INSTITUTO POLITÉCNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERIA CAMPUS ZACATECAS

Análisis y Diseño de Algoritmos

PRÁCTICA 2

Implementación y Evaluación del Algoritmo de Dijkstra.

DOCENTE

M. en C. Erika Sánchez-Femat

POR EL ALUMNO

Cristian García Nieves

GRUPO

3CM1

FECHA

Viernes 01 de diciembre del 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 este práctica, se explorará su funcionamiento y lo se implementará en Python.

2. Desarrollo

Creación del grafo

Para comenzar con el desarrollo del algoritmo primero fué necesario crear la parte para agregar y crear el grafo, es decir, la parte en la que vamos a especificar cuantos nodos tenemos, para luego poner el peso que hay de un nodo a otro.

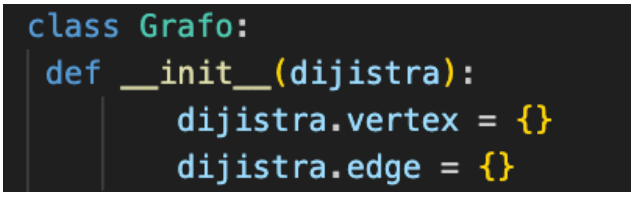

Figura 1: metodo inicial

Para comenzar creamos una clase llamada **"Grafo"** donde definimos un método de inicialización, es decir se ejecuta esa parte en cuanto se utiliza la clase grafo.

Entonces en cuanto se ejecuta la clase grafo, creamos dos diccionarios, un dicionario **"vertex"** que será donde guardaremos los vertices y otro diccionario **edge** que será donde agregaremos las aristas del grafo.

Luego creamos un metodo para agregar un vértice en el diccionario **"vertex"** donde antes de agregar un vértice debemos asegurarnos de que el vertice que estamos intentando agregar, no exista en el diccionario, por lo que se utilizó un condicional if para comprobar ese aspecto.

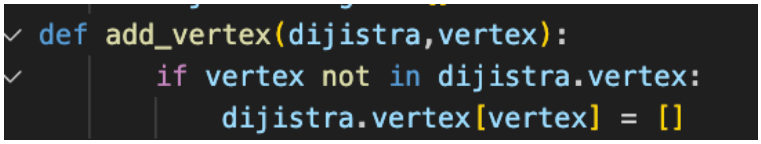

Figura 2: Método add vertex

Si el vertice que estamos intentando agregar, no existe, vamos a agregar ese vértice en el diccionario que creamos, llamado dijkstra.vertex, donde la clave sería el vértice que queremos agregar y el atributo lo agregamos como una lista vacía

Muy bien, entonces ahora ya temnemos el método para agregar nuestros vértices, ahora sigue para agregar nuestra aristas. Para agregar nuestras aristas, debemos crear otro método para agregar esas aristas pero debemos verificar que existan los vertices que vamos a utilizar.

```python
def add_edge(dijistra, start_vertex, end_vertex, peso):
    if start_vertex in dijistra.vertex and end_vertex in dijistra.vertex:
        dijistra.edge[(start_vertex,end_vertex)] = peso
        dijistra.vertex[start_vertex].append(end_vertex)
        dijistra.vertex[end_vertex].append(start_vertex)
    else:
        print("Algun vertice no existe en el grafo")
```

Figura 3: Método add edge

Aqui primero en el diccionario edge agregamos una nueva clave que es el primer vertice y el segundo, como atributo es el peso que hay entre esos 2 vertices.
Luego para agregar los vecinos lo que hacemos es llenar la lista vacía que habíamos creado en el diccionario de edge. Donde primero el la clave del vertice de inicio, agregamos en la lista que es el atributo, el vertice que está como vecino. y repeteimos el mimo paso pero al contrario, es decir usamos como clave el vertice final y agregamos a la lista el vertie vecino.

```python
class Grafo:
    def __init__(dijistra):
        dijistra.vertex = {}
        dijistra.edge = {}
    def add_vertex(dijistra,vertice):
        if vertice not in dijistra.vertex:
            dijistra.vertex[vertice] = []
    def add_edge(dijistra, start_vertex, end_vertex, peso):
        if start_vertex in dijistra.vertex and end_vertex in dijistra.vertex
            dijistra.edge[(start_vertex,end_vertex)] = peso
            dijistra.vertex[start_vertex].append(end_vertex)
            dijistra.vertex[end_vertex].append(start_vertex)
        else:
            print("Algun vertice no existe en el grafo")


    def __str__(dijistra):
        return f"Vertices: {dijistra.vertex}\n Aristas: {dijistra.edge}"
```

Figura 4: código con elementos esenciales para crear el grafo.

