Reporte de algoritmo de Dijkstra

José Manuel Tapia Avitia.
Matrícula: 1729372
jose.tapiaav@gmail.com
https://github.com/jose-tapia/1729372MC

20 de octubre de 2017

El presente reporte tiene la finalidad de analizar y mostrar el algoritmo de Dijkstra, su comportamiento en grafos y el como podríamos implementarlo en Python.

1. Dijkstra

1.1. Descripción del algoritmo

Comunmente queremos llegar a nuestro destino lo más pronto posible, ya sea para por un compromiso o por el simple hecho de querer optimizar ciertos aspectos cotidianos, solo que tenemos el problema de tener distintos caminos y/o rutas para tomar, y de manera clara, no podemos recorrer todas para saber cuál es la mejor. Si planteamos nuestros caminos y rutas en un Grafo como aristas con peso, el algoritmo de Dijkstra nos ayuda a conseguir dicho objetivo con una complejidad realmente buena.

Definiremos como ruta.ªquel camino consistente de aristas del grafo, con punto de partida el nodo inicial y final aquel en donde nos encontremos.

El algoritmo que implementaremos necesita un Grafo y el nodo inicial, en donde se realizará lo siguiente:

- Comenzamos nuestra ruta en el nodo inicial, con una distancia recorrida de 0.
- Dentro de las rutas alcanzadas, nos tomaremos la que tenga la menor distancia.
- De esta ruta, la expandiremos a sus nodos adyacentes, si estos no han sido ya visitados por una ruta , agregando estas nuevas rutas a nuestro conjunto de rutas con la nueva distancia recorrida.
- El proceso se repite hasta ya no tener más rutas.

1.2. Estructura de datos auxiliar, *Heap*

Todo suena muy fácil por el momento... pero surge un problema, ¿cómo podemos obtener la ruta con menor distancia?

La primer idea que podemos tener podría se la de tener un arreglo con las rutas. Agregar una ruta sería en constante con un *append*. Cada vez que necesitemos la ruta con distancia óptima, buscarla. En caso de querer eliminarla, tendríamos que buscarla y luego eliminarla. Lo malo de esta idea es que ambas operaciones, buscar y eliminar, tienen complejidad $\mathcal{O}(n)$, en donde n es la longitud del arreglo.

Otra idea no tan mala podría ser el ordenar el arreglo de manera que la ruta con menor distancia este a nuestro alcance siempre. Así lograríamos que las operaciones de buscar y eliminar tengan complejidad $\mathcal{O}(1)$, pero la desventaja es que tendríamos que ordenar el arreglo cada que tengamos que agregar una ruta, de manera más optima, podria ser complejidad $\mathcal{O}(n)$ agregando una sola ruta, o un conjuntto de rutas en $\mathcal{O}(nlogn)$, lo cuál sigue siendo una mala complejidad.

Podemos auxiliarnos de una estructura de datos que nos permite realizar cada operación en complejidad $\mathcal{O}(logn)$, lo cuál nos garantiza una eficiencia en el algoritmo muy buena. Dicha estructura es la del $Heap$. Yo la implemente con los siguientes métodos:

- *__init__*: Necesaria para inicializar el $Heap$.
- $empty()$: Retorna verdadero si el $Heap$ no tiene elementos o falso en caso de tenerlos.
- $top()$: Retorna el menor elemento que se encuentre en el $Heap$.
- $push(x)$: Agrega al $Heap$ el elemnto x.
- $pop()$: Retorna y elimina del $Heap$ el menor elemento que se encuentre en el momento.

La implementación en Python que realice es la siguiente:

```python
class Heap(object):
    def __init__(self):
        self.hp=[0]

    def empty(self):
        return len(self.hp)==1

    def top(self):
        if len(self.hp)>1:
            return self.hp[1]
        else:
            return None

    def push(self,x):
        tam=len(self.hp)
        w=tam
        self.hp.append(x)
        while w>0:
            ww=int(w/2)
            if ww>0:
                if self.hp[ww]>self.hp[w]:
                    self.hp[ww],self.hp[w],w=self.hp[w],self.hp[ww],ww
                else:
                    break
            else:
                break

    def pop(self):
        tam=len(self.hp)-1
        self.hp[1],self.hp[tam]=self.hp[tam],self.hp[1]
        tam-=1
```

```python
        w=1
        while w<=tam:
            i,d=self.hp[w],self.hp[w]
            if 2*w<=tam:
                i=self.hp[2*w]
            if 2*w+1<=tam:
                d=self.hp[2*w+1]
            if self.hp[w]>i or self.hp[w]>d:
                if i<d:
                    self.hp[w],self.hp[2*w]=self.hp[2*w],self.hp[w]
                    w=2*w
                else:
                    self.hp[w],self.hp[2*w+1]=self.hp[2*w+1],self.hp[w]
                    w=2*w+1
            else:
                break
        return self.hp.pop()
```

