



**INSTITUTO POLITÉCNICO NACIONAL**  
Unidad Profesional Interdisciplinaria de Ingeniería  
Campus Zacatecas.

**Práctica 02:** Implementación y Evaluación del Algoritmo de Dijkstra.

**MATERIA:**  
Análisis y Diseño de Algoritmos

**DOCENTE:**  
Erika Sanchez Femat

**GRUPO:**  
3CM2

**ALUMNA:**  
America Lizbet Flores Alcala

## Introduccion

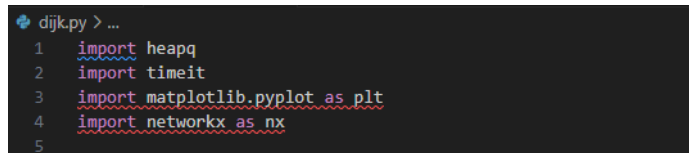
En la práctica siguiente, investigue, comprendi y aplique el algoritmo de Dijkstra, el cual sirve para encontrar los caminos más cortos desde un vértice de inicio en un grafo ponderado dirigido. Puedo decir que adquiri conocimiento sobre su funcionamiento y cómo implementarlo. Además, realicé modificaciones en el código siguiendo las pautas indicadas. Durante este proceso, explore nuevas habilidades en Python. Sin embargo, lo más destacado fue el logro de un funcionamiento correcto del código.

## Desarrollo

Ahora bien explicare un poco de lo que fue la implementacion del codigo Dijkstra

1. Como primer paso estuvo importar las librerias que utilice.

- **heapq**: Se utiliza para implementar una cola de prioridad en este algoritmo.
- **timeit**: Se utiliza para medir el tiempo que tarda en ejecutarse el algoritmo, cabe mencionar que como eran tiempos muy cortos tuve que buscar informacion y me salio esta libreria que sirve para eso.
- **matplotlib.pyplot**: Se utiliza para la visualización gráfica.
- **networkx**: Se utiliza para para la manipulación y visualización de grafos.



```
dijk.py > ...
1 import heapq
2 import timeit
3 import matplotlib.pyplot as plt
4 import networkx as nx
5
```

2. Despues implemente la clase *Graph*, la cual se contiene lo siguiente:

- Contiene su constructor (`__init__(self)`). Cuando se crea una instancia de la clase *Graph*, se inicializa con un diccionario vacío llamado grafo. Este diccionario se utiliza para almacenar la estructura del grafo, donde las claves son los vértices y los valores son diccionarios que representan los vecinos de cada vértice y los pesos de las aristas.
- `add_vertex(self, vertice)`: Este método agrega un vértice al grafo. Verifica si el vértice ya existe en el diccionario grafo antes de agregarlo. Si el vértice no existe, se agrega como clave con un diccionario vacío como su valor asociado.
- `add_edge(self, inicio, fin, peso)`: Este método agrega una arista ponderada al grafo. Verifica si tanto el vértice de inicio como el vértice final existen en el diccionario grafo. Si ambos vértices existen, agrega una entrada en el diccionario de vecinos del vértice de inicio con el vértice final como clave y el peso como valor.
- `imprimir_grafo(self)`: Este método imprime la representación del grafo en forma de diccionario. Itera a través de los vértices y sus vecinos, mostrando las conexiones.
- `visualizar_grafo(self)`: Este método utiliza la biblioteca NetworkX y Matplotlib para visualizar el grafo. Crea un grafo dirigido (*DiGraph*), agrega nodos y aristas ponderadas, y luego utiliza `nx.draw` y `plt.show()` para mostrar el grafo en un gráfico.

```

6
7 class Graph:
8     #Constructor de la clase
9     def __init__(self):
10         self.grafo = {} #Diccionario para almacenar la estructura del grafo
11
12     def add_vertex(self, vertice): #Funcion que agrega un vertice al grafo
13         if vertice not in self.grafo: #Verifica si el vertice ya existe
14             self.grafo[vertice] = {} #Si no lo agrega como clave (vacío) al diccionario
15
16     def add_edge(self, inicio, fin, peso): #Funcion que agrega una arista al grafo
17         if inicio in self.grafo and fin in self.grafo: #Verifica si el vertice ya existe
18             self.grafo[inicio][fin] = peso #Este método agrega una arista ponderada entre dos vértices del grafo siempre y cuando existan.
19         else:
20             print("¡Los vertices no existen!")
21
22     def imprimir_grafo(self):
23         for vertice, vecinos in self.grafo.items():
24             print(f"Vertice {vertice}: {vecinos}")
25
26     def visualizar_grafo(self):
27         G = nx.DiGraph()
28         for inicio, vecinos in self.grafo.items():
29             for fin, peso in vecinos.items():
30                 G.add_edge(inicio, fin, weight=peso)
31
32         pos = nx.spring_layout(G)
33         nx.draw(G, pos, with_labels=True, node_size=700, node_color="pink", font_size=8, font_color="purple", font_weight="bold", arrowsize=10)
34         etiquetas_aristas = {(inicio, fin): peso for inicio, vecinos in self.grafo.items() for fin, peso in vecinos.items()}
35         nx.draw_networkx_edge_labels(G, pos, edge_labels=etiquetas_aristas, font_color="red")
36
37         plt.show()
38
39

