



Practica3: Algoritmo de Dijkstra

Autor: Daniel Juárez Ávila

Materia: Análisis y diseño de Algoritmos

Zacatecas, Zac. a 1 de diciembre de 2023

1. Introducción

La teoría de grafos desempeña un papel crucial en diversos campos, proporcionando un marco sólido para modelar y analizar las relaciones entre entidades. En este documento, nos sumergimos en la implementación del algoritmo de Dijkstra, un algoritmo fundamental en la teoría de grafos utilizado para encontrar los caminos más cortos entre nodos en un grafo. El algoritmo explora diversos caminos, asignando pesos a las aristas y determinando la manera más eficiente de llegar a un destino desde un punto de inicio dado. A través de una explicación y análisis detallados, buscamos arrojar luz sobre las complejidades del algoritmo de Dijkstra y su aplicación en la resolución de problemas del mundo real.

2. Desarrollo

2.1. Procedimiento

Para la implementación en código, primero creamos una clase llamada "Graph", la cual tendrá como datos una lista de vértices y una lista de aristas. Para poder añadir los vértices se crea un método, el cual, recibe como parámetro un valor de tipo String el cual será el nombre de nuestro vértice este valor de String se añadira a la lista de vértices. Para poder agregar una arista se necesitará un método que reciba como parámetros: el vértice de salida, el vértice final y una ponderación, para poder agregar una arista hace falta evaluar si los dos vértices de entrada existen en la lista que contiene los vértices del grafo.

En este caso primero evaluamos el caso de cuando los dos vértices existen, por lo cual, se agregan los tres valores a la lista de aristas, los tres valores van en una lista por lo cual, nuestra lista de aristas es una multilista.

```
sw1 = False
sw2 = False
i = 0
while sw1 == False and i < len(self.vertex):
    sw1 = True if self.vertex[i] == vertex1 else False
    i = i + 1
i = 0
while sw2 == False and i < len(self.vertex):
    sw2 = True if self.vertex[i] == vertex2 else False
    i = i + 1
if sw1 and sw2:
    self.edges.append([vertex1, vertex2, pond])
elif sw1 == False:
    print(f"El vértice {vertex1} no existe")
else:
    print(f"El vértice {vertex2} no existe")
```

Para implementar el algoritmo de Dijkstra hicimos uso de la forma que tiene el procedimiento como tabla así como de la recursividad en programación. Primero vamos a tener un método llamado "*dijkstra1*" el cual se encarga de organizar las aristas en una multilista para así simular la tabla de el procedimiento en Dijkstra, ya con una multilista organizada retornamos un método llamado "*dijkstra2*" (el cual definiremos mas adelante), mandando como parámetro la multilista, el vértice de inicio, el vértice final y un número 1 el cual nos servirá mas adelante para saber cuantos pasos llevamos.

En el método "*dijkstra2*" vamos a aplicar la recursividad en el algoritmo de Dijkstra, este programa funciona de la siguiente manera:

Empezamos con un vertice de inicio en el cual vamos a revisar todos los vertices a los cuales puede ir, en la multilista tenemos una sección para cada vertice en la cual, a su vez, esta relacionada con una multilista que guarda listas de dos datos, el primero es el vertice del cual venimos y el segundo la ponderación total que llevamos por ese camino.

Ejemplo: `[['A', ['B', 5]]]`

Ahora que ya tenemos todas los caminos posibles para el primer vertice, ahora el siguiente paso es eliminar de la lista el vertice que estamos usando como inicio, ya que para el algoritmo de Dijkstra no podemos volver por un camino que ya tomamos por lo cual lo sacamos de la lista, así nuestra lista se va a ir reduciendo cada vez mas. Para elegir el siguiente vertice por el cual vamos a pasar tenemos que elegir el cual lleva la ponderación mas pequeña hasta el momento, de ese valor se va a tomar el proximo vertice que se va a usar y se va a llamar de maner recursiva a la función `"dijkstra2"` para calcular el siguiente paso, mandando como parametros la tabla modificada, el pivote(proximo vertice a tomar) como inicio, el vertice final y al contador que recibimos con el numero 1 lo vamos a aumentar en 1 para mostrarque avanzamos al siguiente paso.