El *Heap* nos sirve particularmente para el problema que tuvimos, puesto que podemos visualizarlo como un conjunto que nos permite agregar elementos, preguntar por el menor elemento y eliminarlo de manera eficaz. Cabe a resaltar que el *Heap* lo implemente en una clase, para poder crear diversos $Heap's$.

1.3. Implementación de Dijkstra

La implementación de Dijkstra se realizará como un método de la clase Grafo ya que esto nos permite un manejo más como del mismo.

Mostraremos primero la clase de Grafo que tenemos actualmente y después de ello el método de Dijkstra ya implementado.

```python
class Grafo(object):
    def __init__(self):
        self.vertices=set()
        self.aristas=dict()
        self.vecinos=dict()

    def agrega(self,v):
        self.vertices.add(v)
        if not v in self.vecinos:
            self.vecinos[v]=set()

    def conecta(self,u,v,peso=1):
        self.agrega(u)
        self.agrega(v)
        self.aristas[(u,v)]=self.aristas[(v,u)]=peso
        self.vecinos[u].add(v)
        self.vecinos[v].add(u)

    @property
    def complemento(self):
        comp=Grafo()
```

```python
        for a in self.vertices:
            for b in self.vertices:
                if a!=b and (a,b) not in self.aristas:
                    comp.conecta(a,b,1)
```

El método de Dijkstra es:

```python
  def dijkstra(self,ini):
        bsq=Heap()
        bsq.push((0,ini,()))
        visitados=set()
        respuesta=dict()
        while not bsq.empty():
            (dist,nodo,path)=bsq.pop()
            if nodo in visitados:
                continue
            visitados.add(nodo)
            respuesta[nodo]=(dist,descomponer((nodo,path)))
            for w in self.vecinos[nodo]:
                if w in visitados:
                    continue
                d=self.aristas[(nodo,w)]
                bsq.push((d+dist,w,(nodo,path)))
        return respuesta
```

En la implementación podemos ver la implementación casi directa del algoritmo descrito inicialmente. Para almacenar las rutas optimas, hacemos uso de un diccionario en donde a cada nodo le es asignado el camino más optimo junto a su distancia o ningún camino. Para poder almacenar el camino de manera sencilla se hizo uso del poder de los parentesis, mejor conocido como tuplas. En donde $(x_n, (x_{n_1}, (x_{n-2}, (\ldots (x_2,\ x_1) \ldots))))$ representa el camino $[x_1,\ x_2,\ x_3, \ldots,\ x_n]$, es decir, inicia en x_1, camina a x_2, que continua por x_3, $\ldots$ así hasta llegar a x_n. Podemos ver en donde se realiza dicha comprensión en la tupla que se inserta en el *Heap* *bsq* al momento de expandirse en sus vecinos w el *nodo*. usamos la función *descomponer* que recibe una tupla de la forma $(x_n, (x_{n_1}, (x_{n-2}, (\ldots (x_2,\ x_1) \ldots))))$ y regresa un arreglo con el camino $[x_1,\ x_2,\ x_3, \ldots,\ x_n]$. Dicha función es la que siguiente:

```python
def descomponer(w):
    if w==():
        return []
    (x,y)=w
    return descomponer(y)+[x]
```

El algoritmo de Dijkstra agrega a lo más dos veces las m aristas del Grafo (tratando de expandirse de cada lado de la arista) y visita a lo más una vez cada uno de los n nodo (vértice). Por lo tanto la complejidad del algoritmo sería $\mathcal{O}((n+m)log(n+m))$ aproximadamente, ya que cada operación de agregar y quitar del *Heap* se tarda $\mathcal{O}(logn)$.

