Unidad 1

Análisis de Complejidad Computacional

Docente:

M. en C. Erika Sánchez-Femat

Practica 3:

Algoritmos Voraces

por

Giovanna Inosuli Campos Flores

December 1, 2023

1 Introduction

Se mostrará la implementación en Python del algoritmo de Dijkstra, utilizado para encontrar la ruta más corta entre dos puntos en un grafo ponderado. Desarrollado por Edsger Dijkstra en 1956, este algoritmo ha sido ampliamente utilizado en informática y matemáticas. En la primera sección de este trabajo, se presentará una definición del algoritmo en Python, explicando sus características principales y cómo funciona.

2 Algoritmo

2.1 Definición del algoritmo

El algoritmo de Dijkstra es un método utilizado para encontrar la ruta más corta entre dos puntos en un grafo ponderado. Se basa en asignar un peso a cada arista del grafo y calcular las distancias mínimas desde un nodo inicial a todos los demás nodos. El algoritmo comienza inicializando las distancias y marcando el nodo inicial como visitado. Luego, selecciona el nodo no visitado con la distancia mínima y actualiza las distancias de sus nodos vecinos. Este proceso se repite hasta que todos los nodos hayan sido visitados. El algoritmo de Dijkstra tiene aplicaciones en diversas áreas como la búsqueda de la ruta más corta en un mapa, la optimización de redes de transporte y el análisis de redes sociales, entre otras.

2.2 Funcionamiento del algoritmo

El algoritmo de Dijkstra es un método eficiente para encontrar la ruta más corta entre dos nodos de un grafo ponderado. Para su funcionamiento, se siguen una serie de pasos. En el paso 1, se inicializan las distancias, asignando un valor infinito a todos los nodos excepto al nodo de inicio, al que se le asigna un valor de 0. En el paso 2, se selecciona el nodo actual, que es el nodo con la distancia más pequeña entre los nodos no visitados. En el paso 3, se actualizan las distancias de los nodos vecinos al nodo actual, recalculando sus distancias si es necesario. Por último, en el paso 4, se repiten los pasos anteriores hasta que todos los nodos hayan sido visitados o hasta encontrar la ruta más corta al nodo destino. Este algoritmo tiene diversas aplicaciones, como encontrar la ruta más corta en un mapa, optimizar redes de transporte, analizar redes sociales, entre otros ejemplos.[1]

3 Implementación en Python

3.1 Implementación del Grafo

Define una clase Graph que represente el grafo ponderado dirigido. Esta clase debe tener un constructor para inicializar la estructura del grafo, un método

add vertex para agregar vértices y un método add edge para agregar aristas con pesos.

```python
class Graph:
    def __init__(self): #constructor
        # Utilizamos un diccionario
        #para almacenar los vértices y las aristas con pesos
        self.graph = {}

    def add_vertex(self, vertex):
        # Agrega un vértice al grafo si no existe
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, inicio, final, peso):
        # Agrega una arista con peso entre los vértices dados
        if inicio in self.graph and final not in self.graph[inicio]:
            self.graph[inicio].append((final, peso))
        else:
            print(f"Error")
```

3.2 Implementación del Algoritmo de Dijkstra

Define una función dijkstra que tome el grafo y un vértice de inicio como entrada y devuelva los caminos mínimos desde el vértice de inicio a todos los demás vértices. Puedes utilizar un diccionario para mantener un registro de las distancias mínimas.

```python
def dijkstra(graph, inicio):
    # Inicializar el diccionario de distancias mínimas
    distancias= {vertex: float('infinity') for vertex in graph}
    distancias[inicio] = 0

    # Usar un heap para manejar las prioridades de los vértices
    cola_prioridad= [(0, inicio)]

    while cola_prioridad:
        distancia_actual, vertice_actual= heapq.heappop(cola_prioridad)

        # Si la distancia actual es mayor que la registrada, continuar al siguiente vértice
        if distancia_actual > distancias[vertice_actual]:
            continue

        # Actualizar las distancias para los vértices adyacentes
        for vecino, peso in graph[vertice_actual]:
            distancia = distancia_actual + peso

            # Si se encuentra un camino más corto, actualizar la distancia
            if distancia < distancias[vecino]:
                distancias[vecino] = distancia
                heapq.heappush(cola_prioridad, (distancia, vecino))

    return distancias
```