Finalmente agregamos un else para que en caso de que un vertice no exista, pues informar eso, y creamos otro método para que si queremos imprimir los vertices y aristas, tengamos la posibilidad de hacerlo.

Algiritmo Dijkstra

Una vez terminado lo necesario para crear el grafo, se creó el método para implementar el algoritmo de **Dijkstra**

```
def dijkstra(dijistra, vertice_inicio):
    distancias = {v: float('infinity') for v in dijistra.vertex}
    distancias[vertice_inicio] = 0

    lista = [(0, vertice_inicio)]
```

Figura 5: Creamos la función de dijkstra

Aquí lo que hacemos es pedimos como parametro un vertice de inicio. Después creamos otro diccionario llamado **"distancias"** donde llenamos cada clave como un vector, y el atributo lo dejamos como infinito. Sin embargo, en el vertice de inicio, sabemos que la distancia que hay del vertice de inicio a si mismo es 0, por lo que a la clave de vertice inicio le cambiamos su atributo por 0 //

Después creamos una lista llamada **"heap"** que contiene principalmente dos atributos, primero la distancia y luego el vertice de inicio.

```
lista = [(0, vertice_inicio)]

while lista:
    distancia_actual, vertice_actual = heapq.heappop(lista)
    if distancia_actual > distancias[vertice_actual]:
        continue
    for vecinos in dijistra.vertex[vertice_actual]:

        peso = dijistra.edge.get((vertice_actual, vecinos), float('infinity'))
        peso_alrevez = dijistra.edge.get((vecinos, vertice_actual), float('infinity'))
        peso = min(peso, peso_alrevez)
        distance = distancia_actual + peso
        if distance < distancias[vecinos]:
            distancias[vecinos] = distance
            heapq.heappush(lista, (distance, vecinos))


return distancias
```

Figura 6: Resto de la funcion Dijkstra

Finalmente implementamos el algoritmo que se encarga de verificar y calcular las distancias a los vecinos desde el vertice de inicio. Donde obtenemos el peso del vertice de inicio al vecino y luego al reves, para comparar cual es menor y posteriormente sumarla a la distancia actual. luego comparamos si la distancia que obtuvimos es menor a la distancia que teníamos para actualizar el diccionario.
Y así se repite mientras se van agregando y elimiando elementos de la lista hasta que ya no quede ninguno.

Medición del tiempo

Para medir el tiempo lo que se hizo fué agregar en el metodo principal de la clase grafo dos variables, uno para el tiempo inicial y otro para el tiempo final.

```
def __init__(dijistra):
    dijistra.vertex = {}
    dijistra.edge = {}
    dijistra.inicial = 0
    dijistra.final = 0
```

Figura 7: Definicion de variables para medicion de tiempo

En este caso, lo que interesa es calcular principalmente el tiempo que tarda en encontrar las distancias minimas, por lo que nos enfocaremos en calcular el tiempo que toma ejecutarse la función **"dijkstra"**.

```
def dijkstra(dijistra, vertice_inicio):
    dijistra.inicial = time.time()
    distancias = {v: float('infinity') for v in dijistra.vertex}
    distancias[vertice_inicio] = 0
```

Figura 8: Obtenemos el tiempo que se lleva desde que se inició el programa

Guardamos en la variable inicial el tiempo transucrrido desde que se inició el código, esto lo hacemos para poder saber en que momento inició el código y cuando tengamos el tiempo cuando acaba, poder restar ese tiempo inicial y así obtener el tiempo que tardó solo el algoritmo de dijkstra.

```
                heapq.heappush(lista, (distance, vecinos))


    dijistra.final = time.time()
    return distancias
```

Figura 9: Obtenemos el tiempo que se lleva cuando terminó la función dijkstra

```python
def tiempo_tardado(dijistra):
    return (dijistra.final-dijistra.inicial)
```

Figura 10: Función para obtener tiempo

Ya tenemos lista la función para obtener el tiempo tardado, por lo que cuando queramos saber el tiempo, solo debemos mandarlo a llamar.

