8, 'E': 4
Este es un ejemplo simple de cómo aplicar el algoritmo de Dijkstra a mano en un grafo pequeño. La clave es ir explorando los vecinos y actualizando las distancias según sea necesario.

2.2. Implementación en Python

```python
import heapq
# Creamos una clase para representar nuestro Grafo
class Graph:
    def __init__(self):
        # Inicializamos un diccionario para almacenar los vértices y sus conexiones
        self.vertices = {}

    # Método para agregar un nuevo vértice al grafo
    def add_vertex(self, vertex):
        # Verificamos si el vértice no existe ya antes de agregarlo
        if vertex not in self.vertices:
            # Agregamos el vértice al diccionario con una lista vacía de conexiones
            self.vertices[vertex] = []

    # Método para agregar una arista (con peso) entre dos vértices
    def add_edge(self, from_vertex, to_vertex, weight):
        # Añadimos la conexión desde un vértice de inicio a otro con su respectivo peso
        self.vertices[from_vertex].append((to_vertex, weight))

# Función para el algoritmo de Dijkstra
def dijkstra(graph, start_vertex):
    # Creamos un diccionario para almacenar las distancias mínimas
    distances = {vertex: float('infinity') for vertex in graph.vertices}
    # Establecemos la distancia mínima desde el vértice inicial a él mismo como 0
    distances[start_vertex] = 0
    # Creamos una cola de prioridad con la distancia actual y el vértice actual
    priority_queue = [(0, start_vertex)]

    # Mientras haya elementos en la cola de prioridad
    while priority_queue:
        # Obtenemos el vértice actual y su distancia
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # Si la distancia actual es mayor que la registrada, no hacemos nada
        if current_distance > distances[current_vertex]:
```

```
            continue

        # Iteramos sobre los vecinos del vértice actual
        for neighbor, weight in graph.vertices[current_vertex]:
            # Calculamos la nueva distancia
            distance = current_distance + weight

            # Si la nueva distancia es menor que la registrada, actualizamos la distancia
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                # Agregamos el vecino y su nueva distancia a la cola de prioridad
                heapq.heappush(priority_queue, (distance, neighbor))

    # Devolvemos las distancias mínimas
    return distances

# Ejemplo de uso
grafo = Graph()
grafo.add_vertex("A")
grafo.add_vertex("B")
grafo.add_vertex("C")
grafo.add_edge("A", "B", 3)
grafo.add_edge("A", "C", 5)
grafo.add_edge("B", "C", 2)

# Definimos el vértice de inicio
start_vertex = "A"
# Llamamos a la función dijkstra para obtener las distancias mínimas
resultado = dijkstra(grafo, start_vertex)
# Imprimimos los resultados
print(f"Caminos mínimos desde {start_vertex}: {resultado}")
```

