Reporte de estructuras de datos

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

5 de octubre de 2017

El presente reporte tiene la finalidad de mostrar a grandes rasgos las caracteristicas principales de las estructuras de datos Pilas, Filas y Grafos, utilizandolas para implementar los algoritmos de DFS, BFS, encontrar el Centro y diametro de un grafo.

1. Pila

La estructura de datos Pila es una forma de organizar objetos, en donde nos permite incluir objetos, obtener el ultimo objeto y conocer la cantidad de objetos en la estructura. Podemos imaginarnos la Pila como el funcionamiento de una pila de platos, podemos agregar platos arriba, quitar el plato de la parte superior y saber cuantos platos hay, más no podemos quitar ó agregar un plato intermedio.

Definiremos la clase Pila con los siguientes métodos:

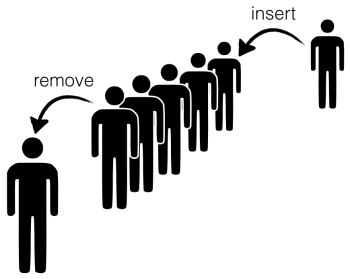
- *__init__*: Inicializa la pila como un arreglo vacio.
- *obtener*: Regresa el último objeto, es decir, el último de nuestro arreglo actual, si es que existe, y además lo elimina.
- *meter*: Incluye un objeto e a la pila, el cual sería agregado al final del arreglo.
- Como propiedad, *longitud* que regresa la cantidad de objetos en la pila, es decir, la longitud del arreglo.

```
class Pila(object):
    def __init__(self):
        self.a=[]
    def obtener(self):
        return self.a.pop()
    def meter(self,e):
        self.a.append(e)
        return len(self.a)
    @property
        def longitud(self):
        return len(self.a)
```

La complejidad de cada operación es $\mathcal{O}(1)$. A la estructura Pila también se le conoce como estructura $FILO$ (First In, Last out).

2. Fila

La siguiente estructura de datos es la Fila, también conocida como Cola. Nos permite incluir objetos, obtener el primer objeto y conocer la cantiad de objetos en la estructura. Como su nombre lo indica, esta estructura se asemeja a una fila como las que conocemos, al llegar una persona se forma al final, la persona que lleva más tiempo esperando pasa primero y podemos saber cuántas personas están en la fila.



Definiremos la clase Fila con los siguientes métodos:

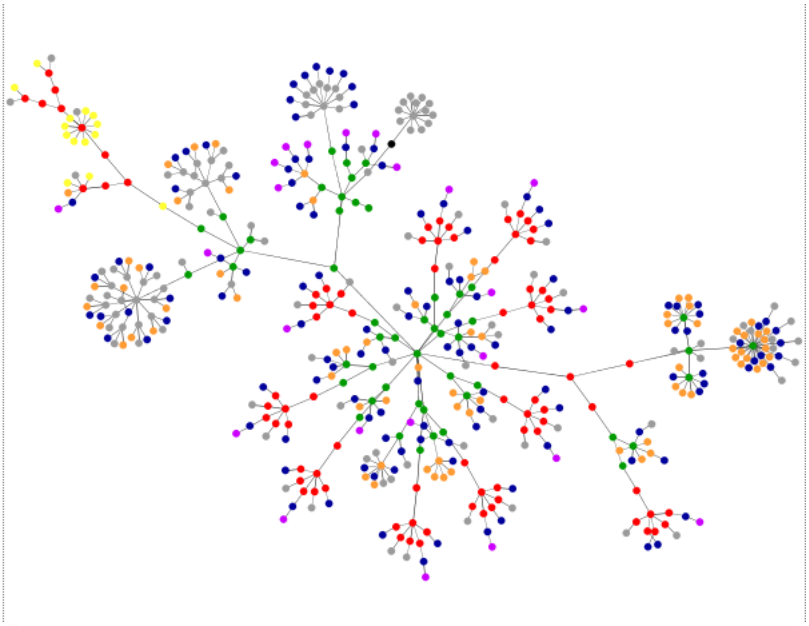
- *__init__*: Inicializa la fila como un arreglo vacio.
- *obtener*: Regresa el primer objeto, es decir, el primer objeto de nuestro arreglo actual, si es que existe, y además lo elimina.
- *meter*: Incluye un objeto e a la fila, el cual sería agregado al final del arreglo.
- Como propiedad, *longitud* que regresa la cantidad de objetos en la fila, es decir, la longitud del arreglo.

```
class Fila(object):
    def __init__(self):
        self.a=[]
    def obtener(self):
        return self.a.pop(0)
    def meter(self,e):
        self.a.append(e)
        return len(self.a)
    @property
    def longitud(self):
        return len(self.a)
```