2.1. Complejidad

La complejidad del algoritmo de Dijkstra es $O((V+E)\log V)$, donde V es el número de vértices y E es el número de aristas en el grafo. Esta complejidad se deriva de la utilización de un heap binario para mantener las distancias mínimas durante la ejecución del algoritmo. La operación clave en el bucle principal implica extracción e inserción en el heap, ambas operaciones con complejidad logarítmica. Aunque el algoritmo puede adaptarse a diversos tipos de grafos, la complejidad refleja el costo de explorar los vértices y aristas en función de su cantidad, con un componente logarítmico asociado al uso eficiente del heap. En resumen, la complejidad refleja la eficiencia del algoritmo al encontrar el camino más corto desde un vértice de inicio a todos los demás vértices en un grafo ponderado.

3. Resultados

Primer ejemplo

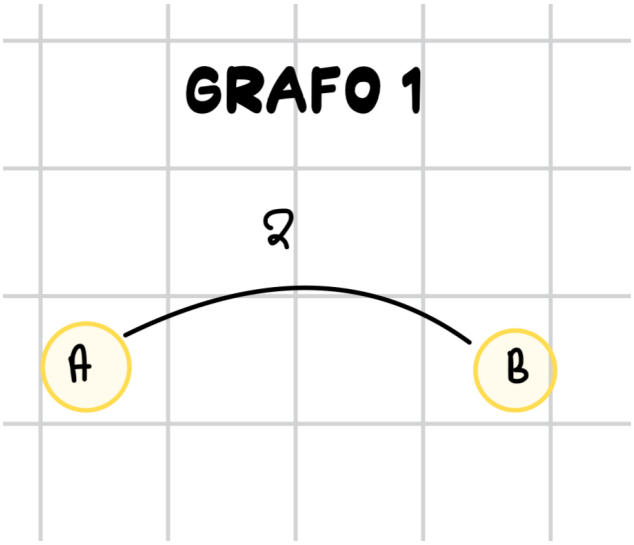

Figura 11: Ejemplo grafo 1

En este ejemplo tenemos un grafo con 2 vertices, entonces la solución de ese grafo es muy sencillo, sin embargo vamos a comprobar el resultado del algoritmo y el tiempo que tarda desde el vertice de inicio A.

```
vertice_inicio = 'A'
#GRAFO 1
grafo_1 = Grafo()

grafo_1.add_vertex('A')
grafo_1.add_vertex('B')

grafo_1.add_edge('A','B',2)

T1 = grafo_1.dijkstra(vertice_inicio)
TI_1 = grafo_1.tiempo_tardado()
print(f"GRAFO 1: Caminos mínimos desde {vertice_inicio}: {T1}")
print("Tiempo tardado: ",TI_1)
```

Figura 12: Codigo de grafo 1

Definomos los vertices A Y B para posteriormente asignarle el peso que hay entre a y b Aqui nos mues-

```
GRAFO 1: Caminos mínimos desde A: {'A': 0, 'B': 2}
Tiempo tardado:  7.867813110351562e-06
```

Figura 13: Resultado de grafo 1

tra los caminos miminos desde a hasta los otos vertices. Además vemos que el tiempo que tardó fue de aproximadamente 7.8 mili segundos.

Segundo ejemplo

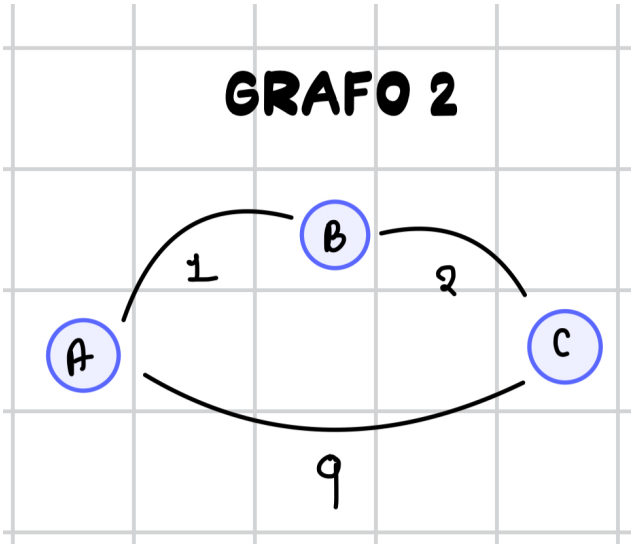

Figura 14: Ejemplo grafo 2

En este ejemplo tenemos un grafo con 3 vertices, aquí se ve mas notable el trabajo del algoritmo ya que la distancia mas corta desde a hasta c es pasar primero por b y luego llegar a c

```python
#GRAFO 2
grafo_2 = Grafo()

grafo_2.add_vertex('A')
grafo_2.add_vertex('B')
grafo_2.add_vertex('C')

grafo_2.add_edge('A','B',1)
grafo_2.add_edge('A','C',9)
grafo_2.add_edge('B','C',2)

T2 = grafo_2.dijkstra(vertice_inicio)
TI_2 = grafo_2.tiempo_tardado()
print(f"GRAFO 2: Caminos mínimos desde {vertice_inicio}: {T2}")
print("Tiempo tardado: ",TI_2)
```

