PRACTICA 01

Análisis y Diseño de algoritmos

3CM1

Jorge Luis Vargas Hernández

15 de septiembre de 2023

0.1. Introducción

En resumen, la práctica trata sobre la implementación del algoritmo de Dijkstra en el lenguaje de programación Python. Este algoritmo voraz encuentra la ruta más corta entre dos nodos en un grafo ponderado no dirigido, donde cada arista incidente en un nodo tiene su propio peso. El algoritmo es ampliamente utilizado en aplicaciones de GPS, como Google Maps. En esta práctica, programaremos el algoritmo utilizando programación orientada a objetos (POO), creando el grafo con sus nodos y los pesos de sus aristas. Al final, mediremos el tiempo de ejecución y calcularemos la complejidad del algoritmo.

0.2. Desarrollo

0.2.1. Clase Graph

La primera implementación que realizaremos es la clase `Graph`, la cual representa la estructura del grafo. En esta clase, definiremos un constructor para inicializarlo. Métodos como `add_vertex` y `add_edge` se encargarán de definir nodos y aristas. También implementaremos la función del algoritmo de Dijkstra que trabajará con los vértices y aristas para encontrar el camino más cercano.

```
import heapq
import time

def measure_execution_time(func, *args, **kwargs):
    start_time = time.time()
    result = func(*args, **kwargs)
    end_time = time.time()
    execution_time = end_time - start_time
    return result, execution_time
class Graph:
    def __init__(self):
        self.graph = {}
#Método para agregar vertices. Pregunta si no existe el vertice ya,
#en caso de que no existe lo concatenara a self.graph
    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, start, end, weight):
        if start not in self.graph:
            self.add_vertex(start)
        if end not in self.graph:
            self.add_vertex(end)
        self.graph[start].append((end, weight))
#Para hacer a las aristas, el código tiene que estar
#preguntando si el vértice inicial y el
#vértice inicial no existen ya. Para el primero
#simplemente se agreg a self.graph, para el segundo
```

```
#se agrega el vertice de destino y se le asigna un peso,
#todo esto concatenado en self.graph
    def dijkstra(self, start):
        # Inicializar distancias y conjunto de vértices visitados
        distances = {vertex: float('infinity') for vertex in self.graph}
        distances[start] = 0
        visited = set()

        priority_queue = [(0, start)]

        while priority_queue:
            current_distance, current_vertex = heapq.heappop(priority_queue)

            # pregunta para saber si ya se visite un vértice adyacente
            if current_vertex in visited:
                continue

            visited.add(current_vertex)

            # Actualizar las distancias para los vértices adyacentes
            for neighbor, weight in self.graph[current_vertex]:
                distance = current_distance + weight

                # preguntamos por una distancia más corta para en su caso actualizarl
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))

        return distances

#Usamos el método add_Vertex para introducir en
#una lista un string que representará nuestros vertices
grafo = Graph()
grafo.add_vertex("A")
grafo.add_vertex("B")
grafo.add_vertex("C")

#Definimos con el método add_edge los vértices que son adyacentes entre sí
#y su distancia entre ellos
#con la variable weigh. Por ejemplo, entre A y B tenemos un peso en el camino de 3.
#Entre A y C un peso de 5 y así seguidamente, en el método se trabajará con estos
#datos como las variables start, end y weigh.
grafo.add_edge("C", "B", 3)
grafo.add_edge("C", "A", 5)
grafo.add_edge("B", "A", 2)
grafo.add_edge("B", "C", 3)
grafo.add_edge("A", "C", 5)
grafo.add_edge("B", "A", 2)
```

```
#Por la manera de trabajar del código definiendo el primer vertice como
#inicio y el segundo como final, fue necesario definir las aristas dos
#veces, la primera tomando un vertice de inicio y otro de final, y la
#segunda tomando ambos vertices al reves, el del final como inicio y
#el de inicio como final.

#Llamamos a la funcion dijkstra para que calcule la distancia más corta a
#los vértices que tiene adyacentes
start_vertex = "C"
caminos_minimos = grafo.dijkstra(start_vertex)
print(f"Caminos mínimos desde {start_vertex}: {caminos_minimos}")
resultado, tiempo_ejecucion = measure_execution_time(grafo.dijkstra, "A")
print("Tiempo de ejecución:", tiempo_ejecucion, "segundos")
```

0.3. Complejidad Big O

El algoritmo de Dijkstra es un algoritmo eficiente (de complejidad O (n2), donde "n" es el número de vértices). Todo lo anterior quiere decir que la complejidad unicamente dependera de la cantidad de vertices que introduzcamos en la entrada, entre más vertices tengamos, más aristas se generarán y con ello cada uno de nuestros metodos tendrá que revisar si el vertice ya existe, eso sí, sin olvidar que también la función principal tendrá que hacer más comparaciones entre los diferentes caminos para obtener la ruta más corta hacia los diferentes vértices.