Cuando estamos en el segundo paso en adelante vamos a tener dos modificaciones. La primera es que cuando revisamos los vertices a los cuales podemos ir, la multilista no se va a actualizar, en vez de esto se va a usar un append para meter otro paso, ejemplo: `[['A', ['B', 5], ['C', 10]]]`, y cuando no existe un vertice para hacer referencia se tomara como estaba en el pasó anterior esto con el fin de que todas las multilistas conserven el mismo tamaño y que no se pierda ningun camino posible. La segunda diferencia es que vamos a tener que evaluar si el paso en el que estamos es el mas optimo, ya que el pasó anterior puede que tenga una ponderación menor, cuando esto sucede tenemos que reemplazar el valor actual por el anterior, esto se debe a que irse por la arista mas corta no siempre te lleva al camino con menor ponderación por lo cual hay que verificar si existe otro camino que conlleva menor ponderación para llegar al mismo vertice.

```

if c > 1:
    for i in range(0, len(tab)):
        if tab[i][c][1] > tab[i][c-1][1] and tab[i][c-1][1]!=0:
            tab[i][c][1] = tab[i][c-1][1]
            tab[i][c][0] = tab[i][c-1][0]

```

Finalmente, despues de todos estos pasos podemos llegar a nuestro caso base, el cual va a ser cuando el vertice que elegimos es justo el vertice al que queremos llegar, esto se puede ver en el programa como `start == end`. Cuando este caso aparezca es porque ya hemos verificado los caminos posibles hasta ese vertice por lo cual tenemos que tomar todas las posibles ponderaciones, diferentes de cero, de el vertice final y verificar cual es la mas pequeña, esto nos estaría dando el valor minimo y es el valor que vamos retornar.

2.2. Complejidad

En el mejor de los casos vamos a tener solamente dos vertices relacionados por una arista, por lo cual primero comprobaría todos los caminos posibles para el primer vertice, el cual es solo uno, para despues ir al caso base el cual retornara la unica ponderación que contiene. El mejor de los casos podría tener una complejidad de $O(n)$ ya que no genera muchos procedimientos.

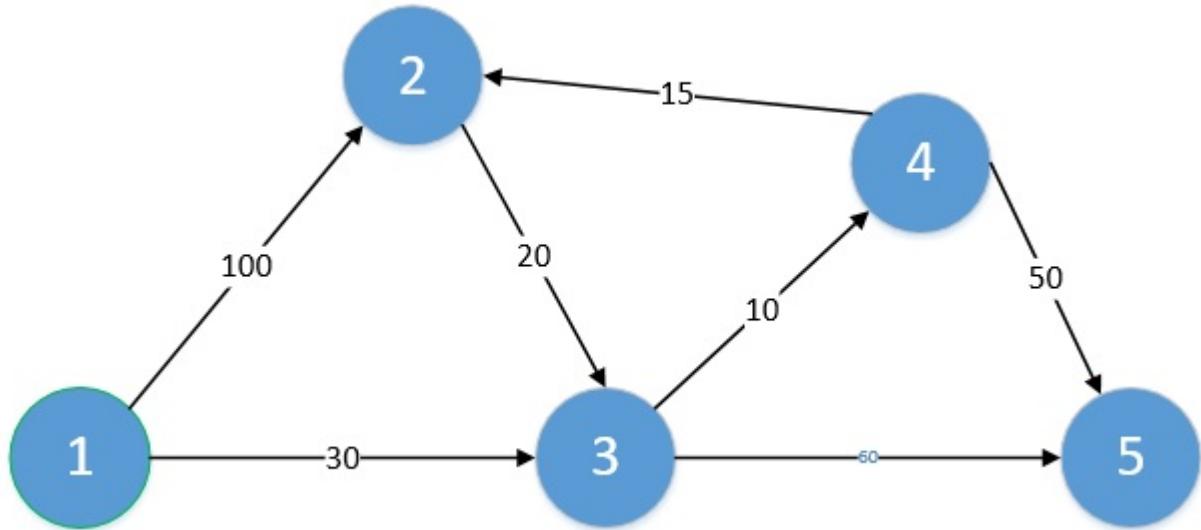
Esto puede cambiar para un peor caso en el que dispongamos de multiples vertices y todos esten relacionados entre sí, este caso haría que por cada vertice tengamos que analizar cada camino con todos los vertices restantes. Por lo cual si tuvieramos 5 vertices el primero se compararía con 4 el segundo con 3 el tercero con dos y así sucesivamente. Dando como resultado $5!$ de comparaciones entre vertices.