```

3. Como siguiente paso cree la función dijkstra, la cual implementa el algoritmo de Dijkstra para encontrar los caminos más cortos desde un vértice de inicio en un grafo ponderado dirigido. A continuacion explicare implementacion de la funcion:

Se definen las siguientes variables:

- **distancias**: Un diccionario que almacena las distancias más cortas desde el vértice de inicio hasta cada uno de los demás nodos en el grafo.
- **cola**: Una cola de prioridad implementada como una lista de tuplas. Cada tupla contiene la distancia acumulada y el nodo actual.

Despues se hace la iteracion principal y se muestra el resultado:

- **Iteracion principal del algoritmo**:
  - El algoritmo sigue iterando mientras haya nodos en la cola de prioridad.
  - En cada iteración, se extrae el nodo con la menor distancia acumulada de la cola (esto se hace utilizando `heapq.heappop` para mantener la cola como una cola de prioridad).
  - Si la distancia actual es mayor que la distancia almacenada en el diccionario distancias para el nodo actual, se omite el procesamiento de ese nodo y se continúa con el siguiente.
- **Explorar a los vecinos**:
  - Para cada vecino del nodo actual y su respectivo peso, se calcula la distancia acumulada sumando la distancia actual y el peso de la arista que los conecta.
  - Si la nueva distancia calculada es menor que la distancia almacenada en el diccionario distancias para el vecino, se actualiza la distancia y se agrega el vecino a la cola de prioridad con la nueva distancia acumulada.
- **Resultado**:
  - Una vez que la cola está vacía, se han calculado las distancias más cortas desde el vértice de inicio hasta todos los demás nodos en el grafo, y el diccionario distancias se devuelve como resultado.

```

41
42 #Implementar el metodo pedido en la practica
43 def dijkstra(grafo, vertice_inicio):
44     distancias = {nodo: float('infinity') for nodo in grafo}
45     distancias[vertice_inicio] = 0
46     cola = [(0, vertice_inicio)]
47
48     while cola:
49         distancia_actual, nodo_actual = heapq.heappop(cola)
50         if distancia_actual > distancias[nodo_actual]:
51             continue
52         for vecino, peso in grafo[nodo_actual].items():
53             distancia = distancia_actual + peso
54             if distancia < distancias[vecino]:
55                 distancias[vecino] = distancia
56                 heapq.heappush(cola, (distancia, vecino))
57
58     return distancias

```

4. Por ultimo en esta parte del código se muestra un ejemplo de uso del grafo y del algoritmo de Dijkstra

- **Creación del grafo:** Se instancia un objeto de la clase Graph llamado grafo.
- **Agregar Aristas:** Se añaden aristas ponderadas entre los vértices del grafo.
- **Imprimir el grafo:** Se utiliza el método `imprimir_grafo()` para mostrar la representación del grafo después de agregar vértices y aristas.
- **Calcular caminos mínimos desde un vértice de inicio (Dijkstra):**
  - Se inicia un temporizador (inicio) antes de ejecutar el algoritmo de Dijkstra.
  - Se especifica un vértice de inicio.
  - Se llama a la función `dijkstra` con el grafo y el vértice de inicio, y se almacena el resultado en la variable resultado.
  - Se imprime el resultado, que son las distancias más cortas desde el vértice de inicio hasta todos los demás nodos en el grafo.
  - Se detiene el temporizador (fin) y se calcula el tiempo total de ejecución.
  - **Visualizar el grafo:** Se utiliza el método `visualizar_grafo()` para mostrar el grafo utilizando NetworkX y Matplotlib.

```

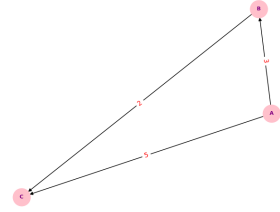
59 # Ejemplo de uso
60 grafo = Graph()
61
62 # Agregar vértices
63 grafo.add_vertex("A")
64 grafo.add_vertex("B")
65 grafo.add_vertex("C")
66
67 # Agregar aristas
68 grafo.add_edge("A", "B", 3)
69 grafo.add_edge("A", "C", 5)
70 grafo.add_edge("B", "C", 2)
71
72
73 # Imprimir el grafo
74 grafo.imprimir_grafo()
75
76 # Calcular caminos mínimos desde un vértice de inicio
77 inicio = timeit.default_timer()
78 start_vertex = 'A'
79 resultado = dijkstra(grafo.grafo, start_vertex)
80
81 print(f"Distancias más cortas desde el nodo {start_vertex}: {resultado}")
82 fin = timeit.default_timer()
83 tiempo_ejecutado = fin - inicio
84 print("Tiempo ejecutado", tiempo_ejecutado)
85
86 # Visualizar el grafo
87 grafo.visualizar_grafo()
88