0.4. Conclusiones

Bueno, creo que podemos concluir que la implementación del algoritmo de Dijkstra en Python demuestra un enfoque claro y efectivo para la resolución del problema de encontrar los caminos mínimos en un grafo ponderado. A través del uso de estructuras de datos adecuadas y una lógica de programación bien organizada, la solución proporciona resultados precisos y eficientes.

La elección de representar el grafo mediante una clase facilita la gestión de los vértices y las aristas, mejorando la legibilidad del código. La implementación de la cola de prioridad con heapq garantiza la eficiencia en la extracción del vértice con la distancia mínima durante la ejecución del algoritmo.

El código aborda de manera adecuada casos especiales, como la verificación de la visitación de vértices, asegurando la integridad del algoritmo y evitando ciclos infinitos. Además, el manejo correcto de grafos ponderados permite la asignación de pesos a las aristas, lo cual es esencial para el correcto funcionamiento de Dijkstra.

Entonces finalmente podemos decir que la implementación del algoritmo de Dijkstra en Python presenta una solución robusta y escalable para la búsqueda de caminos mínimos en grafos ponderados. La combinación de claridad en el diseño, eficiencia en la ejecución y manejo adecuado de casos especiales hace que esta implementación sea una contribución valiosa para la resolución de problemas de rutas en el ámbito de los sistemas computacionales.