3.3 Prueba con un Grafo de Ejemplo

Crea un grafo de ejemplo utilizando la clase Graph y agrega algunos vértices y aristas con pesos. Ejemplo:

```
# Ejemplo de uso:
grafo = Graph()# Crear un Grafo
# Agregar vértices
grafo.add_vertex("A")
grafo.add_vertex("B")
grafo.add_vertex("C")
grafo.add_vertex("D")
grafo.add_vertex("E")
grafo.add_vertex("F")
# Agregar aristas con pesos
grafo.add_edge("A", "B", 2)
grafo.add_edge("B", "C", 1)
grafo.add_edge("C", "D", 8)
grafo.add_edge("C", "F", 5)
grafo.add_edge("C", "E", 3)
grafo.add_edge("D", "E", 11)
grafo.add_edge("E", "F", 6)
grafo.add_edge("F", "A", 3)
# Mostrar grafo
print(grafo.graph)
```

3.4 Prueba del Algoritmo de Dijkstra

Utiliza la función dijkstra para encontrar los caminos mínimos desde un vértice de inicio dado. Puedes imprimir los resultados para verificar que el algoritmo funcione correctamente. Ejemplo:

```
# Calcular los caminos mínimos desde el vértice 'A'
resultados = dijkstra(grafo.graph, 'A')
# Mostrar los resultados de Dijkstra
for vertex, distancia in resultados.items():
    print(f'Distancia mínima desde A hasta {vertex}: {distancia}')
```

4 Análisis del Algoritmo

4.1 Medición del Tiempo

Se utilizó el módulo ”time” para medir el tiempo de ejecución de la función de dijkstra, para ello se incluyo otra función llamada ”medir_tiempo” que ejecuta

la función, mide el tiempo de ejecución y luego imprime el tiempo de que se tardó en realizar la función de dijkstra.

```python
#funcion para medir tiempo
def medir_tiempo(funcion, *args, **kwargs):
    inicio_tiempo = time.time()
    resultado = funcion(*args, **kwargs)
    fin_tiempo = time.time()
    tiempo_ejecucion = fin_tiempo - inicio_tiempo
    print(f'Tiempo de ejecución: {tiempo_ejecucion} segundos')
    return resultado
```