La complejidad de cada operación es $\mathcal{O}(1)$. A la estructura Fila también se le conoce como estructura $FIFO$ (First In, First out).

3. Grafo

Definiremos Grafo como el conjunto de vértices (también conocidos como nodos) que están conectados por medio de aristas (también conocidass como arcos). De manera general, nos sirve como medio para representar situaciones, por ejemplo, relaciones de amistad, caminos entre ciudades o islas, fluidos de liquido o energía, entre otras aplicaciones.



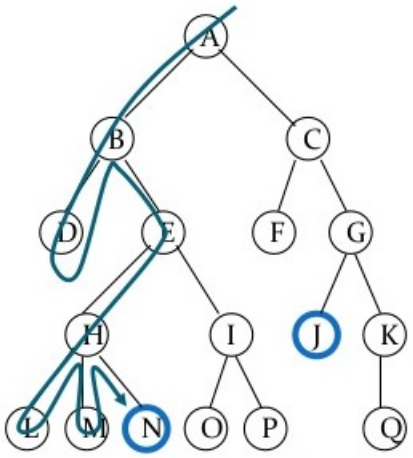
Definiremos la clase Grafo con los siguientes métodos:

- *__init__*: Inicializa el set de vértices y los diccionarios de las aristas y los vecinos de los vértices (Se dice que u es vecino de v si existe una arista que los une).
- *agrega*: Incluye el nodo v al grafo, si no estaba, inicializa su lista de vecinos como un set vacio.
- *conecta*: Dado dos nodos u, v y de manera opcional un *peso*, se interpreta que hay una arista entre u y v con valor *peso*, realizando las respectivas actualizaciones a los sets de u y v.
- Como propiedad, *complemento*, que regresa un grafo que cumple que tiene los mismos vértices del grafo original, solo si una pareja u, v de nodos no tiene una arista que los una, los une, y si sí la tiene, no los une.

```
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()
        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)
```

4. DFS

El algoritmo DFS hace referencia a Depth First Search, que en español se interpretaría como Búsqueda en profundidad, dando una idea más clara de lo que hace. Dado un vértice inicial, el algoritmo realiza un recorrido del grafo cómo si fueramos caminando por los vértices, es decir, de un vértice u, se desplaza a otro vértice v no visitado adyacente a u. Al terminar de "visitar"v, trata de visitar sus otros vértices adyacentes.



En la imagen anterior, la DFS realizaría el recorrido de la siguiente forma: A B D E H L M N I O P C F G J K Q. La siguiente implementación del algoritmo, dado un grafo y un vértice inicial, regresa el arreglo de vértices en el orden en que fueron visitados.

Para ello, se inicializa el arreglo viS de aquellos vértices que han sido visitados, una Pila, en donde se introduce el primer vértice.

Mientras la pila tenga vértices que checar, checará si el vértice ya ha sido recorrido, en caso de que sí, se omitirá, sino, se visitará dicho vértice y se agrega a la lista vis.

