Estructuras de datos

Luis Enrique López Nerio Universidad Autńoma de Nuevo León
luiselopeznerio@gmail.com

4 de Octubre del 2017

Abstract

En este reporte describiré brevemente algunas estructuras de datos que son útiles dentro de las ciencias computacionales además se introducirá y trabajara con el concepto de grafo; primeramente introduciremos las estructuras de filas y pilas, posteriormente se definirá lo que es un grafo desde un punto de vista matemático además de proporcionar código para su manejo en python. Por ultimo utilizando las estructuras de filas y pilas se describirán algunos algoritmos de gran importancia para el estudio de grafos.

I. Introduccíon

En las ciencias computacionales la abstracción es un aspecto importante que nos permite representar objetos que pueden presentarse dentro del mundo real, la abstracción nos permite simplificar un problema para darle una solución de manera computacional.

Dentro de la abstracción en las ciencias computacionales entran las estructuras de datos, básicamente una estructura de datos es la organización de datos de una manera particular, esto permitirá que la computadora maneje la información de una manera que sea eficiente y útil. Por lo pronto veremos 3 estructuras de datos, estas 3 estructuras son las siguientes:

- Filas
- Pilas
- Grafos

Estas estructuras seran algo asi como un tipo de dato especial que nosotros definiremos, esto es, asi como int, float, list, son tipos de datos que se pueden utilizar en python, nosotros definiremos estas estructuras de datos, a su vez, tal y como los tipos int, float, list, etc, nuestras estructuras contaran con metodos que nos permitiran manipularlas u obtener información de estas.
Al nosotros ser los que definamos los metodos de nuestras estructuras de datos, contaremos con mucha flexibilidad al momento de especificar lo que nosotros deseemos que realizen nuestros metodos.

II. Pilas

La primera estructura que veremos sera la de pila, una pila es basicamente una lista o arreglo, pero esta lista o arreglo cuenta con una forma especifica en la que se pueden accesar o insetar datos, esto es, si en una lista nosotros podiamos tener acceso a cualquier elemento de la lista, eliminarlo, cambiarlo, etc. Esto no sera posible en una pila.

En una pila solo hay una forma en la que se pueden accesar a los datos guardados en ella, la forma en la que se obtendra un dato sera mediante un metodo de extracción, y solo se podra accesar al elemento mas recientemente agregado a la pila. A continuación se presenta el codigo para crear esta estructura de datos en python.

Figure 1: *Torres de Hannoi*

```
class pila(object):
    def __init__(self):
        self.a=[]

    def obtener(self):
        return self.a.pop()

    def meter(self, e):
        self.a.append(e)

    @property
    def longitud(self):
        return len(self.a)

    def __str__(self):
        return "<" + str(self.a)+ ">"
```

Codigo 1.-definición de la clase pila

Algunos ejemplos de situaciones que podrian interpretarse como pilas, son las pilas de platos, el juego de la torres de hannoi, libros puestos uno arriba del otro, etc.

III. Fila

La estructura de fila es una estructura de datos muy parecida a la de pila, al igual que una pila, la fila es una lista o arreglo, la forma en la que obtendremos un dato de una fila será de la siguiente manera, cada vez que queramos obtener un elemento de la fila, solo podremos obtener el elemento que tiene más tiempo en la fila, obviamente esto es muy parecido a una fila de espera, por ejemplo de un supermercado en la cual la forma en la que se atienden a los clientes es atendiendo a los clientes que tienen más tiempo en la fila.

A continuación se presenta el codigo desarrollado para generar una estructura de datos de fila en python, como se puede ver toma todos lo metodos declarados para la clase pila y solo cambia el metodo de extracción de elementos, esto se puede hacer gracias a que la nueva clase fila hereda todos lo metodos de pila mediante la instruccion **class fila (pila)**.

```
class fila(pila):##quitas el que ha estado mas tiempo QUEUE
```

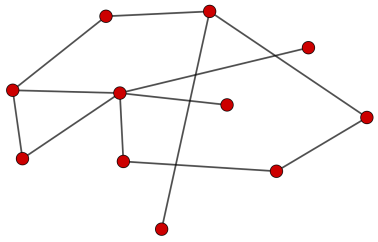

Figure 2: *Grafo1*

```
2     def obtener(self):
3         return self.a.pop(0)
```

Codigo 2.-definición de la clase fila

IV. Grafos

Los grafos son un objeto matemático estudiado por la teoría de grafos, esta tería se fue desarrollando desde el siglo XVII, sim embargo esta rama de las matemáticas empezo a gozar de un gran auge a partir de los aÃśos 1920 con un interes sostenido, este interes fue debido a la vasta cantidad de aplicaciones que iban desde las ciencias computacionales, ingenieria, investigación de operaciones, etc.