Figura 15: Codigo de grafo 2

Asignamos los vertices A,B,C, los pesos y ejecutamos el código.

```
GRAFO 2: Caminos mínimos desde A: {'A': 0, 'B': 1, 'C': 3}
Tiempo tardado:  4.76837158203125e-06
```

Figura 16: Resultado de grafo 2

Aqui nos muestra los caminos miminos desde a hasta los otos vertices. Además vemos que el tiempo que tardó fue de aproximadamente 4.7 mili segundos.

Tercer ejemplo

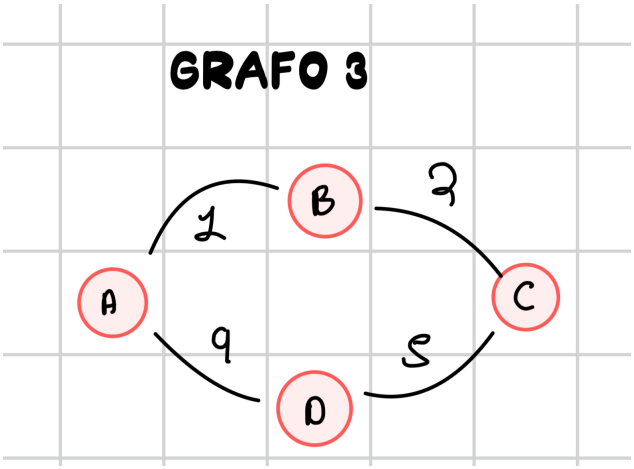

Figura 17: Ejemplo grafo 3

En este ejemplo tenemos un grafo con 4 vertices, aquí se ve mas notable el trabajo del algoritmo ya que la distancia mas corta desde a hasta los vértices restaness

```
#GRAFO
grafo_3 = Grafo()

grafo_3.add_vertex('A')
grafo_3.add_vertex('B')
grafo_3.add_vertex('C')
grafo_3.add_vertex('D')

grafo_3.add_edge('A','B',1)
grafo_3.add_edge('B','C',2)
grafo_3.add_edge('A','D',9)
grafo_3.add_edge('D','C',5)

T3 = grafo_3.dijkstra(vertice_inicio)
TI_3 = grafo_3.tiempo_tardado()
print(f"GRAFO 3: Caminos mínimos desde {vertice_inicio}: {T3}")
print("Tiempo tardado: ",TI_3)
```

Figura 18: Codigo de grafo 3

Asignamos los vertices A,B,C,D, los pesos y ejecutamos el código.

```
GRAFO 3: Caminos mínimos desde A: {'A': 0, 'B': 1, 'C': 3, 'D': 8}
Tiempo tardado:  6.198883056640625e-06
```

Figura 19: Resultado de grafo 3

Aqui nos muestra los caminos miminos desde a hasta los otos vertices. Además vemos que el tiempo que tardó fue de aproximadamente 6.2 mili segundos.