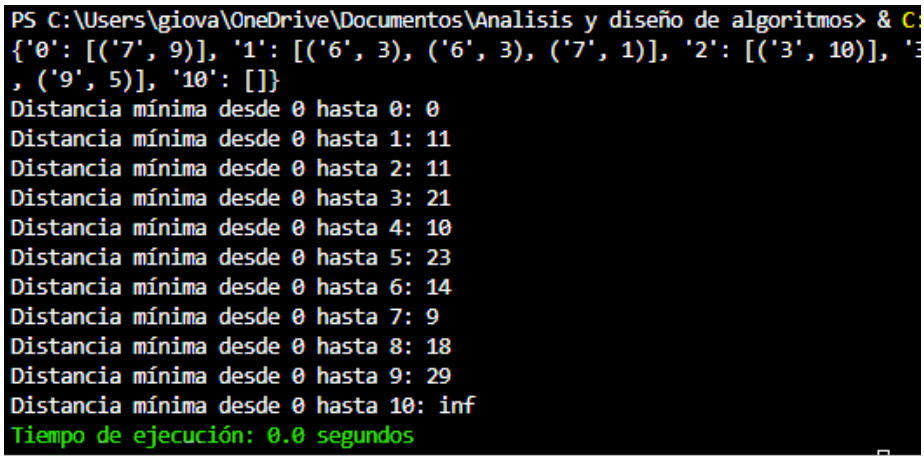
4.1.1 Resultados

Se realizó 5 casos diferentes, tratando de cambiar el número de vértices, el número de aristas y también del peso de cada arista, para ello utilizamos el módulo de "random" para poner un rango de valores y se vayan colando manera aleatoriamente para poder visualizar el tiempo de ejecución de las pruebas.

```python
# Ejemplo de uso:
grafo = Graph()# Crear un Grafo
# Agregar vértices
num_vertices = 6 #asignamos el numero de vertices deseados
for i in range(num_vertices):
    grafo.add_vertex(str(i))
# Agregar aristas con pesos aleatorios
for _ in range(num_vertices * 2):  # Aproximadamente el doble de aristas que d
    inicio= str(random.randint(0, num_vertices - 1))
    final = str(random.randint(0, num_vertices - 1))
    peso = random.randint(1, 10)
    grafo.add_edge(inicio, final, peso)

# Mostrar grafo
print(grafo.graph)

# Calcular los caminos mínimos desde el vértice '0' (por ejemplo)
resultados = dijkstra(grafo.graph, '0')

# Mostrar los resultados de Dijkstra
for vertex, distancia in resultados.items():
    print(f'Distancia mínima desde 0 hasta {vertex}: {distancia}')

# Calcular los caminos mínimos desde el vértice '0' y medir el tiempo de ejecu
medir_tiempo(dijkstra, grafo.graph, '0')
```

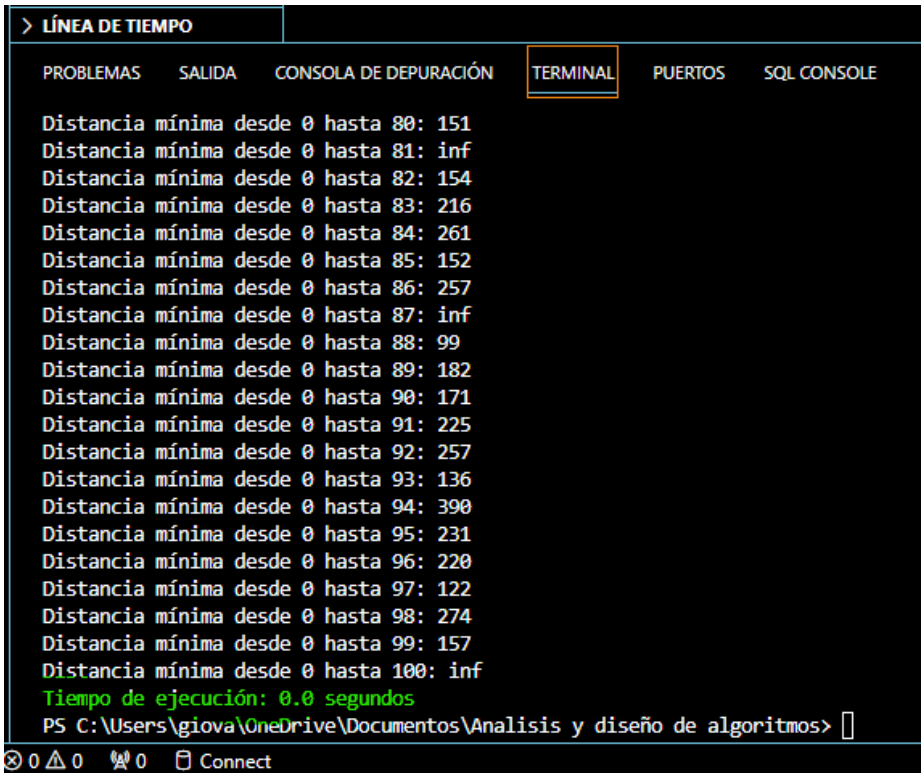
4.1.2 Casos

1. Primer caso, se realizó una prueba sencilla poniendo que realizara un grafo de 10 vértices y que tuviera el doble de aristas, y sus pesos podían tomar valores de 1 a 10, obtuvimos el siguiente resultado:



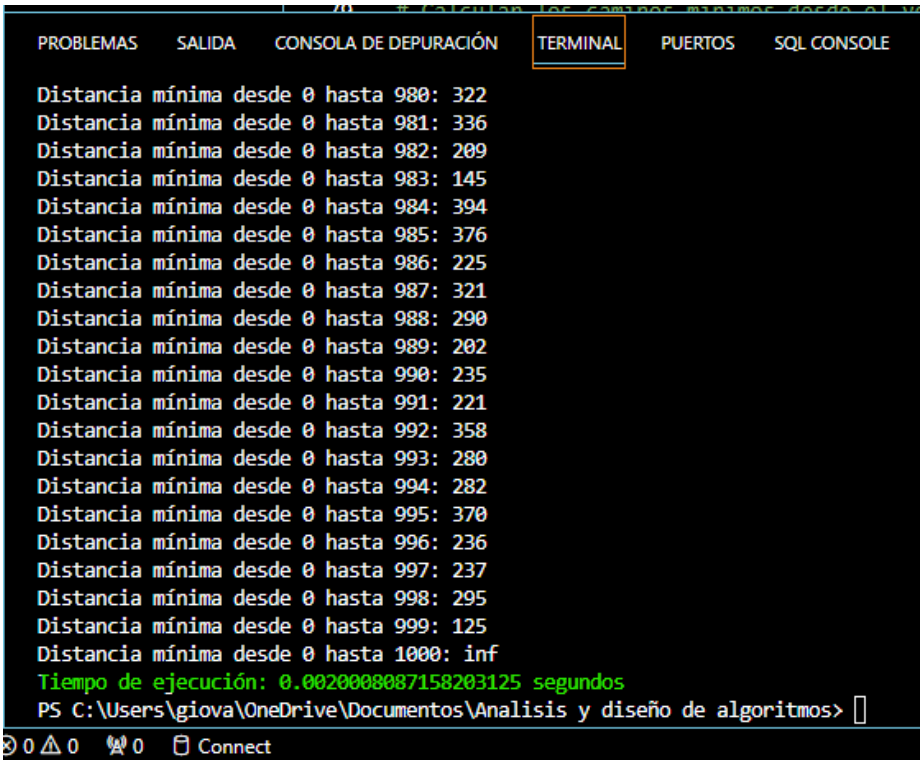
   Se puede observar que el tiempo de ejecución es igual a 0, es todo quiere decir que no le costó casi nada realizar encontrar los caminos de menor peso

2. segundo caso, se quiso observar cuanto era tiempo si un grafo tenía 100 vértices y casi el tripe de aristas con pesos de un rango de 1 a 100, su resultado fue él, siguiente