Cabe a resaltar que el algoritmo de Dijkstra funciona solo en grafos donde las aristas tienen pesos no negativos. La demostración de su funcionamiento se basa en ello, además que con aristas negativas se pueden hallar contra-ejemplos en donde no da un correcto funcionamiento. Para grafos con aristas negativas existen algoritmos como $Bellman - Ford$.

1.4. Prueba del algoritmo

En esta sección probaremos la implementación del algoritmo de Dijkstra con varios grafos. Ejemplificaremos los grafos en cómo se pueden insertar en nuestro programa de Python, y uno que otro una representación gráfica de los mismos.

Para crear grafos extensos, definiremos una función que reciba n y m, que significan la cantidad de nodos que tendrá el Grafo y cuantos vecinos tendrá al menos cada nodo respectivamente.

He aqui la implementación de dicha función:

```python
def crearGrafo(n,m):
    G=Grafo()
    for x in range(1,n+1):
        adyacentes=[]
        for k in range(m):
            adyacentes.append((random.randint(x+1,n),random.randint(1,100))
        for (u,d) in adyacentes:
            G.conecta(x,u,d)
    return G
```

Igual, para poder representar dichos grafos, implemente un método que nos permita visualizar dicho grafo por medio de los vértices que lo componen y sus aristas.

```python
def __str__(self):
    nodos="Vertices:\n"
    for w in self.vertices:
        nodos+=" "+str(w)+" "
    vis=set()
    edges="Aristas:\n"
    for w in self.aristas:
        (x,y)=w
        if w in vis:#Ya que puede que nos encontremos con (x,y) y (y,x)
            continue
        if (y,x) in vis:
            continue
        vis.add(w)
        edges+=" "+str(w)+":\t"+str(self.aristas[w])+"\n"
    return nodos+"\n"+edges
```

Entonces, si llamamos a la función *print* con argumento un Grafo , nos imprimirá los vértices y aristas del mismo.

Igual, para poder representar la respuesta que nos retorna la función Dijkstra , implementamos la siguiente función, en donde la primera columna es el vértice destino, la segunda, la menor distancia posible para llegar a ella y el camino que cumple dicha distancia.

```python
def imprimirDijkstra(resultado):
    rans=Heap()
    for w in resultado:
        (dist,path)=resultado[w]
        rans.push((dist,w,path))
    while not rans.empty():
        (dist,w,path)=rans.pop()
        linea=" "+str(w)+": "+str(dist)+"\t"+str(path)
        print(linea)
```

1.4.1. Primer Grafo

Representación del Grafo :

```
Vertices:
 1  2  3  4  5
Aristas:
 (1, 4): 5
 (1, 2): 2
 (2, 3): 14
 (2, 4): 5
 (2, 5): 4
 (3, 5): 34
 (4, 5): 48
```

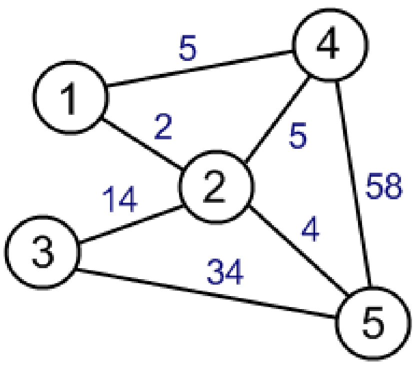
Resultado del Dijkstra :

```
 1: 0   [1]
 2: 2   [1, 2]
 4: 5   [1, 4]
 5: 6   [1, 2, 5]
 3: 16  [1, 2, 3]
```

Para que el programa lo interprete:

```
G1=Grafo()
G1.conecta(1,4,5)
G1.conecta(1,2,2)
G1.conecta(2,3,14)
G1.conecta(2,4,5)
G1.conecta(2,5,4)
G1.conecta(3,5,34)
G1.conecta(4,5,48)
print(G1.dijkstra(1))
```

Representación gráfica:



1.4.2. Segundo Grafo

Representación del Grafo :

```
Vertices:
 C   E   F   H   B   A   D   I   G
Aristas:
 ('A', 'B'): 22
 ('A', 'C'): 9
 ('A', 'D'): 12
 ('B', 'C'): 35
 ('B', 'F'): 36
 ('B', 'H'): 34
 ('C', 'D'): 4
 ('C', 'E'): 65
 ('C', 'F'): 42
 ('D', 'I'): 30
 ('E', 'F'): 18
 ('E', 'G'): 23
 ('F', 'G'): 39
 ('F', 'H'): 24
 ('G', 'H'): 25
 ('G', 'I'): 21
 ('H', 'I'): 19
```