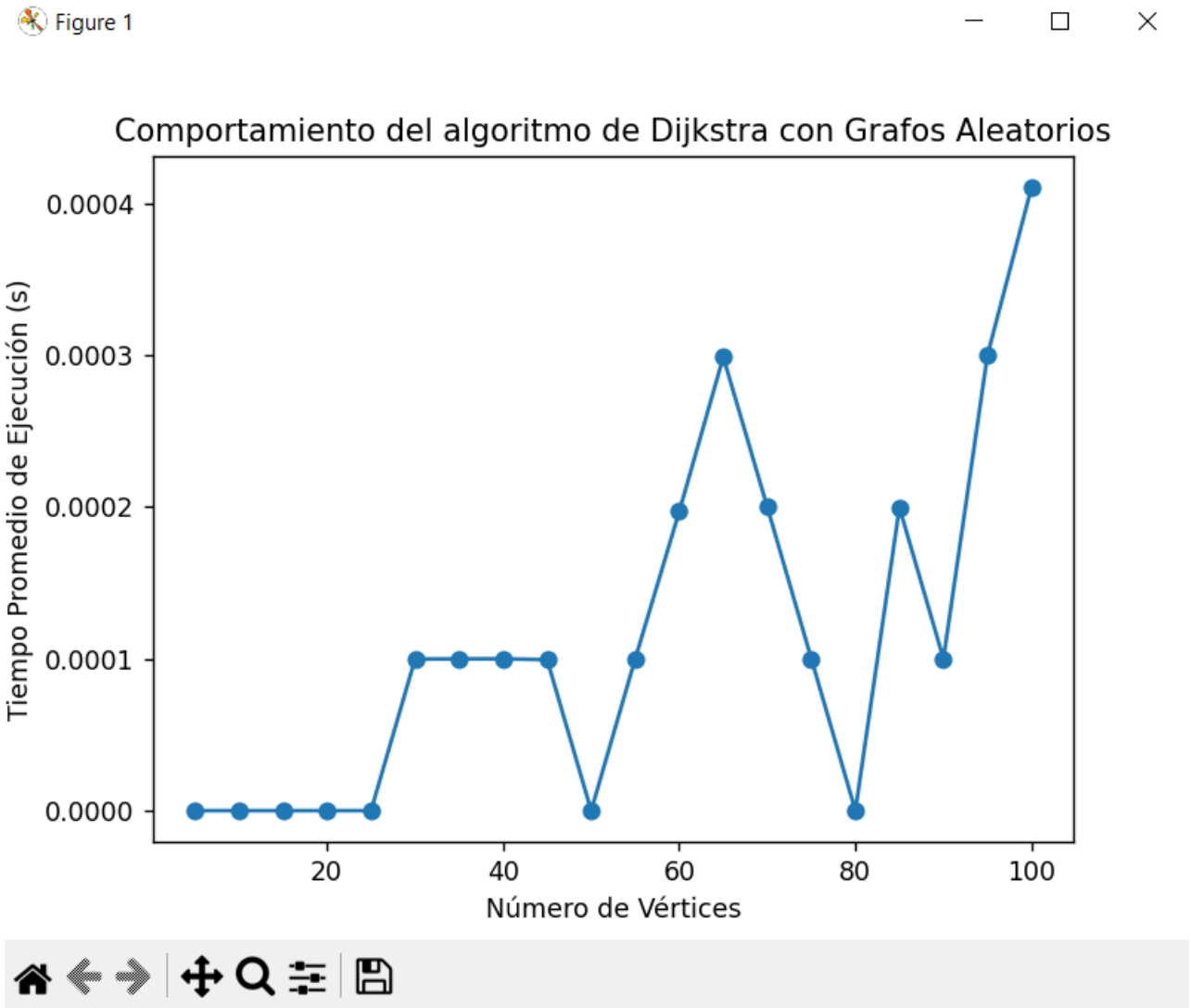
2.3. Análisis del Algoritmo

MEDICIÓN DEL TIEMPO:

```
import heapq  # Importamos el módulo heapq para implementar la cola de prioridad
import time  # Importamos el módulo time para medir el tiempo de ejecución
import random  # Importamos el módulo random para generar números aleatorios
import matplotlib.pyplot as plt  # Importamos matplotlib para graficar

# Creamos una clase para representar nuestro Grafo
class Graph:
    def __init__(self):
        # Inicializamos un diccionario para almacenar los vértices y sus conexiones
        self.vertices = {}
```

```
    # Método para agregar un nuevo vértice al grafo
    def add_vertex(self, vertex):
        # Verificamos si el vértice no existe ya antes de agregarlo
        if vertex not in self.vertices:
            # Agregamos el vértice al diccionario con una lista vacía de conexiones
            self.vertices[vertex] = []

    # Método para agregar una arista (con peso) entre dos vértices
    def add_edge(self, from_vertex, to_vertex, weight):
        # Añadimos la conexión desde un vértice de inicio a otro con su respectivo peso
        self.vertices[from_vertex].append((to_vertex, weight))

# Función para el algoritmo de Dijkstra
def dijkstra(graph, start_vertex):
    # Creamos un diccionario para almacenar las distancias mínimas
    distances = {vertex: float('infinity') for vertex in graph.vertices}
    # Establecemos la distancia mínima desde el vértice inicial a él mismo como 0
    distances[start_vertex] = 0
    # Creamos una cola de prioridad con la distancia actual y el vértice actual
    priority_queue = [(0, start_vertex)]

    # Mientras haya elementos en la cola de prioridad
    while priority_queue:
        # Obtenemos el vértice actual y su distancia
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # Si la distancia actual es mayor que la registrada, no hacemos nada
        if current_distance > distances[current_vertex]:
            continue

        # Iteramos sobre los vecinos del vértice actual
        for neighbor, weight in graph.vertices[current_vertex]:
            # Calculamos la nueva distancia
            distance = current_distance + weight

            # Si la nueva distancia es menor que la registrada, actualizamos la distancia
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                # Agregamos el vecino y su nueva distancia a la cola de prioridad
                heapq.heappush(priority_queue, (distance, neighbor))

    # Devolvemos las distancias mínimas
    return distances

# Función para generar un grafo aleatorio
```

```python
def generate_random_graph(num_vertices):
    G = Graph()

    for i in range(1, num_vertices + 1):
        G.add_vertex(str(i))

    for i in range(1, num_vertices + 1):
        for j in range(i + 1, num_vertices + 1):
            if random.choice([True, False]):
                weight = random.randint(1, 10)
                G.add_edge(str(i), str(j), weight)

    return G

# Realizar experimentos
max_vertices = 100
execution_times = []
num_vertices_list = []

# Iteramos sobre diferentes tamaños de grafo
for i in range(5, max_vertices + 1, 5):
    total_time = 0
    # Realizamos 10 experimentos por tamaño de grafo
    for _ in range(10):
        graph = generate_random_graph(i)

        start_time = time.time()
        dijkstra(graph, "1")
        end_time = time.time()

        total_time += end_time - start_time

    # Calculamos el tiempo promedio de ejecución
    avg_time = total_time / 10
    execution_times.append(avg_time)
    num_vertices_list.append(i)

# Graficar los resultados
plt.plot(num_vertices_list, execution_times, marker='o')
plt.xlabel('Número de Vértices')
plt.ylabel('Tiempo Promedio de Ejecución (s)')
plt.title('Comportamiento del algoritmo de Dijkstra con Grafos Aleatorios')
plt.show()
```



