



**Instituto Politecnico Nacional**

Unidad Profecional Interdiciplinaria de  
Ingenieria Campus Zacatecas

December 2, 2023

## **Ingenieria en Sistemas Computacionales**

**Practica 3:**Implementacion y evaluacion del  
algoritmo de Dijkstra

**Maestra:**

M. en C. Erika Sanchez Femat

**Materia:**

Analisis y Diseño de Algoritmos

**Alumno:**

Alejandro Ulloa Reyes

**Grupo:**

3CM2

# 1 Introduction

El algoritmo de dijkstra se denominaria como un algoritmo voraz ya que busca llegar a su objetivo de la manera mas rapida posible, uno puede ponerle ciertas restricciones para que las cumpla al momento de llegar a su objetivo, pero concisamente el algoritmo de dijkstra consta de un grafo en el que existen varios nodos y aristas con un determinado peso, para moverse entre estos nodos tambien esta la regla de que existe un sentido por el cual se llega al nodo o arista este puede ser de un solo sentido o doble sentido, el nodo puede interconectarse con mas de un nodo o arista eso depende del grafo que se tenga.

Al momento de inicializar el algoritmo se empieza en un nodo con una distancia inicial de 0 y los demas nodos aun sin saber su ponderacion real tendran un distancia minima de infinito, a continuacion analizaremos los nodos vecinos a nuestro nodo inicial para determinar cual sera el camino mas corto para llegar al nodo destino, despues de verificar cada nodo vecino y de que hayamos encontrado su ponderacion real que sabemos ahora no es infinito escojaremos el nodo con la ponderacion mas baja para trasladarnos, aqui empezara a repetirse los mismos pasos que antes pero con la carga extra del recorrido que se ha hecho hasta ahora, cada que se requiera mover hacia un nodo la ponderacion se tendra que ir sumando y siempre visitara los nodos aledaños primaro para descartar la ponderacion infinita que se tenia en un principio, finalmente llegaremos al nodo destino.

# 2 Implementacion

El constructor de la clase Grafo `__init__(self)` inicializa los tres atributos: vertices, aristas (`AdamC`) y distancias. Los vértices del grafo se guardan en un conjunto, las aristas se guardan en un diccionario, y las distancias se guardan en un diccionario que registra las distancias mínimas desde el vértice de inicio.

```
import heapq
import time

class Grafo:
    def __init__(self):
        self.vertex = set()
        self.AdamC = {} #aristas :)
        self.distancias = {}
```

Figure 1: Clase grafo y constructor

En `add_vertex(self, valor)` se utiliza para agregar un vértice al grafo. Se agrega el vértice al conjunto de vertices, y se inicializa su distancia en infinito en el diccionario de distancias.

```
def __init__(self):
    self.distancias = {}

def add_vertex(self, valor):
    self.vertex.add(valor)
    self.distancias[valor] = float('inf')
    self.AdamC[valor] = []
```

Figure 2: Funcion para vertices

En `add_edge(self, desde, hacia, peso)` agregaremos los parametros de `desde`, `hacia`, `peso` en donde `desde` es el vertice donde empieza la arista, `hacia` es el vertice donde llega la arista y finalmente `peso` es la ponderacion que hay de `desde` hasta `hacia`.

Apartir de ahí accederemos a la lista de aristas salientes desde el vértice con `self.aristas[desde]` se lanzará un error si el vértice desde aún no existe en el diccionario de aristas, por lo que debe agregarse primero utilizando `add_vertex`.

`add_edge` se encargara de ampliar la estructura del grafo al incluir datos sobre aristas dirigidas y ponderadas. Esta función permitira especificar cómo los vértices están conectados entre sí y la ponderación asociada con esas conexiones a medida que se construye el grafo.

```
def add_edge(self, desde, hacia, peso):
    self.AdamC[desde].append((hacia, peso))
```

Figure 3: Funcion para aristas

Para `def dijkstra(self, inicio)` se creara un diccionario de distancias que asigna una distancia inicial de infinito a cada vértice (`float('inf')`). Luego, se establece la distancia entre el vértice de inicio y sí mismo en cero porque la distancia entre ambos es cero, despues la cola de prioridad (`heap`) se inicia con una tupla `(0, inicio)`, donde 0 es la distancia acumulada desde el vértice de inicio hasta el momento. Las tuplas con las distancias más pequeñas están en la parte superior del heap.

El bucle principal se ejecuta mientras haya elementos en el heap, mientras que en cada iteración, se extrae la tupla con la distancia mínima y el vértice actual del heap usando `heappq.heappop(heap)`. Se verifica si la distancia extraída (`dist`) es mayor que la distancia registrada actual para el vértice actual (`self.distancias[actual]`). Si es así, significa que ya se ha descubierto una solución más rápida y que este vértice no se está procesando.