Resultado del Dijkstra :

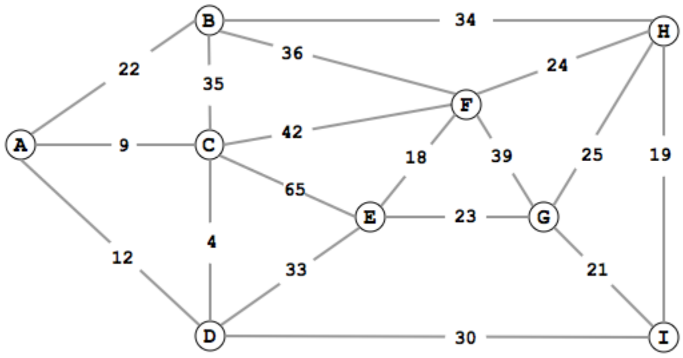
```
 A: 0    ['A']
 C: 9    ['A', 'C']
 D: 12   ['A', 'D']
 B: 22   ['A', 'B']
 I: 42   ['A', 'D', 'I']
 F: 51   ['A', 'C', 'F']
 H: 56   ['A', 'B', 'H']
 G: 63   ['A', 'D', 'I', 'G']
 E: 69   ['A', 'C', 'F', 'E']
```

Para que el programa lo interprete:

```
G2=Grafo()
G2.conecta('A','B',22)
G2.conecta('A','C',9)
G2.conecta('A','D',12)
G2.conecta('B','C',35)
G2.conecta('B','F',36)
G2.conecta('B','H',34)
G2.conecta('C','D',4)
G2.conecta('C','E',65)
G2.conecta('C','F',42)
G2.conecta('D','I',30)
G2.conecta('E','F',18)
G2.conecta('E','G',23)
G2.conecta('F','G',39)
```

```
G2.conecta('F','H',24)
G2.conecta('G','H',25)
G2.conecta('G','I',21)
G2.conecta('H','I',19)
print(G2.dijkstra('A'))
```

Representación gráfica:



1.4.3. Tercer Grafo

Representación del Grafo .

```
Vertices:
 h   g   c   k   a   d   f   b   i   e   j
Aristas:
 ('a', 'b'): 4
 ('a', 'd'): 1
 ('a', 'e'): 2
 ('b', 'c'): 10
 ('b', 'f'): 5
 ('c', 'g'): 8
 ('c', 'k'): 7
 ('d', 'e'): 2
 ('d', 'h'): 14
 ('e', 'f'): 11
 ('e', 'h'): 18
 ('f', 'g'): 6
 ('f', 'i'): 13
 ('g', 'j'): 17
 ('g', 'k'): 9
 ('h', 'i'): 15
 ('i', 'j'): 16
 ('j', 'k'): 12
```

Resultado del Dijkstra :

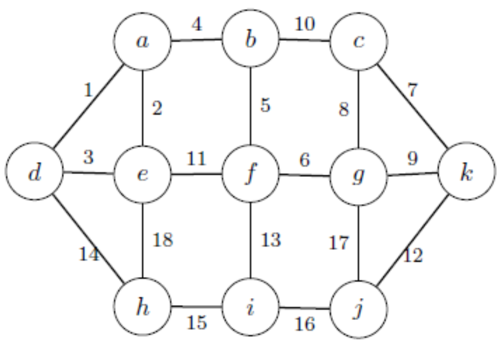
```
 a: 0   ['a']
 d: 1   ['a', 'd']
 e: 2   ['a', 'e']
 b: 4   ['a', 'b']
 f: 9   ['a', 'b', 'f']
 c: 14  ['a', 'b', 'c']
```

```
g: 15   ['a', 'b', 'f', 'g']
h: 15   ['a', 'd', 'h']
k: 21   ['a', 'b', 'c', 'k']
i: 22   ['a', 'b', 'f', 'i']
j: 32   ['a', 'b', 'f', 'g', 'j']
```

