



Instituto Politécnico Nacional

INGENIERÍA EN SISTEMAS COMPUTACIONALES

IMPLEMENTACIÓN Y EVALUACIÓN DEL ALGORITMO DE DIJKSTRA

Análisis y diseño de algoritmos

Maestra:

Erika Sanchez-Femat

Autor:

Jose Martin Cortes Lozano

3CM1

1 de Diciembre del 2023

Introducción

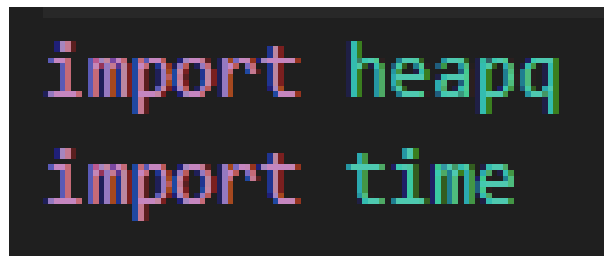
En este trabajo se implementa el algoritmo de Dijkstra en python, además se evalúa su tiempo de ejecución para grafos dirigidos de diferente cantidad de vértices y aristas, igualmente se evalúa la complejidad $\text{Big}(O)$ del algoritmo.

Con el algoritmo de Dijkstra, puedes encontrar la ruta más corta o el camino más corto entre los nodos de un grafo. Específicamente, puedes encontrar el camino más corto desde un nodo (llamado el nodo de origen) a todos los otros nodos del grafo, generando un árbol del camino más corto.

Este algoritmo es usado por los dispositivos GPS para encontrar el camino más corto entre la ubicación actual y el destino del usuario. Tiene amplias aplicaciones en la industria, especialmente en aquellas áreas que requieren modelar redes.

Desarrollo

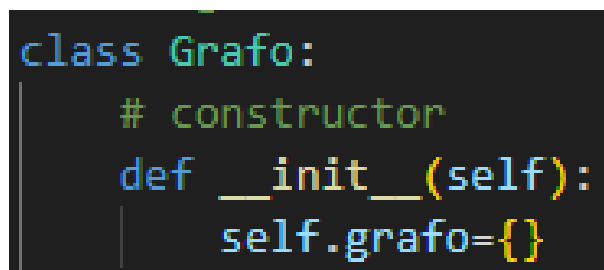
Primero se importaron dos librerías para usarlas en el código como se observa en la figura 2. Donde la librería `time` sirve para medir el tiempo y la librería `heapq` servirá para hacer un árbol donde se ingresarán los datos del grafo cuando se vaya a calcular el camino mas corto.



```
import heapq
import time
```

Figura 2: Librería

Después de esto se creó una clase llamada `grafo`, en ella, primero se hizo el constructor que representa un objeto de la clase, por lo tanto dentro del constructor se creó un diccionario vacío para meter allí los datos de cada vértice que hay en el grafo como lo demuestra la figura 3.



```
class Grafo:
    # constructor
    def __init__(self):
        self.grafo={}
```

Figura 3: Clase y Constructor

Dentro de la misma clase se creó una función que se llamó `agregar vertice` que toma como parámetro un vertice, su principal función es que que ingresa el vertice al grafo que fue creado en el constructor, primero verifica si el grafo esta vacío, si es así, ingresa el vertice en el grafo. En caso de que el grafo ya tenga datos, primero verifica si el dato no está en el grafo, si no está, lo añade, si el dato está en el grafo solo muestra un mensaje, eso para evitar que se tengan vértices repetidas en el grafo, esto se observa en la figura 4.

```
def agregar_vertice(self, vertice):
    if self.grafo == None:
        self.grafo[vertice]={}
    else:
        if vertice not in self.grafo:
            self.grafo[vertice]={}
        else:
            print(f'{vertice} es un vertice repetido')
```

Figura 4: Agregar vértices

La ultima función dentro de la clase se llama agregar arista, esta función pide como parámetros un vertice de inicio, uno final y un peso entre ambos vértices, primero se verifica que el vertice inicio y final no estén en el grafo, si no están, se añaden al grafo y después se añaden como una tupla que tiene un valor determinado que es el peso. En caso de que ya estén en el grafo, solo resta añadirlos como una tupla con su peso correspondiente como se muestra en la figura 5

```
def agregar_arista(self, inicio, final, peso):
    if inicio and final not in self.grafo:
        self.agregar_vertice(inicio)
        self.agregar_vertice(final)
        self.grafo[inicio][final]=peso
    else:
        self.grafo[inicio][final]=peso
```

Figura 5: Agregar arista

A fuera de la clase, se creó una función que se llamó diji, y su función es obtener el camino mínimo de un grafo determinado usando el algoritmo de Dijkstra.