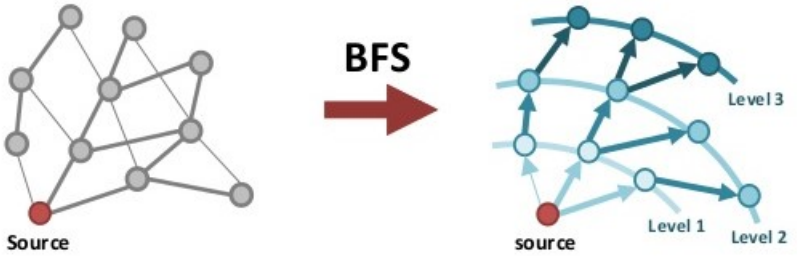
Para realizar el recorrido de un vértice u, se introducen a la pila los vecinos v de u que no han sido visitados, con la posibilidad de que se encuentren en la pila (Esto no es un problema, ya que al momento de llegar a un vértice ya visitado en la pila, se omitirá).

En caso de que la pila se quede sin elementos, signfica que el algoritmo ha concluido y el recorrido realizado se encontrará en vis, arreglo que se regresará.

```
def DFS(graph,ini):
    vis=[]
    bsq=Pila()
    bsq.meter(ini)
    while bsq.longitud>0:
        act=bsq.obtener()
        if act in vis:
            continue
        vis.append(act)
        vecinos=graph.vecinos[act]
        for w in vecinos:
            if w not in vis:
                bsq.meter(w)
    return vis
```

5. BFS

El algoritmo BFS hace referencia a Breadth First Search, que en español se interpretaría como Búsqueda en amplitud. Dado un vértice inicial, el algoritmo realiza un recorrido del grafo visitando primero los vértices más cercanos al vértice inicial, como si fuera una .ºnda de expansión".



En la imagen anterior, la BFS se logra visualizar el como visita los vértices, dando un . La siguiente implementación del algoritmo, dado un grafo y un vértice inicial, regresa el arreglo de vértices en el orden en que fueron visitados.

Para ello, se inicializa el arreglo viS de aquellos vértices que han sido visitados, una Fila, en donde se introduce el primer vértice.

A diferencia de la DFS, en la fila se tiene en orden de cercanía los vértices v al vértice inicial.

Mientras la fila tenga vértices que checar, se realizará el recorrido para estos.

Para realizar el recorrido de un vértice u, se introducen a la fila los vecinos v de u que no han sido visitados ni se encuentren en la fila (Si se encuentra en la fila, tiene una distancia al vértice igual o menor a la que se lograría al moverse de u a v, por lo cual es mejor no agregarla a la fila).

En caso de que la fila se quede sin elementos, signfica que el algoritmo ha concluido y el recorrido realizado se encontrará en vis, arreglo que se regresará.

```
def BFS(graph,ini):
    vis=[ini]
    bsq=Fila()
    bsq.meter(ini)
    while bsq.longitud>0:
        act=bsq.obtener()
        vecinos=graph.vecinos[act]
        for w in vecinos:
            if w not in vis:
                vis.append(w)
                bsq.meter(w)
    return vis
```

6. Centro de un grafo

Definiremos para un grafo lo siguiente:
Para dos vértices u, v en el grafo:

- Un camino de u a v es un recorrido de vértices que inicia en u y termina en v.
- La longitud de un camino de u a v es la cantidad de aristas del recorrido.
- La distancia entre u y v es la menor longitud entre todos los caminos de u a v.

El centro de un grafo es el vértice que cumple que la distancia más grande a cualquier vértice del grafo es la mínima posible.

El objetivo del algoritmo es dado un grafo, obtener un centro para el mismo.
Para ello, primero definiremos algunas funciones auxiliares, como *zip*, que recibe dos arreglos, junta los elementos respectivos y los regresa en un solo arreglo.

```
def zip(a,b):
    return [(a[i],b[i]) for i in range(min(len(a),len(b)))]
```

La función *superBFS* recibe un grafo y un vértice inicial, en donde es básicamente el mismo algoritmo que el BFS, solo que tiene añadido a cada vértice la distancia al vértice inicial.

```
def superBFS(graph,ini):
    vis =[ini]
    dist=[0]
    bsq=Cola()
    bsq.meter((ini,0))
    while bsq.longitud>0:
        (act,d)=bsq.obtener()
        vecinos=graph.vecinos[act]
        for w in vecinos:
            if w not in vis:
                vis.append(w)
                dist.append(d+1)
                bsq.meter((w,d+1))
    return zip(vis,dist)
```

Con estás funciones auxiliares, calcular el *centro* es más sencillo.