Para explorar los nodos aldeanos se iterara apartir del nodo actual con `(for vecino, peso in self.aristas[actual]):` y se calculara la nueva distancia acumulada hasta el vecino (`nueva_distancia`) sumando la distancia actual (`dist`) y el peso de la arista que lleva al vecino. La nueva distancia se actualiza y se agrega una tupla al heap con la nueva distancia y el vecino si la nueva distancia es menor que la distancia registrada para el vecino (`self.distancias[vecino]`). Al final del bucle, el diccionario `self.distancias` contiene las distancias mínimas desde el vértice de inicio hasta todos los demás vértices alcanzables en el grafo.

```
def dijkstra(self, inicio):
    self.distancias = {vertex: float('inf') for vertex in self.vertex}
    self.distancias[inicio] = 0
    heap = [(0, inicio)]

    while heap:
        (dist, actual) = heappq.heappop(heap)

        if dist > self.distancias[actual]:
            continue

        for vecino, peso in self.AdamC[actual]:
            nueva_distancia = dist + peso

            if nueva_distancia < self.distancias[vecino]:
                self.distancias[vecino] = nueva_distancia
                heappq.heappush(heap, (nueva_distancia, vecino))
```

Figure 4: Funcion que implementa el algoritmo de Dijkstra

Finalmente en `def main()` se crea una instancia de la clase Grafo y se agregan vértices y aristas de ejemplo para probar el funcionamiento del algoritmo, se define un vertice de inicio, se mide el tiempo de ejecucion antes y despues de haber llamado `dijkstra` con la funcion `time.time()` para finalizar se imprimen las distancias desde el punto de inicio y el tiempo que le tomo realizar el algoritmo.

```

def main():
    grafo = Grafo()
    grafo.add_vertex('A')
    grafo.add_vertex('B')
    grafo.add_vertex('C')
    grafo.add_vertex('D')

    grafo.add_edge('A', 'B', 5)
    grafo.add_edge('B', 'C', 8)
    grafo.add_edge('C', 'D', 3)
    grafo.add_edge('A', 'D', 7)

    inicio = 'A'

    start_time = time.time()
    grafo.dijkstra(inicio)
    end_time = time.time()

    print(f"Distancias mínimas desde el vértice {inicio}: {grafo.distancias}")
    print(f"Tiempo de ejecución: {end_time - start_time} segundos")

if __name__ == "__main__":
    main()

```

Figure 5: Funcion main que define el grafo y los pesos entre los vertices

### 3 Análisis del algoritmo de Dijkstra

En la medición de tiempos se compilo 5 veces con los mismos pesos, despues se cambiaron los pesos otras 2 veces y se volvio a compilar 5 veces mas cada cambio, para asi tener un promedio de los tiempos en cada prueba. A continuacion se mostrara una tabla con los tiempos en cada ocacion.

# Compilacion	Tiempo(s)
1	1.0967
2	2.9087
3	1.4781
4	1.4543
5	1.5735
Promedio	1.7022

Table 1: Ponderaciones de: A,B=5, B,C=8, C,D=3, A,D=7

# Compilacion	Tiempo(s)
1	1.5020
2	1.3589
3	1.5497
4	1.1444
5	9.7751
Promedio	3.066

Table 2: Ponderaciones de: A,B=4, B,C=15, C,D=7, A,D=2

# Compilacion	Tiempo(s)
1	1.2874
2	1.5497
3	1.6927
4	1.9311
5	1.4066
Promedio	1.5735

Table 3: Ponderaciones de: A,B=30, B,C=48, C,D=27, A,D=72

La complejidad del algoritmo de Djikstra depende de la estructura de datos utilizada para implementar la cola de prioridad. Para mantener la cola de prioridad en la implementación anterior, se utiliza un heap binario. Esto da como resultado una complejidad de tiempo de  $O((V + E)\log V)$ , donde V es el número de vértices y E es el número de aristas en el grafo. La cola de prioridad y la inicialización de las distancias toman  $O(V + E * \log V)$ . Esto se debe a la inserción de las distancias iniciales en el heap y la creación del diccionario de distancias. El bucle principal se ejecuta V veces. Se realiza una operación de extracción de cola de prioridad (`heapq.heappop(heap)`) en cada iteración, que toma  $O(\log V)$ . El número total de iteraciones sobre los vecinos de cada vértice es E. Se realiza una operación de inserción en el heap (`heapq.heappush(heap, (nueva_distancia, vecino))`) en cada iteración, que consume  $O(\log V)$ .

Es importante tener en cuenta que esta complejidad presupone que el heap binario es eficiente, lo cual es normalmente el caso en la práctica. Además, la complejidad puede reducirse a  $O(E \log V)$  en grafos dispersos (donde E es mucho menor que  $V^2$ ).

## 4 Conclusiones

Como práctica de algoritmos voraces podemos decir que Dijkstra es un ejemplo perfecto de como este trabaja para encontrar las rutas más cortas incluyendo cada parámetro impuesto ya que sin estos podríamos decir que si llegaría al punto deseado pero rompería las reglas. Un detalle que quiero suponer depende de la máquina en la que se compila el programa es el hecho de que algunos tiempos tomaron demasiado en compilar (cuestión de algunos segundos más) pero es algo que me resulta interesante.

## References

Algoritmo De Dijkstra. (2013, 26 junio). MaxAG. <https://maximilianoag.wordpress.com/unidad-3/trabajo-de-investigacion-dijktra/>

Coding games and programming challenges to code better. (s. f.). CodinGame. <https://www.codingame.com/playgrounds/caminos-mas-cortos-con-el-algoritmo-de-dijkstra/el-algoritmo-de-dijkstra>