Para que el programa lo interprete:

```python
G3=Grafo()
G3.conecta('a','b',4)
G3.conecta('a','d',1)
G3.conecta('a','e',2)
G3.conecta('b','c',10)
G3.conecta('b','f',5)
G3.conecta('c','g',8)
G3.conecta('c','k',7)
G3.conecta('d','e',2)
G3.conecta('d','h',14)
G3.conecta('e','f',11)
G3.conecta('e','h',18)
G3.conecta('f','g',6)
G3.conecta('f','i',13)
G3.conecta('g','j',17)
G3.conecta('g','k',9)
G3.conecta('h','i',15)
G3.conecta('i','j',16)
G3.conecta('j','k',12)
print(G3.dijkstra('a'))
```

Representación gráfica:



1.4.4. Cuarto Grafo

Representación del Grafo :

```
Vertices:
 1   2   3   4   5   6   7   8   9   10   11   12   13   14   15
Aristas:
 (1, 3):        75
 (1, 15):       20
 (1, 14):       65
 (2, 9):        58
 (2, 5):        32
 (2, 11):       50
 (3, 15):       17
 (3, 14):       89
 (3, 3):        97
 (4, 4):        93
 (4, 7):        30
 (4, 6):        34
 (5, 9):        4
 (5, 15):       50
 (6, 14):       33
 (6, 10):       46
 (7, 8):        76
 (7, 9):        25
 (7, 15):       22
 (8, 15):       55
 (8, 11):       70
 (8, 10):       2
 (9, 14):       5
 (9, 11):       68
 (10, 13):      78
 (10, 11):      6
 (10, 10):      91
 (11, 14):      50
 (11, 12):      31
 (11, 13):      20
 (12, 15):      12
 (12, 12):      76
 (13, 14):      87
 (13, 15):      56
 (14, 15):      40
 (14, 14):      23
 (15, 15):      7
```

Resultado del Dijkstra :

```
 1:      0       [1]
 15:     20      [1, 15]
 12:     32      [1, 15, 12]
 3:      37      [1, 15, 3]
 7:      42      [1, 15, 7]
```

```
14:     60       [1, 15, 14]
11:     63       [1, 15, 12, 11]
9:      65       [1, 15, 14, 9]
5:      69       [1, 15, 14, 9, 5]
10:     69       [1, 15, 12, 11, 10]
8:      71       [1, 15, 12, 11, 10, 8]
4:      72       [1, 15, 7, 4]
13:     76       [1, 15, 13]
6:      93       [1, 15, 14, 6]
2:      101      [1, 15, 14, 9, 5, 2]
```

Para interpretarlo en el programa:

```
G4=crearGrafo(15,3)
print(G4)
imprimirDijkstra(G4.dijkstra(1))
```

1.4.5. Quinto Grafo

Representación del grafo:

```
Vertices:
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19
20
Aristas:
 (1, 9):        37
 (1, 20):       93
 (1, 18):       53
 (2, 9):        85
 (2, 12):       55
 (2, 13):       77
 (3, 9):        30
 (3, 11):       40
 (4, 16):       56
 (4, 5):        64
 (4, 19):       30
 (5, 11):       23
 (5, 5):        86
 (5, 19):       95
 (6, 11):       33
 (6, 19):       27
 (7, 7):        13
 (7, 8):        42
 (8, 13):       76
 (8, 15):       46
 (8, 17):       62
 (9, 11):       63
 (9, 18):       86
 (9, 16):       89
 (10, 10):      76
 (10, 12):      43
```

```
(10, 19):      5
(11, 14):      66
(11, 15):      48
(11, 13):      92
(12, 12):      99
(12, 19):      60
(13, 15):      79
(13, 16):      73
(13, 19):      25
(14, 15):      35
(14, 16):      80
(14, 17):      96
(15, 15):      34
(15, 19):      86
(15, 20):      89
(16, 19):      76
(16, 18):      52
(17, 18):      44
(17, 17):      53
(18, 19):      41
(19, 19):      55
(20, 20):      98
```

Resultado del Dijkstra :

```
1:      0       [1]
9:      37      [1, 9]
18:     53      [1, 18]
3:      67      [1, 9, 3]
20:     93      [1, 20]
19:     94      [1, 18, 19]
17:     97      [1, 18, 17]
10:     99      [1, 18, 19, 10]
11:     100     [1, 9, 11]
16:     105     [1, 18, 16]
13:     119     [1, 18, 19, 13]
6:      121     [1, 18, 19, 6]
2:      122     [1, 9, 2]
5:      123     [1, 9, 11, 5]
4:      124     [1, 18, 19, 4]
12:     142     [1, 18, 19, 10, 12]
15:     148     [1, 9, 11, 15]
8:      159     [1, 18, 17, 8]
14:     166     [1, 9, 11, 14]
7:      201     [1, 18, 17, 8, 7]
```