Dado el grafo, se buscará para cada vértice el vértice más lejano a él (será el último del recorrdio de la *superBFS*). Se mantendrá con variables auxiliares, el centro actual con su respectiva distancia.

La función regresa un arreglo en donde la primera posición, es la distancia del centro al vértice más lejano y la segunda posición es el centro del grafo.

```
def centro(graph):
    mindist,bestni,bestnf=100000000,0,0
    for v in graph.V:
        resbfs=superBFS(graph,v)
        (fin,dist)=resbfs[-1]
        if dist<mindist:
            mindist=dist
            bestni=v
    return [mindist,bestni]
```

7. Diametro de un grafo

El diametro de un grafo se define como la máxima distancia entre cualquier par de vértices u, v del grafo.

Para conseguir el diametro, podemos buscar para cada vértice su respectivo vértice más alejado, e ir guardando el vértice que cumpla con tener la distancia más grande hasta el momento.

La función diametro recibe un grafo, en el cuál realiza el algoritmo antes brevemente mencionado y regresa un arreglo con los siguientes elementos: El diametro y el camino que cumple ser el diametro.

```
def diametro(graph):
    maxdist,bestni,bestnf=-1,0,0
    for v in graph.V:
        resbfs=superBFS(graph,v)
        (fin,dist)=resbfs[-1]
        if dist>maxdist:
            maxdist=dist
            bestni=v
            bestnf=fin
    return [maxdist,bestni,bestnf]
```

8. Detalles de implementación

Para implementar una Pila tuve que aprender primero a programar una clase, ya que aunque exista en el lenguaje de C++, no me había puesto a investigar acerca de ello. Me resultó interesante las características y funcionalidad que te ofrece la clase.

La implementación de Fila es casi identica a la de una Pila, inclusive con el cambio del nombre y cambiando el método de *obtener* era suficiente.

Para la implementación de un grafo no me resulto complicada, gracias a la página que el profesor nos facilito. Además aprendí las estructuras de sets y diccionario, que para la implementación de ciertos métodos para un grafo resultaron desmasiado efectivas.

Antes de implementar el DFS implemente el BFS, ya que en esa se podía hacer una simulación de una fila de espera de manera tranquila.

En cambio, para la DFS no era lo mismo, ya que por la forma en que se comporta la Pila, no podía hacer uso del arreglo *vis*. Recurrí a meter elementos repetidos en la Pila, y al momento de checarlos, omitirlos si ya estaban visitados. A primera vista, esto aumentaria la complejidad del algoritmo, pero por el hecho de solo visitarlo una vez y las demás omitirla, conserva su complejidad.

A simple vista lo que necesitaba para implementar el Centro y el Diametro era saber la distancia más lejana que puede llegar a tener un vértice en concreto. Teniendo esto en claro, una BFS me podría ayudar en ello. Pero la BFS que había implementado solo me regresaba el vértice lejano, más no la distancia. Por lo que debía hacer algunas modificaciones, a lo que simplemente agregue un arreglo que en paralelo se actualizaba con *vis*. Para regresar ambos arreglos, juntarlo con su respectiva pareja se me hizo una buena idea para resolver el problema que se me presentaba.

Teniendo arreglado esos detalles de implementación, la inicialización de las variables auxiliares y el orden en que debía realizar las tareas no resultó complicado.

Como dato, la complejidad de la BFS y DFS es de $\mathcal{O}(V+E)$ en donde V es la cantidad de vértices y E la cantidad de aristas de un grafo dado.