Para saber la complejidad se debe de tomar en cuenta el numero de vertices así como el numero de aristas, ya que por mas vertices que tengamos, si no estan relacionados no generarán demasiadas operaciones, sin embargo en la mayoría de los casos este algoritmo nos da una complejidad de $O(n^2)$

esto debido a la cantidad de for, while y tomando en cuenta que se trata de un algoritmo recursivo por lo que toma diferentes cantidad de pasos según el caso específico.

3. Resultados

3.1. Ejemplo 1

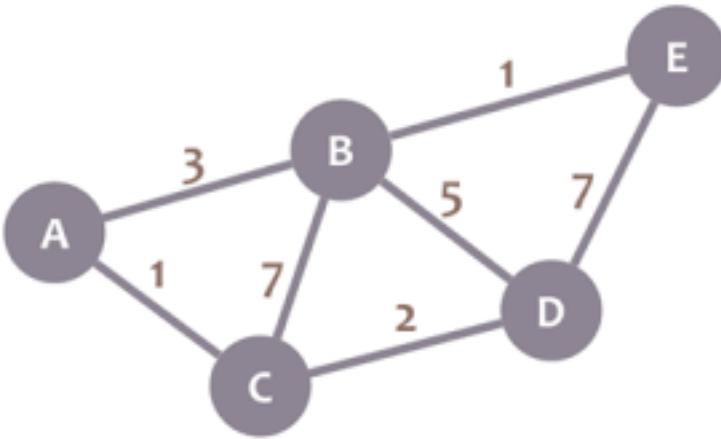


En este ejemplo queremos saber el camino mas corto entre A y E por lo cual empezamos comparando a cuales vertices podemos ir desde A, tenemos como posibilidad a B con 100 de ponderación y C con 30 de ponderación por lo cual nos vamos por C ya que tiene la ponderación mas baja, ya estando en C podemos ir para D con 40 de camino total y para E con 90 de camino total, entonces nos vamos al vertice D, en D podemos ir al vertice B con 55 de camino total o a E con 90 de camino total. En este caso tenemos dos opciones con la misma ponderacion minima que es llegar por D a E y llegar por C a E, en este programa toma la opción de irse por D a E e imprime la ponderación minima de 90

```

[[['A', ['', 0]], ['B', ['A', 100]], ['C', ['A', 30]], ['D', ['', 0]], ['E', ['', 0]]]
[[['B', ['A', 100], ['A', 100], ['A', 100]], ['C', ['A', 30], ['A', 30], ['A', 30]], ['D', ['', 0], ['C', 40], ['', 0]], ['E', ['', 0], ['', 0], ['C', 90]]]
[[['B', ['A', 100], ['A', 100], ['A', 100], ['D', 55], ['A', 100]], ['D', ['', 0], ['C', 40], ['', 0], ['C', 40], ['C', 40]], ['E', ['', 0], ['', 0], ['C', 90], ['', 0], ['D', 90]]]
[[['B', ['A', 100], ['A', 100], ['A', 100], ['D', 55], ['A', 100]], ['E', ['', 0], ['', 0], ['C', 90], ['', 0], ['D', 90]]]
Tiempo de ejecucion: 0.020183324813842773
El camino mas corto entre A y E es de 90
  