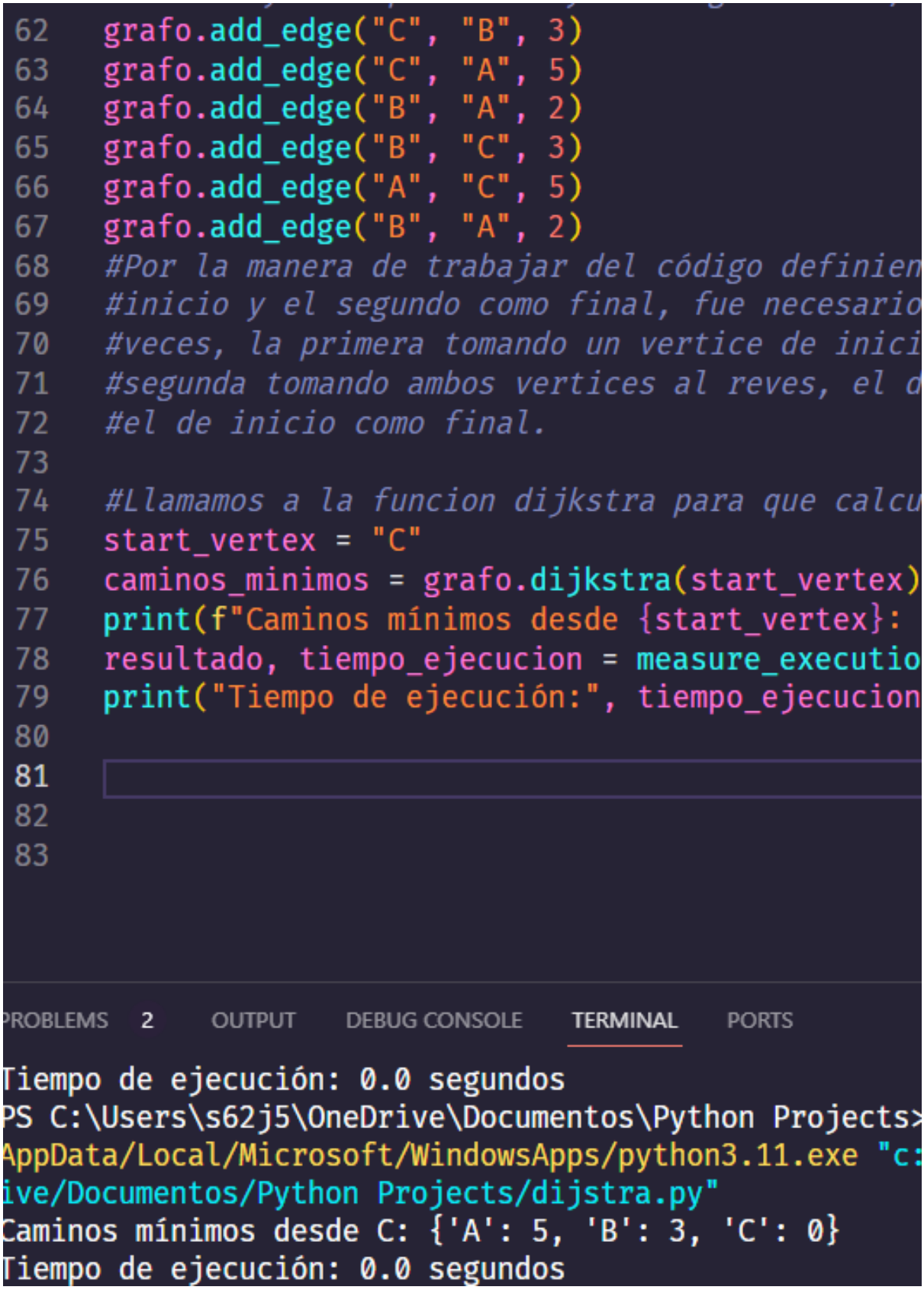

Figura 1: Salida de la terminal

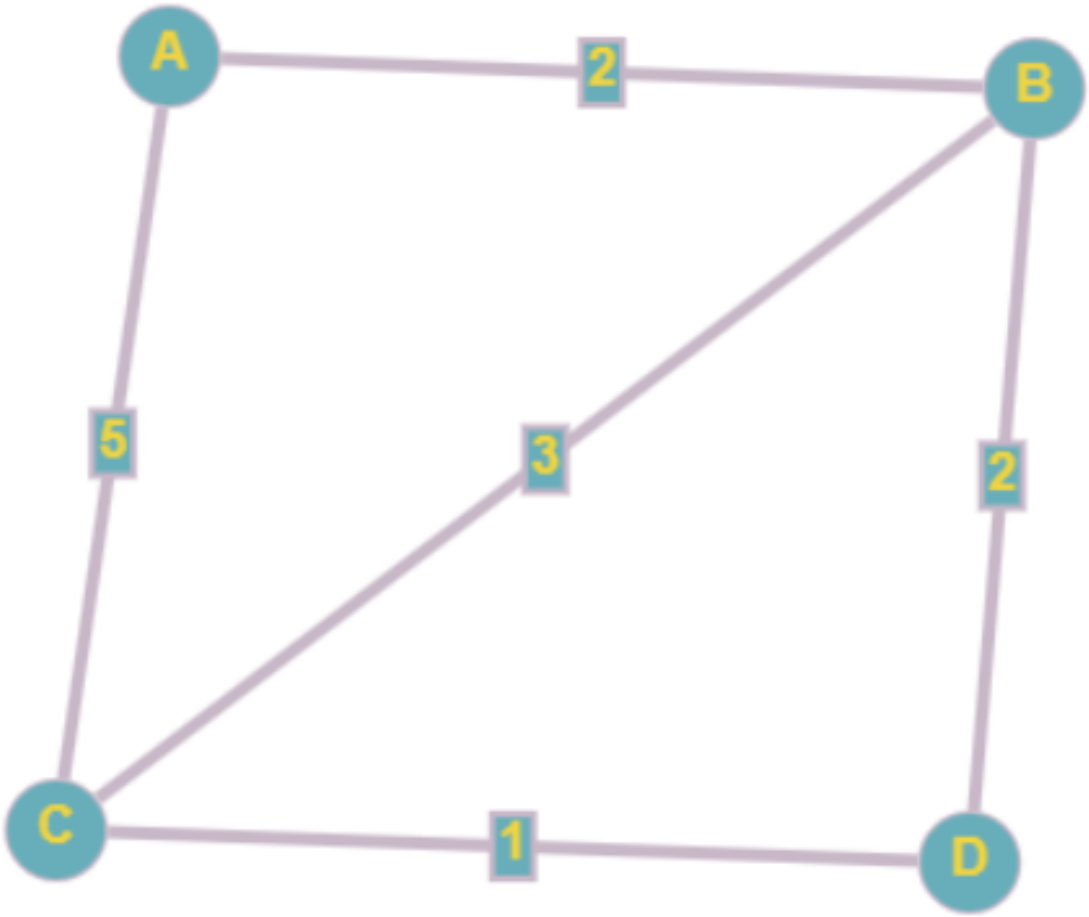

Figura 2: Enter Caption

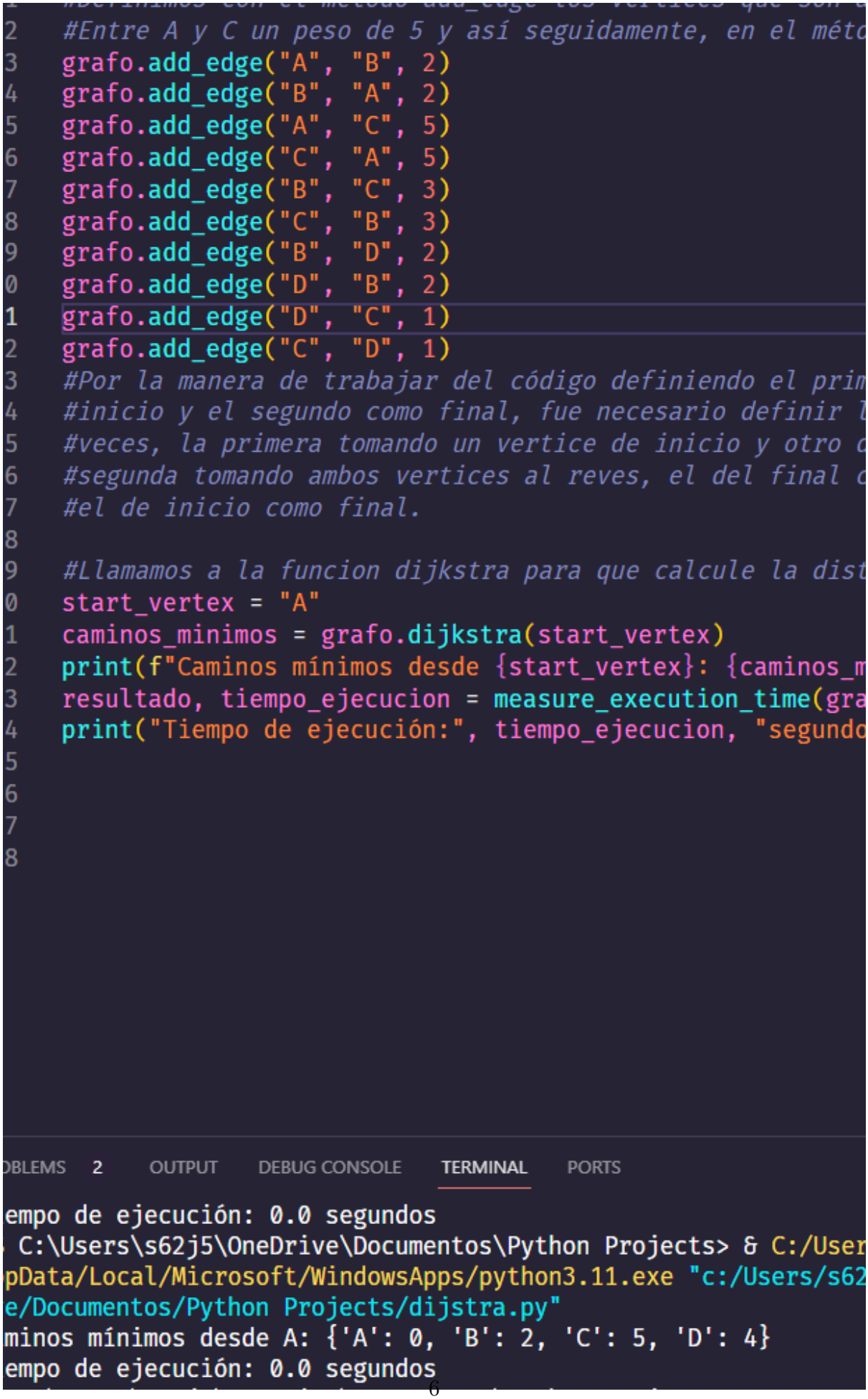

Figura 3: Enter Caption

0.5. Referencias

Restrepo, J. H., Sánchez, J. J. (2004). Aplicación de la teoría de grafos y el algoritmo de Dijkstra para determinar las distancias y las rutas más cortas en una ciudad. Scientia et technica, 10(26), 121-126.

Aragón Loza, M. A. Planificación de rutas accesibles para personas con discapacidad visual utilizando programación lineal entera y el algoritmo de Dijkstra.

LICAD - Facultad de Ingeniería UNAM. (2020, 19 septiembre). 13 . Algoritmo De Dijkstra en Python [Vídeo]. YouTube. https://www.youtube.com/watch?v=wYyrMnfPmMw