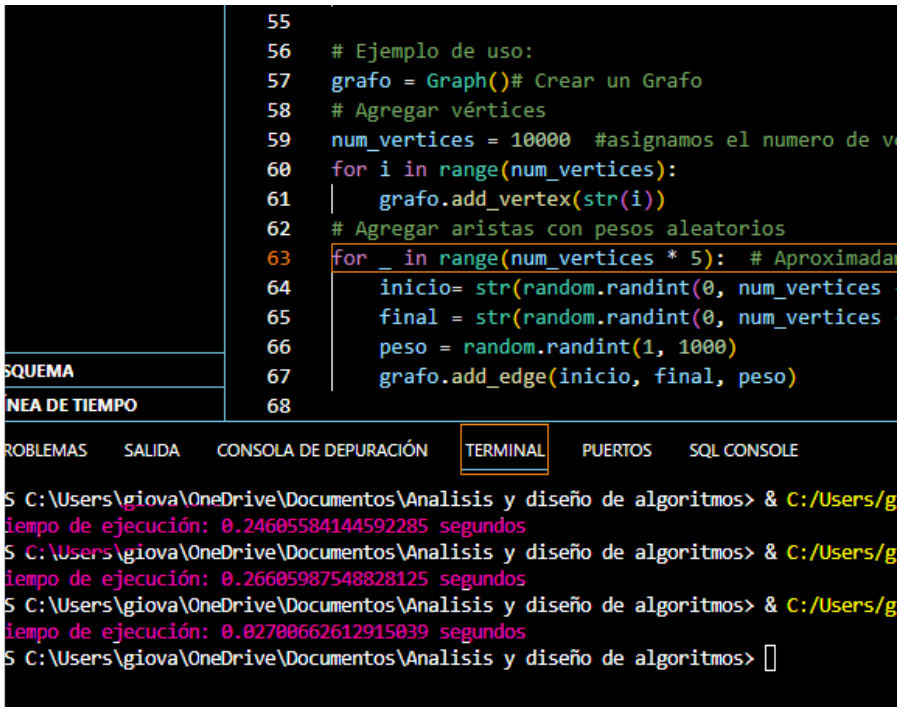


como se observa tuvimos el mismo resultado que el primer caso

3. tercer caso, se consideró solo incrementar el número de vértices a uno de 1000 para que el tiempo de ejecución sea mayor a 0, para poder observar cuanto se toma en realizar el grafo de ese tamaño, por lo que se obtuvo:



4. este caso incrementó el número de vértices a 10000, el número de las aristas sería 5 veces el número de vértices y los pesos podían tomar valores de 1 a 1000, en este caso solo imprimimos el tiempo de ejecución por qué el grafo fue de demasiado grande par imprimir todos los caminos. también realizamos 3 pruebas seguidas para ver un aproximado de los tiempos, porque nos salió lo siguiente:

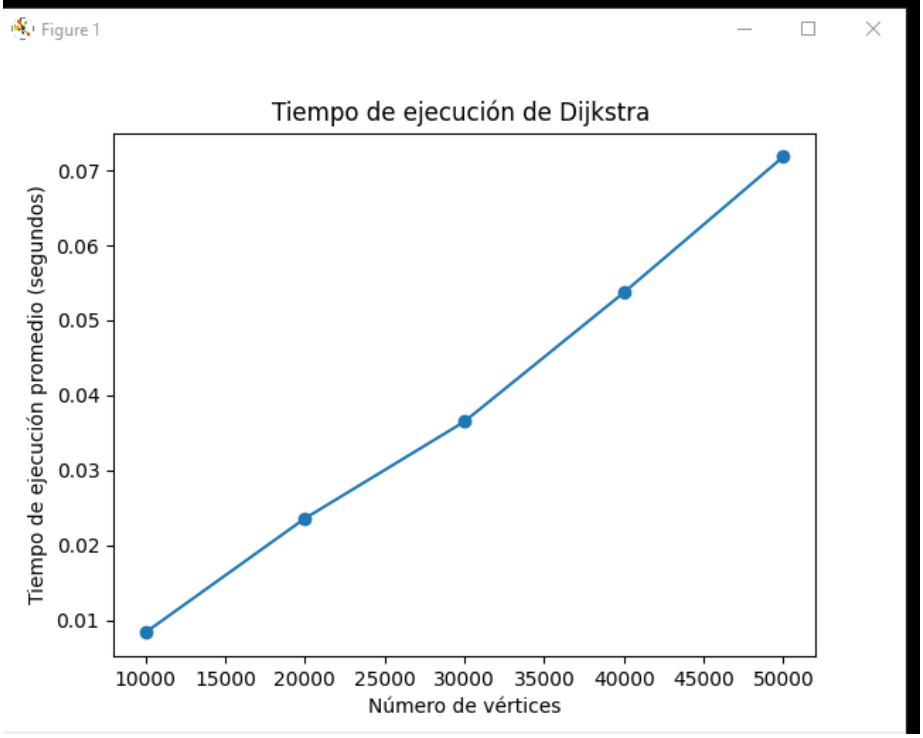


5. para este caso implantamos graficar el promedio de tiempo ejecución de dijkstra con función del número de vértices, por lo que incluimos otras 3 funciones para poder graficar los tiempos de las pruebas que realizamos.

```python
def generar_grafo_aleatorio(num_vertices):
    grafo = Graph()

    for i in range(num_vertices):
        grafo.add_vertex(str(i))

    for _ in range(num_vertices * 2):
        inicio = str(random.randint(0, num_vertices - 1))
        final = str(random.randint(0, num_vertices - 1))
        peso = random.randint(1, 100)
        grafo.add_edge(inicio, final, peso)

    return grafo.graph

def realizar_pruebas(num_pruebas, tamanios_grafos):
    resultados = []

    for num_vertices in tamanios_grafos:
        tiempos_ejecucion = []

        for _ in range(num_pruebas):
            grafo = generar_grafo_aleatorio(num_vertices)
            inicio_tiempo = time.time()
            dijkstra(grafo, '0')  # Suponiendo que siempre iniciamos desde el vértice '0'
            fin_tiempo = time.time()
            tiempo_ejecucion = fin_tiempo - inicio_tiempo
            tiempos_ejecucion.append(tiempo_ejecucion)

        resultados.append((num_vertices, sum(tiempos_ejecucion) / num_pruebas))

    return resultados

def graficar_tiempos(resultados):
    tamanios_grafos, tiempos_promedio = zip(*resultados)
    plt.plot(tamanios_grafos, tiempos_promedio, marker='o')
    plt.title('Tiempo de ejecución de Dijkstra ')
    plt.xlabel('Número de vértices')
    plt.ylabel('Tiempo de ejecución promedio (segundos)')
    plt.show()
```

La función que llamamos "generar_grafo_aleatorio" como indica nos sirve para crear grafos con valores aleatorios, la otra función sirve para que realice el algoritmo de Dijkstra para cada uno de los grafos que hemos generado y también guarde los tiempos ejecución. la última función que agregamos es la función que nos grafique los tiempos de ejecución

```
# Ejemplo de uso
num_pruebas = 10
tamanios_grafos = [10000, 20000, 30000, 40000, 50000]

resultados = realizar_pruebas(num_pruebas, tamanios_grafos)
graficar_tiempos(resultados)
```



Por que observamos en la gráfica, podemos decir que el algoritmo Dijkstra incrementa su tiempo de ejecución con cuando incrementa el número de vértices.

4.2 Cálculo de Complejidad

La complejidad del algoritmo voraz de Dijkstra se ve afectada por varios factores. En primer lugar, la densidad del grafo influye en la complejidad del algoritmo, ya que a mayor densidad, mayor cantidad de aristas debe analizar el algoritmo, lo que aumenta su tiempo de ejecución. Además, el peso de las aristas también afecta la complejidad, ya que si las aristas tienen pesos muy grandes, el algoritmo requerirá más operaciones para calcular el camino más corto. Por otro lado, el número de vértices en el grafo también influye en la complejidad, ya que a mayor número de vértices, mayor será la cantidad de operaciones que debe realizar el algoritmo. En resumen, estos factores pueden afectar de manera significativa la complejidad del algoritmo voraz de Dijkstra, lo que debe tenerse en cuenta al estudiar su rendimiento en diferentes casos de uso.[2] , [3]

continuación mostrará un par de ejemplo como se realiza el algoritmo de Dijsktra:

Sea $G = (V, E)$ un grafo simple no dirigido

Sea V el vértice inicial

Sea L conjunto de vértice de G con ruta optima conocida desde el punto inicial V

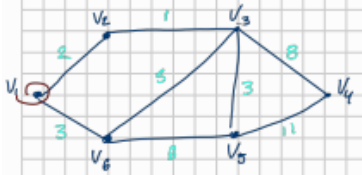
Sea L' conjunto de vértices restantes de G entonces $L' = V - L$

Sea D es una matriz de $(1 \times |V|)$ donde se almacena la información sobre la longitud de los caminos

Sea $D(i)$ las entradas de D entonces:

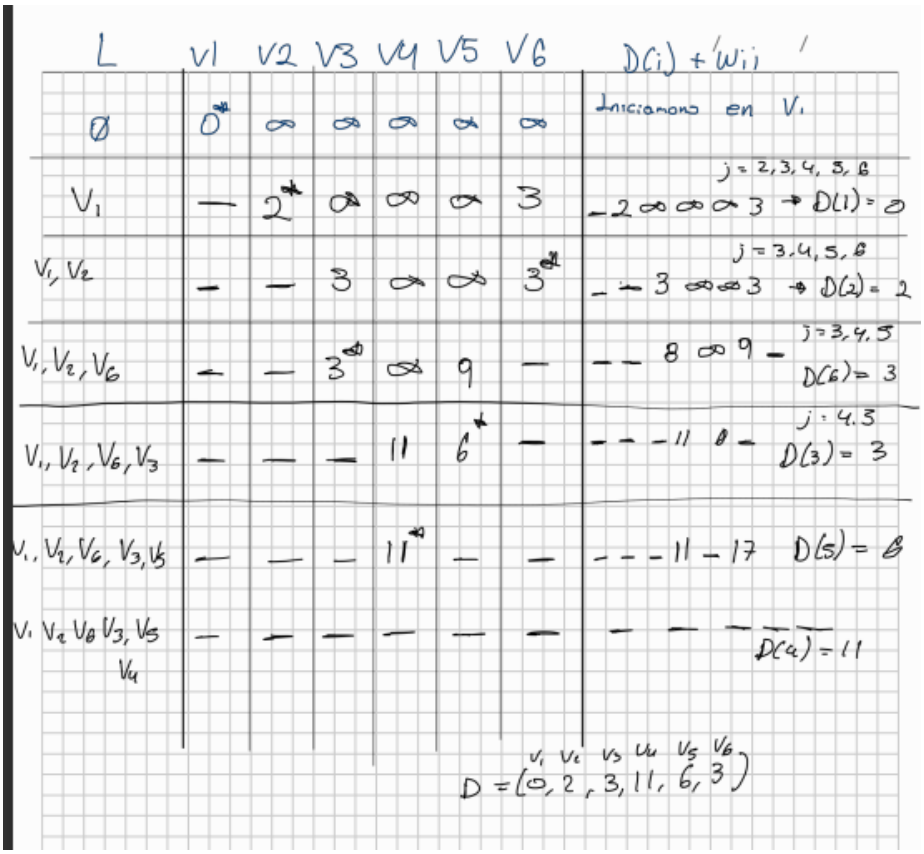
1) Si $V_i \in L$, $D(i)$ es la longitud de la ruta más corta de V_i a V
2) Si $V_i \in L'$, $D(i)$ es la longitud de la mejor ruta conocida, hasta el momento (puede o no ser optima de V_i a V)

ejemplo



$W_{ij} = w(\{V_i, V_j\})$

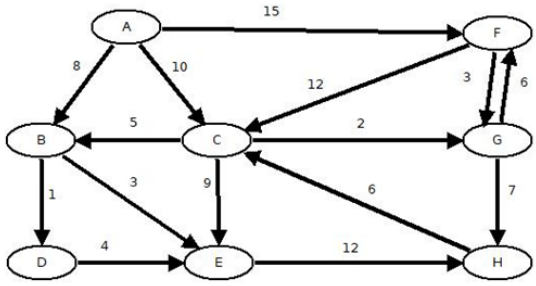
* temporalmente
- definido
∞



El algoritmo voraz de Dijkstra tiene una complejidad espacial que depende del tamaño del grafo de entrada, ya que se deben almacenar las estructuras de datos necesarias para su ejecución. La complejidad espacial del algoritmo es O(V), donde V es el número de vértices del grafo. Esto se debe a que se necesita un arreglo de distancias de tamaño V y un arreglo de visitados de tamaño V para mantener un registro de las distancias mínimas y los vértices visitados durante la ejecución del algoritmo. Además, se requiere una cola de prioridad de tamaño V para determinar el vértice con la distancia mínima en cada iteración. En resumen, la complejidad espacial del algoritmo voraz de Dijkstra es lineal respecto al número de vértices del grafo de entrada.

La complejidad temporal del algoritmo de Dijkstra es O((V+E)logV), siendo V el número de vértices y E el número de aristas en el grafo. Esta complejidad depende en gran medida de la implementación utilizada y puede variar en función de diversos factores, como la densidad del grafo, el peso de las aristas y el número de vértices.

Instrucciones: Aplica el algoritmo de Dijkstra a los siguientes grafos.