```

3.2. Ejemplo 2



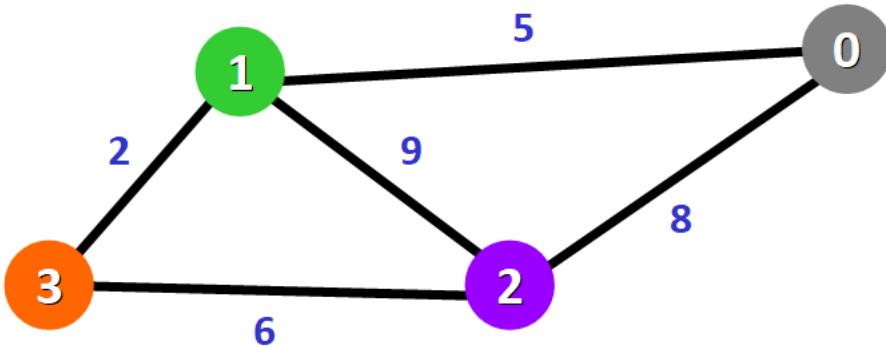
En este ejemplo se necesita llegar de A a E, en la imagen original los caminos no son dirigidos por lo cual en el programa se tomaron como caminos dirigidos. Empezamos en A revisando los caminos posibles, tenemos camino a B de 3 y camino a C de 1, tomamos C pero sin olvidar el camino en B, revisamos ahora a donde se puede ir desde C, hay camino a B con un total de 8 y a D con un total de 3, como a B existen dos maneras de llegar, podemos tomar la que nos cueste menor ponderación, en este caso, desde A a B con ponderación de 3, ahora revisamos los caminos a los que podemos ir desde B, podemos ir solamente a E con un camino total de 4 el cual es el recorrido mas corto en el grafo, tambien se puede tomar el camino de ir por C a D con camino de 3 pero sale exactamente igual resultado.

```

[[['A', [' ', 0]], ['B', ['A', 3]], ['C', ['A', 1]], ['D', [' ', 0]], ['E', [' ', 0]]]
[[['B', ['A', 3], ['C', 8], ['A', 3]], ['C', ['A', 1], ['A', 1], ['A', 1], ['D', [' ', 0], [' ', 0], ['C', 3]], ['E', [' ', 0], [' ', 0], [' ', 0]]]
[[['B', ['A', 3], ['A', 3], ['A', 3]], ['D', [' ', 0], [' ', 0], ['C', 3], [' ', 0]], ['E', [' ', 0], [' ', 0], [' ', 0], ['B', 4]]]
[[['D', [' ', 0], [' ', 0], ['C', 3], [' ', 0], ['C', 3], ['C', 3]], ['E', [' ', 0], [' ', 0], [' ', 0], ['B', 4], [' ', 0], ['D', 10]]]
[[['E', [' ', 0], [' ', 0], [' ', 0], ['B', 4], [' ', 0], ['D', 10]]]
Tiempo de ejecucion: 0.004992246627807617
El camino mas corto entre A y E es de 4

```

3.3. Ejemplo 3



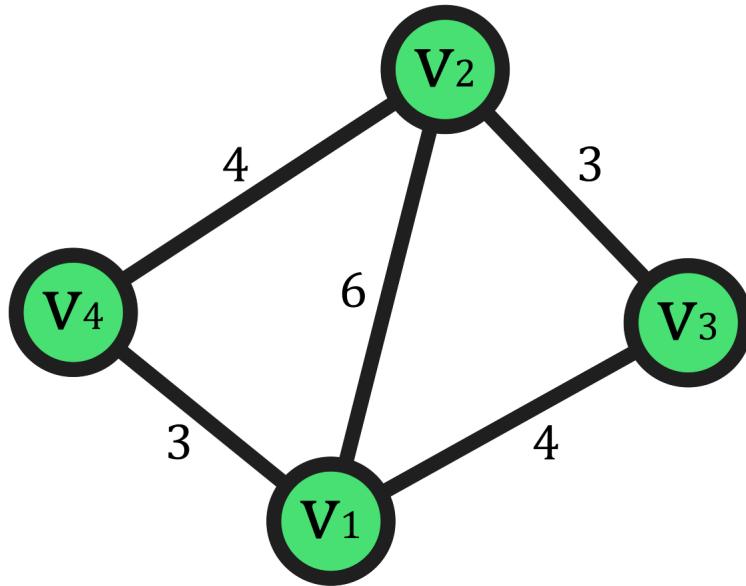
Aquí necesitamos el caminomas corto entre A y D para ello verificamos los caminos a los que podemos ir desde A, podemos ir a B con 5 de ponderación y podemos ir a C con 8 de ponderación, tomamos B por tener menor ponderación, ahora verificamos a donde podemos ir desde B, podemos ir a C con 14 total o ir a D con 7 total, tomamos el camino a D y ya tenemos el recorrido mas corto que sería 7

```

[[['A', ['', 0]], ['B', ['A', 5]], ['C', ['A', 8]], ['D', ['', 0]]]
[[['B', ['A', 5], ['A', 5], ['A', 5]], ['C', ['A', 8], ['B', 14], ['A', 8]], ['D', ['', 0], ['', 0], ['B', 7]]]
[[['C', ['A', 8], ['A', 8], ['A', 8]], ['D', ['', 0], ['', 0], ['B', 7], ['C', 14]]]
[[['D', ['', 0], ['', 0], ['B', 7], ['C', 14]]]]
Tiempo de ejecucion: 0.004992008209228516
El camino mas corto entre A y D es de 7