Para la función diji, se toman como parámetros el grafo creado y un vertice que sera el punto de inicio para calcular el camino mínimo, se inicializan los datos del grafo, en este caso para la primera linea se inicializan cada vertice del grafo con un valor de 9999, esto es para tener un limite de peso y poder calcular el peso mínimo mas fácilmente.

Para la segunda linea del código se inicializa el valor del inicio con 0, esto es por que al ser el primer vertice en ser analizado , es actualmente el camino mas corto, con un valor de 0, así que en cuanto se analicen los demás vértices su valor ira cambiando.

En la tercera linea, se crea una cola que es una cola de prioridad, donde su prioridad va a ser el peso mas pequeño de cada camino.

Estas lineas de código se muestran en la siguiente figura 6

```
def diji (graph, ini):
    distancias = {vertice: 9999 for vertice in graph}
    distancias[ini]=0
    cola=[(0,ini)]
```

Figura 6: Distancias

Luego de esto se inicia un while donde solo se va a detener cuando la cola este vacía, en este sentido se declaran dos variables , una se llama dActual que es la distancia actual del vertice y vActual que es el vertice actual en el que se encuentra la distancia mencionada, estos vértices se ingresan a la cola mediante la función que fue importada al inicio, heapop, que recibe como parámetro a la cola, y lo que hace ver el dato actual y acomodarlo según su prioridad, en este caso es el peso menor.

Después, el if que esta dentro del while, analiza si la distancia actual es mayor al diccionario de distancias en el vertice actual y verifica que el mismo diccionario sea diferente al valor que fue declarado anteriormente, en caso de que pase esto se pasa a la siguiente iteración, como lo muestra la figura 7.

```

while cola:
    dActual, vActual = heapq.heappop(cola)
    if dActual > distancias[vActual] and distancias[vActual] != 9999:
        continue

```

Figura 7: Iteración del while

Enseguida del if y aun dentro del while, se hace un ciclo for donde se declara una variable llamada vecino, y se usa el peso de las aristas para sacar las tuplas de datos que hay en el diccionario usando la función items(), después a la distancia se iguala al vertice actual y al peso, dentro del ciclo for se hace una iteración if donde si la distancia de vecino es igual a 9999 o el diccionario de distancia es menor a las distancias en vecino, se actualiza la distancia de vecino al diccionario de distancias y después con el método de heappush se actualiza la distancia en la cola con el valor que hay en vecino, como lo muestra la siguiente figura 8.

```

continue
for vecino, peso in graph[vActual].items():
    distancia = dActual + peso
    if distancias[vecino] == 9999 or distancia < distancias[vecino]:
        distancias[vecino] = distancia
        heapq.heappush(cola, (distancia, vecino))

```

Figura 8: Ciclo for

Y finalmente se retorna el diccionario de distancias con los caminos mínimos desde el vertice de inicio. Esto se aprecia en la figura 9.

```

return distancias

```

Figura 9: Distancia final

Enseguida del algoritmo de Dijkstra se hizo una función para medir el tiempo llamada tiempos, en esa función toma como parámetro una f que es la función de Dijkstra, luego toma el valor del tiempo inicial en start, luego se inicializa la función a una variable, y después se toma el tiempo final en end. Finalmente se toma el tiempo de ejecución restando $end - start$ para después retornar ese valor. Vista en la figura 10,

```

def tiempos(f):
    start=time.time()
    funcion=f
    end=time.time()
    ejecucion=end-start
    return ejecucion

```

Figura 10: Tiempos

Resultados

La complejidad se calculo mediante dos ejemplos, donde se uso un árbol para llegar a camino mínimo y dando como resultado $O((V + A) \log(V))$, donde V y A son los vértices y aristas respectivamente, este análisis se muestra en la figura 11.