Matriz de Adyacencia

	A	B	C	D	E	F	G	H
Paso 1	∞	(5,c)	(0,∞)	∞	∞	∞	(2,c)	∞
Paso 2	∞	(5,c)	-	∞	∞	(8,g)	(2,g)	(9,g)
Paso 3	∞	(5,b)	-	(6,b)	(8,b)	(8,g)	-	(9,g)
Paso 4	∞	-	-	(6,d)	(8,b)	(8,g)	-	(9,g)
Paso 5	∞	-	-	-	(8,b)	(8,g)	-	(9,g)
Paso 6	∞	-	-	-	-	(8,f)	-	(9,g)
Paso 7	∞	-	-	-	-	-	-	(9,h)
Paso 8	∞	-	-	-	-	-	-	(9,h)

Lista de Adyacencia

	Lista de Adyacencia
A	∞
B	5= c->b
C	0=c
D	6= c->b->d
E	8= c->b->e
F	8=c->g->f
G	2= c->g
H	9= c->g->h

4.2.1 Densidad del grafo

La densidad del grafo es uno de los factores que influyen en la complejidad del algoritmo voraz de Dijkstra. La densidad del grafo se refiere a la cantidad de aristas presentes en relación con la cantidad total de vértices. Cuanto mayor sea la densidad, es decir, cuantas más aristas haya en proporción a los vértices, más tiempo y recursos requerirá el algoritmo para encontrar el camino más corto. Por otro lado, un grafo poco denso, con pocas aristas, permitirá que el algoritmo se ejecute más rápidamente y con menor consumo de recursos. Por lo tanto, al analizar la complejidad del algoritmo de Dijkstra, es esencial tener en cuenta la densidad del grafo en el que se aplica.

4.2.2 Peso de las aristas

El peso de las aristas en el algoritmo de Dijkstra es un factor clave para determinar su complejidad. Este algoritmo utiliza el concepto de grafo ponderado, donde cada arista tiene un peso asignado. La complejidad del algoritmo aumenta a medida que aumenta el peso de las aristas, ya que requiere de más tiempo y recursos para calcular las distancias mínimas. Por lo tanto, es importante tener en cuenta el peso de las aristas al analizar la complejidad del

algoritmo de Dijkstra.

4.2.3 Número de vértices

La complejidad del algoritmo voraz de Dijkstra está directamente influenciada por el número de vértices presentes en el grafo. A medida que aumenta el número de vértices, aumenta el tiempo de ejecución del algoritmo. Esto se debe a que el algoritmo considera todos los vértices del grafo para encontrar el camino más corto desde el vértice inicial a todos los demás vértices. Por lo tanto, a mayor número de vértices, mayor será la cantidad de comparaciones y operaciones necesarias para calcular las distancias mínimas. En resumen, la complejidad del algoritmo de Dijkstra se ve afectada significativamente por la cantidad de vértices presentes en el grafo.

5 Conclusion

En conclusión, el algoritmo voraz de Dijkstra presenta una complejidad temporal dependiente del número de aristas y vértices del grafo, pero es eficiente en comparación con otros algoritmos. La complejidad espacial también es importante, ya que requiere memoria adicional para almacenar los valores de distancia y los vértices visitados. Sin embargo, la importancia de estudiar su complejidad radica en su utilidad para encontrar la ruta más corta en un grafo ponderado dirigido. Aunque factores como la densidad del grafo y el peso de las aristas pueden influir en su complejidad, el algoritmo de Dijkstra es una opción confiable en muchas aplicaciones prácticas. La complejidad del algoritmo de Dijkstra es un aspecto fundamental a considerar al utilizarlo en la resolución de problemas de rutas o en la optimización de redes, por qué se destaca por su eficiencia y simplicidad.

Referencias

References

[1] J. E. Allauca, "Aplicación de la teoría de grafos en la optimización de redes de transporte," Ciencia Inteligente, 2023. `cienciainteligente.com`

[2] C. Delgado Aguilar, "Análisis y simulación de redes inalámbricas de próxima generación, mediante teoría de redes complejas," 2021.`upm.es`

[3] S. Salazar Cárdenas,"... MATEMÁTICOS DE PROPAGACIÓN DE MALWARE Y CIBEREPIDEMIAS BASADOS EN ANÁLISIS DE REDES COMPLEJAS Y ECUACIONES DIFERENCIALES," 2023.`urjc.es`