```

3.4. Ejemplo 4



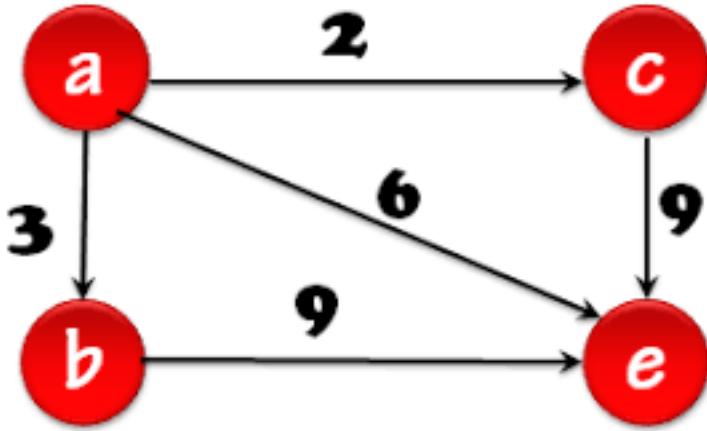
En este caso ocupamos el camino mas corto entre V1 y V2, podemos ir a V2 con 6 de recorrido, a V3 con 4 de recorrido y a V4 con 3 de recorrido, tomamos camino a V4 pero sin olvidar los anteriores, de V4 podemos ir a V2 con 7 de recorrido total, tomando el camino de V3, pasa exactamente lo mismo, entonces como el recorrido de V1 a V2 es menor, este es el recorrido mas corto el recorrido de V1 a V2 con 6 de ponderación.

```

[[['V1', ['', 0]], ['V2', ['A', 6]], ['V3', ['A', 4]], ['V4', ['A', 3]]]
[[['V2', ['A', 6], ['V4', 7]], ['V3', ['A', 4], ['A', 4]], ['V4', ['A', 3], ['A', 3]]]
[[['V2', ['A', 6], ['A', 6], ['V3', 7]], ['V3', ['A', 4], ['A', 4], ['A', 4]]]
[[['V2', ['A', 6], ['A', 6], ['A', 6]]]]
Tiempo de ejecucion: 0.0039958953857421875
El camino mas corto entre V1 y V2 es de 6

```

3.5. Ejemplo 5



Necesitamos el camino mas corto entre A y E. Los caminos que podemos tomar desde A son B con 3 de recorrido, C con 2 de recorrido y E con 6 de recorrido, tomamos B ya que tiene la menor ponderación pero sin olvidar los demás valores, desde B podemos ir a E con 12 de recorrido, si tomamos el camino de C podemos ver que lleva un camino de 11 a E por lo cual lo mejor es tomar el camino de A a E ya que tiene el menor recorrido de 6.

```
[[['A', ['', 0]], ['B', ['A', 3]], ['C', ['A', 2]], ['E', ['A', 6]]]
[[['B', ['A', 3], ['A', 3]], ['C', ['A', 2], ['A', 2]], ['E', ['A', 6], ['C', 11]]]
[[['B', ['A', 3], ['A', 3], ['A', 3]], ['E', ['A', 6], ['A', 6], ['B', 12]]]
[[['E', ['A', 6], ['A', 6], ['A', 6]]]]
Tiempo de ejecucion: 0.010982513427734375
El camino mas corto entre A y E es de 6
```

4. Conclusión

En conclusión, el algoritmo de Dijkstra demuestra ser una herramienta versátil para encontrar los caminos más cortos en un grafo. Su aplicación se extiende a diversos campos, desde enrutamiento de redes hasta logística de transporte. Al proporcionar un desglose detallado de la implementación del algoritmo y analizar sus complejidades, este documento busca mejorar la comprensión y la utilización del algoritmo de Dijkstra en la resolución de problemas del mundo real.