Interpretación en el programa:

```
G5=crearGrafo(20,3)
print(G5)
imprimirDijkstra(G5.dijkstra(1))
```

1.4.6. Sexto Grafo

Representación del Grafo :

```
Vertices:
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19
20  21  22  23  24  25
Aristas:
 (1, 7):        91
 (1, 25):       54
 (1, 15):       67
 (2, 22):       44
 (2, 23):       3
 (2, 8):        3
 (3, 18):       22
 (3, 21):       39
 (3, 25):       15
 (4, 5):        39
 (4, 13):       80
 (5, 11):       73
 (5, 16):       11
 (5, 14):       94
 (6, 19):       1
 (6, 21):       89
 (6, 20):       94
 (7, 17):       12
 (7, 9):        14
 (7, 22):       42
 (8, 14):       5
 (8, 24):       77
 (8, 11):       63
 (9, 18):       57
 (9, 17):       80
 (9, 11):       67
 (10, 21):      77
 (10, 10):      34
 (10, 12):      75
 (11, 15):      40
 (11, 24):      95
 (11, 13):      5
 (12, 13):      32
 (12, 14):      37
 (12, 16):      73
 (13, 25):      2
 (13, 21):      62
 (13, 24):      51
 (14, 20):      52
 (14, 14):      66
 (15, 25):      32
 (15, 19):      89
 (15, 17):      44
```

```
(16, 17):      90
(16, 24):      72
(17, 22):      40
(17, 20):      13
(17, 23):      64
(18, 21):      57
(18, 19):      68
(18, 18):      24
(19, 20):      26
(19, 21):      1
(19, 24):      90
(20, 25):      13
(20, 21):      78
(20, 22):      76
(21, 22):      4
(22, 22):      85
(22, 24):      6
(22, 23):      73
(23, 25):      47
(24, 25):      80
(25, 25):      92
```

Resultado del Dijkstra :

```
1:      0       [1]
25:     54      [1, 25]
13:     56      [1, 25, 13]
11:     61      [1, 25, 13, 11]
15:     67      [1, 15]
20:     67      [1, 25, 20]
3:      69      [1, 25, 3]
17:     80      [1, 25, 20, 17]
12:     88      [1, 25, 13, 12]
7:      91      [1, 7]
18:     91      [1, 25, 3, 18]
19:     93      [1, 25, 20, 19]
6:      94      [1, 25, 20, 19, 6]
21:     94      [1, 25, 20, 19, 21]
22:     98      [1, 25, 20, 19, 21, 22]
23:     101     [1, 25, 23]
2:      104     [1, 25, 23, 2]
24:     104     [1, 25, 20, 19, 21, 22, 24]
9:      105     [1, 7, 9]
8:      107     [1, 25, 23, 2, 8]
14:     112     [1, 25, 23, 2, 8, 14]
5:      134     [1, 25, 13, 11, 5]
4:      136     [1, 25, 13, 4]
16:     145     [1, 25, 13, 11, 5, 16]
10:     163     [1, 25, 13, 12, 10]
```

Interpretación en el programa:

```
G6=crearGrafo(25,3)
print(G6)
imprimirDijkstra(G6.dijkstra(1))
```

2. Detalles de implementación

El algoritmo de Dijkstra es fácil de entender, e inclusive la implementación no es del otro mundo. Lo único que resulto más complicado fue el tener bien estructurado que se realizará y en que orden. Además de tener las estructuras adecuadas, tal como el *Heap*. Me resulto divertido el implementar un *Heap* en Python, teniendo en cuenta que llevaba tiempo sin implementar uno, puesto que en C++ se encontraba un estilo de *Heap* ya implementado, la *priority_queue*. E igual me agrado la idea de comprimir el camino en tuplas y el como descomponerla, ingeniosa la forma de aprovechar las herramientas que Python nos aporta. También me agrado el tener que aprender más sobre la sintaxis de Python, cómo el método *__str__* de una clase para que pueda imprimir. E igual tuve que ingenarmelas para facilitarme el trabajo de pasar el Grafo/resulttado del Dijkstra a texto relativamente bien representado.