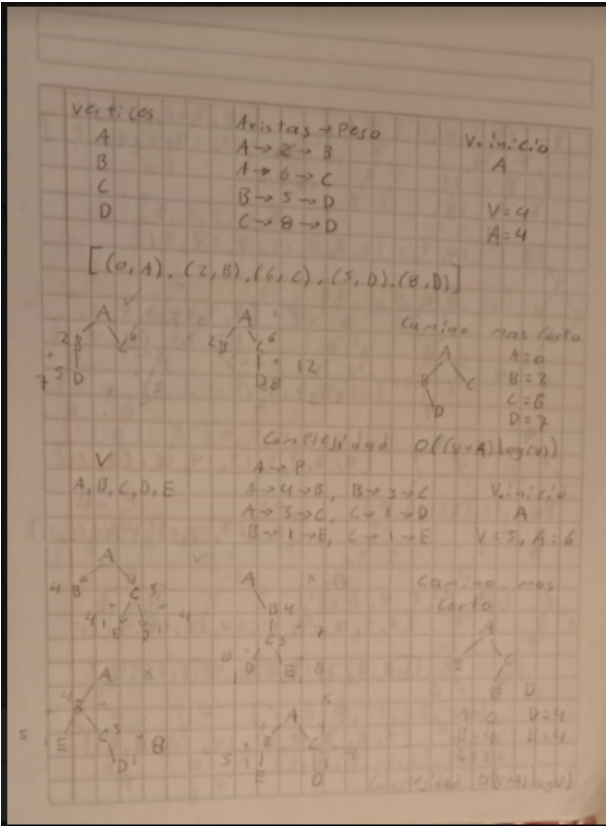


Figura 11: Complejidad *BigO*

La implementación 1 se hizo del siguiente grafo en la figura 12.

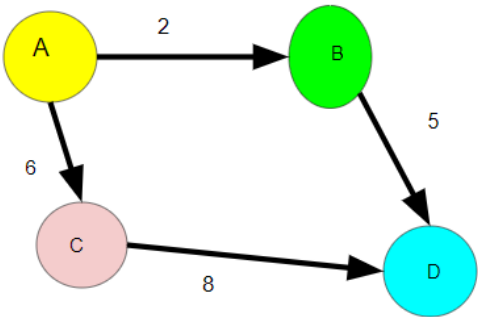


Figura 12: Ejemplo 1

Es un grafo sencillo donde solo hay 4 vértices, se tomo como el vertice de inicio "a" y dio los siguientes resultados en la figura 13

```
{'a': {'b': 2, 'c': 6}, 'b': {'d': 5}, 'c': {'d': 8}, 'd': {}}
distancias minimas desde a
a: 0
b: 2
c: 6
d: 7
Tiempo de ejecucion: 0.0
```

Figura 13: Implementación 1

La implementación 2 se hizo del siguiente grafo que se muestra en la figura 14.

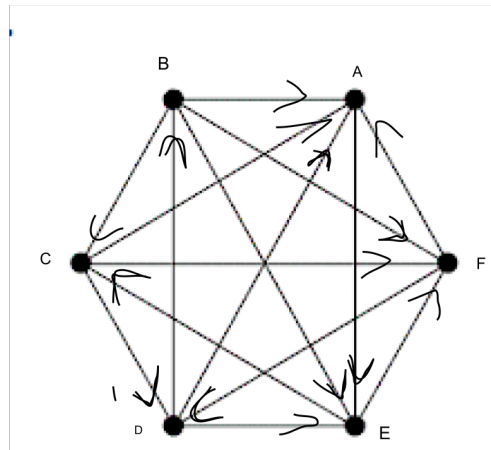


Figura 14: Ejemplo 2

Este grafo es un poco mas complicado, ya que tiene 6 vértices y tiene caminos desde todos los vértices, se les asignó un peso para cada arista y se tomo como vertice de inicio "a", se observa en la figura 15.

```

g2.agregar_arista("b","a",5)
g2.agregar_arista("b","c",6)
g2.agregar_arista("b","e",10)
g2.agregar_arista("b","f",8)
g2.agregar_arista("a","e",3)
g2.agregar_arista("c","f",12)
g2.agregar_arista("c","d",10)
g2.agregar_arista("c","a",16)
g2.agregar_arista("d","b",4)
g2.agregar_arista("d","e",6)
g2.agregar_arista("d","a",3)
g2.agregar_arista("e","c",9)
g2.agregar_arista("e","f",12)
g2.agregar_arista("f","d",16)
g2.agregar_arista("f","a",17)
print(g2.grafo)
inicio="a"