```

- Casos de Ejecucion:

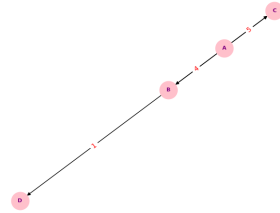
(a)

```
PS C:\Users\America Lizbet\Documents\Practica3\Dijkstra> python dijk.py
Vertice A: {'B': 3, 'C': 5}
Vertice B: {'C': 2}
Vertice C: {}
Distancias más cortas desde el nodo A: {'A': 0, 'B': 3, 'C': 5}
Tiempo ejecutado 0.00032079999880771475
```



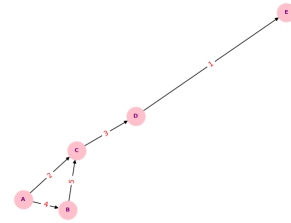
(b)

```
PS C:\Users\America Lizbet\Documents\Practica3\Dijkstra> python dijk.py
Vertice A: {'B': 4, 'C': 5}
Vertice B: {'C': 2, 'D': 1}
Vertice C: {}
Vertice D: {}
Distancias más cortas desde el nodo A: {'A': 0, 'B': 4, 'C': 5, 'D': 5}
Tiempo ejecutado 0.00038700003642588854
```



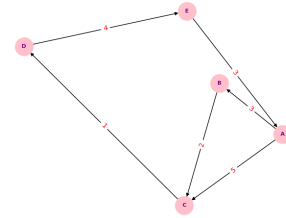
(c)

```
PS C:\Users\America Lizbet\Documents\Practica3\Dijkstra> python dijk.py
Vertice A: {'B': 4, 'C': 2}
Vertice B: {'C': 5}
Vertice C: {'D': 3}
Vertice D: {'E': 1}
Vertice E: {}
Distancias más cortas desde el nodo A: {'A': 0, 'B': 4, 'C': 2, 'D': 5, 'E': 6}
Tiempo ejecutado 0.00038049998693168163
```



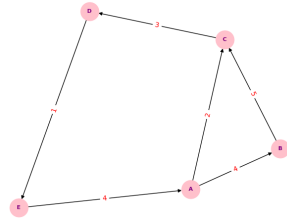
(d)

```
PS C:\Users\America Lizbet\Documents\Practica3\Dijkstra> python dijk.py
Vertice A: {'B': 3, 'C': 5}
Vertice B: {'C': 2}
Vertice C: {'D': 1}
Vertice D: {'E': 4}
Vertice E: {'A': 3}
Distancias más cortas desde el nodo A: {'A': 0, 'B': 3, 'C': 5, 'D': 6, 'E': 10}
Tiempo ejecutado 0.00036339997313916683
```



(e)

```
PS C:\Users\America Lizbet\Documents\Practica3\Dijkstra> python dijk.py
Vertice A: {'B': 4, 'C': 2}
Vertice B: {'C': 5}
Vertice C: {'D': 3}
Vertice D: {'E': 1}
Vertice E: {'A': 4}
Distancias más cortas desde el nodo A: {'A': 0, 'B': 4, 'C': 2, 'D': 5, 'E': 6}
Tiempo ejecutado 0.0003772000200115144
```



- **Complejidad BigO:**  $O((V + E)\log V)$ , donde V es el número de vértices y E es el número de aristas en el grafo

## Conclusion

En conclusion, la experiencia de trabajar con el algoritmo Dijkstra ha sido muy satisfactoria.

Aprovechando la base previa que había adquirido en clases anteriores, pude entender sus fundamentos de manera más sólida. No obstante, la implementación práctica del código presentó algunos desafíos. La tarea de traducir mi comprensión teórica en un código funcional me expuso a desafíos comunes en programación. A pesar de estos obstáculos, logré implementar el algoritmo de manera eficiente. Esta experiencia me permitió no solo reforzar mis habilidades de programación, sino también ganar una apreciación más profunda de la eficiencia de Dijkstra como un algoritmo para encontrar los caminos más cortos desde un vértice de inicio. A pesar de los desafíos, la sensación de logro al hacer que el algoritmo funcione correctamente es una bonita recompensa.

## Referencias

[https://www.freecodecamp.org/espanol/news/  
algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/](https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/)  
[https://runestone.academy/ns/books/published/pythoned/Graphs/  
AnalisisDelAlgoritmoDeDijkstra.html](https://runestone.academy/ns/books/published/pythoned/Graphs/AnalisisDelAlgoritmoDeDijkstra.html)  
<https://www.delftstack.com/es/howto/python/dijkstra-algorithm-python/>