Algoritmo Dijkstra

Nombre: Alexandra Valeria Hernández Quintero **Matricula:** 1663507

Unidad de aprendizaje: Matemáticas Computacionales **Fecha:** 20/10/2017

Resumen: Este reporte trata sobre que es, como funciona y como programar Dijkstra en Python.

¿Qué es el Algoritmo Dijkstra?

El Algoritmo Dijkstra, es un algoritmo que determina la ruta más corta desde un nodo inicial hacia los demás nodos para ello es requerido como entrada un grafo cuyas aristas posean pesos.

¿Cómo funciona Dijkstra?

Imaginemos que se nos hace tarde para llegar al trabajo o escuela, claro está que nosotros buscaremos la ruta más corta para llegar al lugar deseado a tiempo. Dijkstra hace exactamente lo mismo que nosotros haríamos, buscar la ruta más corta, pero dentro de un grafo. En un principio tenemos el grafo con todos sus nodos como no visitados usando la analogía anterior sería como estar parados en nuestro hogar, el algoritmo parte de un nodo inicial el cual será ingresado, a partir de ese nodo evaluaremos sus a sus “hijos” , Dijkstra buscará el “camino” más óptimo, es decir que buscará el “camino” más rápido o con el que le cueste menos llegar a los hijos del nodo inicial. Después busca el que esté más cerca de nuestro nodo inicial, lo toma como punto intermedio y verifica si puede llegar más rápido a través de este nodo a los demás. Después escoge al siguiente más cercano (con las distancias ya actualizadas) y repite el proceso. Esto lo hace hasta que el nodo no visitado más cercano sea nuestro destino.

¿Cómo hacer el Algoritmo Dijkstra en Python?

Antes que nada, tenemos que “traer” o “incluir” unas librerías, ya que más adelante haremos uso de estas.

Ahora bien definimos una función, que tenga como dato de entrada un nodo, el cual será nuestro nodo inicial, después creamos un arreglo en el cual estarán las tuplas de lo que se estará almacenando, donde la distancia en un principio es cero ya que la distancia de un nodo consigo mismo es cero, también estará nuestro nodo inicial y colocamos paréntesis vacíos para guardar el "camino" hacia el nodo inicial. Después creamos un diccionario donde guardaremos las distancias y un conjunto para ir guardando los nodos ya visitados, es decir “marcar” los nodos visitados. Luego hacemos uso de un ciclo While, el cual repetirá las instrucciones mientras haya en el grafo nodos no visitados, dentro del ciclo While, mandamos llamar a la función heappop() la cual está incluida en las librerías de Python , la cual nos va ayudar a tomar la tupla que contiene la distancia menor, de esta forma el algoritmo nos asegura que solo tomara el “camino” más corto, después con una condición verificaremos si el nodo adyacente no lo hemos visitado anteriormente y si aún no lo hemos visitado lo agregaremos a el conjunto de visitado y también agregaremos la distancia de un nodo a otro en el diccionario de distancias ya antes mencionado, hacemos usa de una tupla,

para guardar el nodo adyacente y el camino, para luego usar un ciclo For que abarque cada hijo del nodo actual, y dentro del ciclo For volvemos a verificar si el nodo está dentro del conjunto de los visitados y si no está entonces se toma la distancia del nodo actual hacia el nodo hijo, para después hacer uso de la función heappush() la cual también está incluida en la Liberia de Python al igual que heappop(), heappush() nos ayudará a agregar al arreglo la distancia actual más la distancia hacia el nodo hijo, el nodo hijo "n" hacia donde se va, y el camino. Al final el algoritmo nos regresara el diccionario de las distancias.

Pseudocódigo

```
De heapq importar heappop, heappush

*definir aplanar(L):
    mientras tamaño(L) > 0:
        yield L[0]
        L = L[1]

**definir dijkstra(self, v):
        q = [(0, v, ())]
        distancia = diccionario()
        visitados = conjunto()
        mientras tamaño(q) > 0:
            (l, u, p) = heappop(q)
            Si u no esta en visitados:
                visitados.añadir(u)
                distancia[u] = (l,u,list(aplanar(p))[::-1] + [u])
            p = (u, p)
            Para cada n en self.vecinos[u]:
                Si n no esta en visitados:
                    el = self.E[(u,n)]
                    heappush(q, (l + el, n, p))
        regresar distancia
```

*****aplanar va fuera de la clase del grafo, mientras que **Dijkstra va dentro de la clase del grafo.

A continuación se mostraran los resultados al hacer pruebas con el algoritmo Dijkstra:

5 nodos y 10 aristas		
p	nodos	min (d)
a-a	a	0
a-c	c	1
a-e	e	1
a-b	b	2
a-d	d	2

10 nodos y 20 aristas		
p	nodos	min (d)
c-c	c	0
c-d	d	1
c-j	j	1
c-d-a	a	2
c-b	b	2
c-e	e	2
c-e-f	f	3
c-d-a-i	i	3
c-d-a-i-h	h	6
c-d-a-i-h-g	g	7

25 nodos y 50 aristas		
p	nodos	min (d)
s-s	s	0
s-e	e	1
s-t	t	1
s-v	v	1
s-x	x	1
s-e-d	d	2
s-e-r	r	2
s-x-w	w	2
s-e-y	y	2
s-e-f	f	3
s-e-d-n	n	3
s-e-r-q	q	3
s-e-f-a	a	4
s-e-f-b	b	4
s-e-f-l	l	4
s-e-d-n-m	m	4
s-e-f-o	o	4
s-e-f-b-h	h	5
s-e-f-b-i	i	5
s-e-f-b-p	p	5
s-e-f-a-c	c	6
s-e-f-g	g	6
s-x-w-k	k	7
s-x-w-k-j	j	8
s-v-u	u	9

15 nodos y 30 aristas		
p	nodos	min (d)
k-k	k	0
k-j	j	1
k-l	l	5
k-j-o	o	5
k-l-f	f	6
k-l-m	m	6
k-j-o-n	n	6
k-l-f-a	a	7
k-l-f-b	b	7
k-j-o-n-d	d	7
k-u-o-n-d-e	e	8
k-l-f-b-h	h	8
k-l-f-b-i	i	8
k-l-f-a-c	c	9
k-l-f-g	g	9

20 nodos y 40 aristas		
p	nodos	min (d)
s-s	s	0
s-e	e	1
s-t	t	1
s-e-d	d	2
s-e-r	r	2
s-e-f	f	3
s-e-d-n	n	3
s-e-r-q	q	3
s-e-f-a	a	4
s-e-f-b	b	4
s-e-f-l	l	4
s-e-d-e-m	m	4
s-e-f-o	o	4
s-e-f-b-h	h	5
s-e-f-b-i	i	5
s-e-f-b-p	p	5
s-e-f-a-c	c	6
s-e-f-g	g	6
s-e-f-o-j	j	8
s-e-f-o-j-k	k	9

Conclusiones:

Claro está que entre menos aristas y nodos tenga un grafo, para el algoritmo Dijkstra será más sencillo encontrar el “camino” más corto, también si todos los nodos están conectados entre sí, ya que si uno no está conectado directamente, pues Dijkstra tendrá que hacer un “recorrido” más largo. Una gran desventaja de este algoritmo es que solo trabaja con pesos positivos, si al menos uno de los nodos de un grafo tuviera pesos negativos, entonces este algoritmo no nos serviría de nada.