```

Figura 15: Pesos entre aristas

Por lo que dio los siguientes resultados en la figura 16.

```

{'a': {'e': 3}, 'b': {'a': 5, 'c': 6, 'e': 10, 'f': 8}, 'c': {'f': 12, 'd': 10, 'a': 16}, 'd': {'b': 4, 'e': 6, 'a': 3}, 'e': {'c': 9, 'f': 12}, 'f': {'d': 16, 'a': 17}}
distancias mínimas desde a
a: 0
b: 26
c: 12
d: 22
e: 3
f: 15
Tiempo de ejecución: 0.0

```

Figura 16: Implementación 2

La implementación 3 se hizo del siguiente grafo en la figura 17

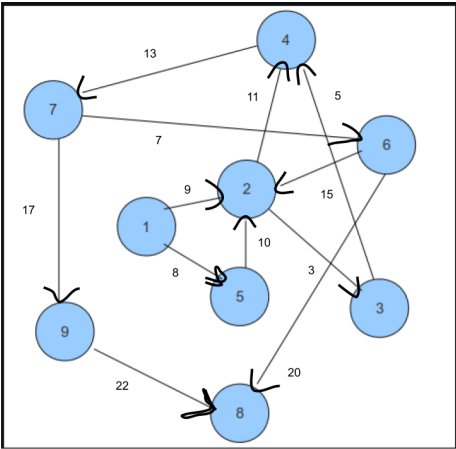


Figura 17: Ejemplo 3

Este grafo tiene 9 aristas y tiene una forma, por llamarlo de algún modo, un tanto extraña, luego se

ingresó los pesos a las aristas como se muestra en 18.

```
g.agregar_arista(1,2,9)
g.agregar_arista(1,5,8)
g.agregar_arista(2,4,11)
g.agregar_arista(2,3,3)
g.agregar_arista(3,4,15)
g.agregar_arista(4,7,13)
g.agregar_arista(5,2,10)
g.agregar_arista(6,2,15)
g.agregar_arista(6,8,20)
g.agregar_arista(7,6,7)
g.agregar_arista(7,9,17)
g.agregar_arista(9,8,22)

print(g.grafo)
inicio=1
```

Figura 18: Pesos 3

Y dio los siguientes resultados en la figura 19

```
distancias minimas desde 1
1: 0
2: 9
3: 12
4: 20
5: 8
6: 40
7: 33
8: 60
9: 50
```

Figura 19: Implementación 3

La implementación 4 se hizo del siguiente grafo en la figura 20.