CÁLCULO DE COMPLEJIDAD

Vamos a analizar la complejidad temporal (Big O) del algoritmo de Dijkstra en el contexto de la implementación que proporcioné.
La complejidad del algoritmo de Dijkstra depende de cómo se implementa el paso de encontrar el vértice con la menor distancia en la cola de prioridad. En este caso, estamos utilizando una cola de prioridad implementada con un heap binario, que es eficiente para estas operaciones.

- **Agregar un vértice a la cola de prioridad:** La operación `heappush` tiene una complejidad de $O(\log V)$, donde V es el número de vértices en el grafo.
- **Eliminar el vértice con la menor distancia de la cola de prioridad:** La operación `heappop`

también tiene una complejidad de $O(\log V)$.

- **Actualizar la distancia de un vértice en la cola de prioridad:** En el peor de los casos, esta operación también tiene una complejidad de $O(\log V)$.

En cada iteración del bucle principal del algoritmo de Dijkstra, realizamos estas operaciones para cada vértice y sus vecinos, lo que da como resultado una complejidad total de $O((V + E) \log V)$, donde V es el número de vértices y E es el número de aristas en el grafo.
En cuanto a la generación del grafo aleatorio, si V es el número de vértices y E es el número de aristas, la complejidad sería $O(V + E)$ porque estamos agregando V vértices y E aristas al grafo.
Por lo tanto, la complejidad total de la función `measure_time`, que ejecuta el algoritmo de Dijkstra varias veces sobre un grafo aleatorio, sería aproximadamente $O(\text{num\_iteraciones} \cdot (V + E) \log V)$, donde num_iteraciones es el número de veces que se ejecuta el algoritmo.
Es importante tener en cuenta que esta es una evaluación general y que la complejidad exacta podría depender de detalles específicos de implementación y del comportamiento específico de los grafos con los que se trabaja.

3. Conclusiones

1. **Eficiencia del Algoritmo:**
   - El algoritmo de Dijkstra es eficiente para encontrar el camino más corto desde un vértice de inicio a todos los demás vértices en un grafo ponderado dirigido con pesos no negativos.
   - La implementación con una cola de prioridad basada en un heap binario contribuye significativamente a la eficiencia del algoritmo.
2. **Complejidad Temporal:**
   - La complejidad del algoritmo de Dijkstra depende en gran medida de la implementación de la cola de prioridad.
   - Utilizando un heap binario, las operaciones de agregar, eliminar y actualizar tienen complejidad $O(\log V)$, donde V es el número de vértices en el grafo.
   - La complejidad total del algoritmo es aproximadamente $O((V + E) \log V)$, donde E es el número de aristas.
3. **Generación del Grafo Aleatorio:**
   - La complejidad de generar un grafo aleatorio con V vértices y E aristas es $O(V + E)$.
   - Esto implica que la generación del grafo contribuye linealmente a la complejidad total del experimento.
4. **Importancia del Contexto y Detalles de Implementación:**
   - La complejidad exacta puede depender de la implementación específica y del comportamiento de los grafos involucrados.
   - Se debe considerar el contexto de la aplicación y la naturaleza de los grafos para evaluar el rendimiento del algoritmo.

5. **Consideraciones Prácticas:**

   - A pesar de la complejidad teórica, el algoritmo de Dijkstra es ampliamente utilizado en la práctica y suele ofrecer buenos resultados para problemas específicos.
   - La elección de estructuras de datos y algoritmos puede influir en el rendimiento real en diferentes escenarios.

En resumen, el algoritmo de Dijkstra, con una implementación eficiente de la cola de prioridad, proporciona soluciones rápidas para encontrar caminos mínimos en grafos ponderados dirigidos con pesos no negativos. La complejidad del algoritmo es razonable en muchos casos prácticos, pero es esencial considerar el contexto y los detalles de implementación para una evaluación completa.

4. Referencias

Referencias

[1] Baase, S., & Van Gelder, A. (2000). *Computer Algorithms: Introduction to Design and Analysis*. Pearson.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms, Fourth Edition*. MIT Press.

[3] Gross, J. L., & Yellen, J. (2005). *Graph Theory and its Applications*, 2nd edition. CRC Press.

[4] Sedgewick, R., & Wayne, K. (2014). *Algorithms: Part I*. Addison-Wesley Professional.

[5] Skiena, S. S. (2009). *The Algorithm Design Manual*. Springer Science & Business Media.