Cuarto ejemplo

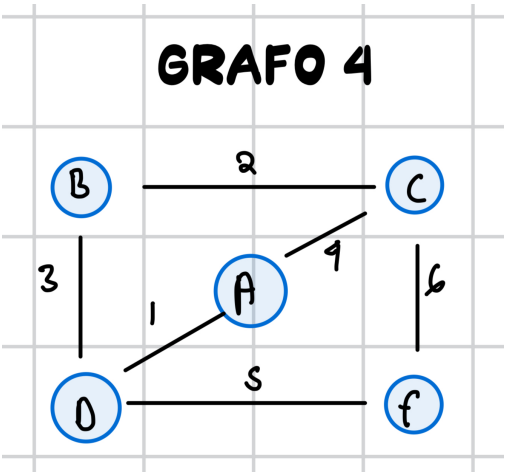

Figura 20: Ejemplo grafo 4

En este ejemplo tenemos un grafo con 5 vertices, aquí se ve mas notable el trabajo del algoritmo ya que la distancia mas corta desde a hasta los vértices restaness

```python
#GRAFO 4
grafo_4 = Grafo()

grafo_4.add_vertex('A')
grafo_4.add_vertex('B')
grafo_4.add_vertex('C')
grafo_4.add_vertex('D')
grafo_4.add_vertex('F')

grafo_4.add_edge('B','C',2)
grafo_4.add_edge('C','F',6)
grafo_4.add_edge('F','D',5)
grafo_4.add_edge('D','B',3)
grafo_4.add_edge('A','C',4)
grafo_4.add_edge('A','D',1)



T4 = grafo_4.dijkstra(vertice_inicio)
TI_4 = grafo_4.tiempo_tardado()
print(f"GRAFO 4: Caminos mínimos desde {vertice_inicio}: {T4}")
print("Tiempo tardado: ",TI_4)
```

Figura 21: Codigo de grafo 4

Asignamos los vertices A,B,C,D,F, los pesos y ejecutamos el código.

```
GRAFO 4: Caminos mínimos desde A: {'A': 0, 'B': 4, 'C': 4, 'D': 1, 'F': 6}
Tiempo tardado:  7.867813110351562e-06
```

Figura 22: Resultado de grafo 4

Aqui nos muestra los caminos miminos desde a hasta los otos vertices. Además vemos que el tiempo que tardó fue de aproximadamente 7.86 mili segundos.

Quinto ejemplo

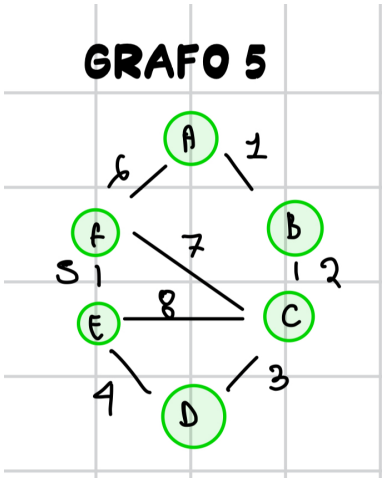

Figura 23: Ejemplo grafo 5

En este ejemplo tenemos un grafo con 6 vertices, aquí se ve mas notable el trabajo del algoritmo ya que la distancia mas corta desde a hasta los vértices restaness

```python
#GRAFO 5
grafo_5 = Grafo()

grafo_5.add_vertex('A')
grafo_5.add_vertex('B')
grafo_5.add_vertex('C')
grafo_5.add_vertex('D')
grafo_5.add_vertex('E')
grafo_5.add_vertex('F')

grafo_5.add_edge('A','B',1)
grafo_5.add_edge('B','C',2)
grafo_5.add_edge('C','D',3)
grafo_5.add_edge('D','E',4)
grafo_5.add_edge('E','F',5)
grafo_5.add_edge('A','F',6)
grafo_5.add_edge('F','C',7)
grafo_5.add_edge('E','C',8)
T5 = grafo_5.dijkstra(vertice_inicio)
TI_5 = grafo_5.tiempo_tardado()
print(f"GRAFO 5: Caminos mínimos desde {vertice_inicio}: {T5}")
print("Tiempo tardado: ",TI_5)
```

Figura 24: Codigo de grafo 5

Asignamos los vertices A,B,C,D,E,F, los pesos y ejecutamos el código.

```
GRAFO 5: Caminos mínimos desde A: {'A': 0, 'B': 1, 'C': 3, 'D': 6, 'E': 10, 'F': 6}
Tiempo tardado:  1.0967254638671875e-05
```

Figura 25: Resultado de grafo 5

Aqui nos muestra los caminos miminos desde a hasta los otos vertices. Además vemos que el tiempo que tardó fue de aproximadamente 10.96 mili segundos.

4. Conclusiones

En esta práctica puse a prueba mis conocimientos adquiridos, principamente en el uso de una clase en python, además también puse en práctica el uso de los diccionarios. Donde primero lo utilizo para crear un diccionario de los vértices y luego otro diccionario para establecer las aristas y los pesos. Y aunque en la parte del método de Dijkstra me resultó un poco comiplicado entender y el saber como se puede programar el algoritmo. Primero hice el código donde tenía un gráfo con 3 vertices e iba analizando e imprimiendo los valores que tenía en cada momento para así análizar lo que sucedía en cada momento y así saber que continúa y poder programarlo. Sin embargo, se logró programar y entender el funcionamiento de este algoritmo

5. Referencias

freeCodeCamp.org