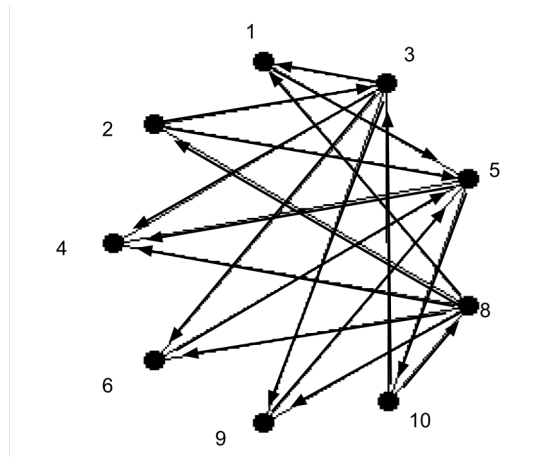


Figura 20: Ejemplo 4

Este es un grafo con 10 vértices y muchas aristas, luego se le asignó un peso a cada camino como se observa en la figura 21.

```
g3.agregar_arista("1","5",5)
g3.agregar_arista("2","3",6)
g3.agregar_arista("2","5",3)
g3.agregar_arista("3","1",7)
g3.agregar_arista("3","4",8)
g3.agregar_arista("3","6",4)
g3.agregar_arista("3","9",12)
g3.agregar_arista("5","4",10)
g3.agregar_arista("6","5",13)
g3.agregar_arista("8","1",4)
g3.agregar_arista("8","2",6)
g3.agregar_arista("8","4",9)
g3.agregar_arista("8","6",7)
g3.agregar_arista("9","5",16)
g3.agregar_arista("10","8",15)
g3.agregar_arista("10","3",17)
print(g3.grafo)
inicio="8"
d3=diji(g3.grafo,inicio)
```

Figura 21: Pesos 4

Los resultados obtenidos se observan en 22. En este caso, el valor del vertice 10=9999, es así porque a mi entender no hay ningún camino que llegue desde el inicio de 8 y hasta el final 10, por lo tanto que da el valor asignado en el código de Dijkstra como máximo.

```

distancias minimas desde 8
1: 4
2: 6
3: 12
4: 9
5: 9
6: 7
8: 0
9: 24
10: 9999
Tiempo de ejecucion: 0.0

```

Figura 22: Implementación 4

La implementación 5 se hizo del siguiente grafo en la figura 23.

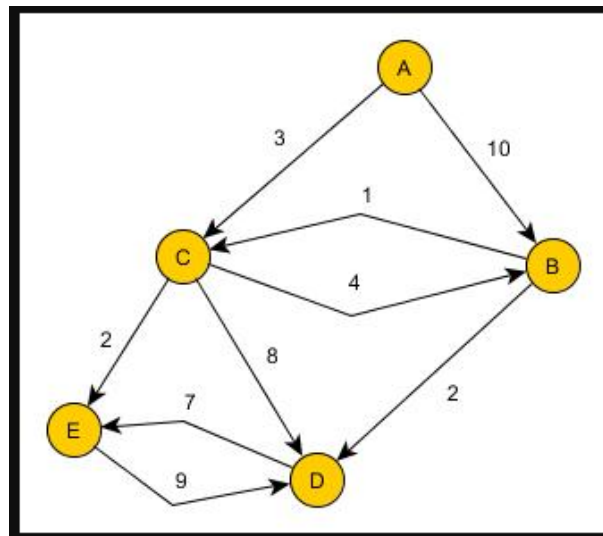


Figura 23: Ejemplo 5

Este grafo parece el mas complicado de todos porque esta en tercera dimension, pero el algoritmo funciona igual sin importar la forma del grafo. Después de ingresar los pesos al código, se ejecutó el programa y dio los siguientes resultados, que se observan en la figura 24.

```
distancias minimas desde a
a: 0
b: 7
c: 3
d: 9
e: 5
Tiempo de ejecucion: 0.0
```

Figura 24: Implementación 5

En todos los ejemplos, los tiempos dan 0, eso significa que para la computadora, ejecuta el algoritmo en un instante, ese instante es tan pequeño que es como si no hubiera pasado de tiempo.

Conclusión

Puedo concluir que este algoritmo es muy útil a la hora de hacer mapas, de encontrar el camino mas óptimo de manera sencilla, además al tener una complejidad $O((V + A) \log(V))$ y ser un algoritmo voraz, tiene una complejidad aceptable.

Igualmente, al implementar el código se aplicaron conceptos que se vieron en clase como el heap. Del mismo modo, al calcular la complejidad manualmente, se usaron métodos similares a los usados en clase.

Referencias

- Navone, E. C. (2023, 2 agosto). Algoritmo de la ruta más corta de Dijkstra - Introducción gráfica y detallada. freeCodeCamp.org. <https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/>
- Greyrat, R. (2022, 5 julio). HEAP y Priority Queue usando el módulo HEAPQ en Python – Barcelona Geeks. <https://barcelonageeks.com/heap-y-priority-queue-usando-el-modulo-heapq-en-python/>
- ¿Cómo usar el método Python Items () - Programador Clic. (s. f.). <https://programmerclick.com/article/2710824767/>
- HEAPQ — HEAP Queue Algorithm. (s. f.). Python documentation. <https://docs.python.org/3/library/heapq.html>