Un grafo, en terminología matemática es un conjunto de vé rtices y aristas conectando al menos a un subconjunto de estos puntos,un grafo se puede representar con la siguiente letra: $\mathcal{G}$. De manera mas formal un grafo $\mathcal{G}$ es un par de conjuntos $\mathcal{G} = (V, E)$, V es un conjunto de n vertices y E es un conjunto de aristas que son tipicamente representados como pares de vertices (v_1, v_2).

Dado un grafo $\mathcal{G}$ se define el complemento de un grafo, el cual sera un grafo con los mismos vertices de $\mathcal{G}$ pero con las aristas que no estan en E, esto se puede interpretar como conectar todos los vertices que no estan conectados en el grafo original y quitar las aristas originales. Se definirÃą a un camino de v a w como una sucesiÃşn de aristas adyacentes que empiezan en v y terminan en w y el largo de un camino sera el número de aristas que contiene ese camino. La distancia de dos vertices $dist(v, w)$ entre v y w sera el largo minimo de todos lo caminos que se pueden formar de v a w.

El diametro de un grafo $\mathcal{G}$ se definira como $diam(\mathcal{G})$ diametro de $mathcalG$ es la distancia maxima.

$$diam(\mathcal{G}) = max_{v \in V, w \in V} dist(v, w)$$

En el siguiente codigo se presenta el codigo para formar una clase que tenga las propiedades de un grafo dentro de python.

```
1 class grafo:
```

```
    def __init__(self):
        self.V =set() #un conjunto
        self.E = dict() #un mapeo de pesos a aritstas
        self.vecinos = dict() #un mapeo

    def agrega(self, v ):
        self.V.add(v)
        if not  v in self.vecinos: # vecindad de v
            self.vecinos[v]= set() #inicialmente no tiene nada
    def conecta(self, v , u , peso = 1):
        self.agrega(v)
        self.agrega(u)
        self.E[(v,u)] = self.E[(u,v)] = peso
        self.vecinos[v].add(u)
        self.vecinos[u].add(v)

    def complemento(self):
        comp= grafo()
        for v in self.V:
            for w in self.V:
                if v != w  and (v,w) not in self.E:
                    comp.conecta(v,w,1)
        return comp

    def aristas(self):
        return g.E

    def vertices(self):
        return g.V

    def __str__(self):
        return "Aristas= " + str(self.E)+"\nVertices = " +str(self.V)
```

Codigo 3.-definición de la clase grafo

V. Busqueda por profundidad y anchura

Con las estructuras de datos que creamos (filas, pila, grafos) podemos representar de manera abstracta objectos que se presentan en el mundo real, ahora lo siguiente sera desarrollar algoritmos para obtener información util de estos objectos, los primeros algoritmos que veremos seran la busqueda por profundidad y busqueda por anchura.

La busqueda por anchura de un grafo realiza basicamente el siguiente procedimiento: dado un grafo y un vertice las busqueda por anchura regresara un vector con los vertices del grafo, pero no con cualquier orden, el orden del vector que debe regresar la busquedad por anchura debera de regresar los vectores con distancia 1 del vertice dado, posteriormente los vertices con distancia 2 al vertice dado y asi sucesivamente. Para realizar este algoritmo se hara uso de las estructuras de datos de fila y de grafo, el codigo se presenta a continuación:

```
def BFS_N(g, ni):
    visitados= dict()    ##dicionario con llaves igual a nodos y valores igual a
    distancia de nodo inicial
    Xvisitar=fila()
    Xvisitar.meter(  (ni,0)   )
    while Xvisitar.longitud > 0: ##mientras haya alguien en fila
        nodo = Xvisitar.obtener()
        if nodo[0] not in visitados:
```

```
            visitados[nodo[0]]=nodo[1]
            for vecino in g.vecinos[nodo[0]]:
                #vecinos_d.append( (e,nodo[1]+1) )
                Xvisitar.meter((vecino,nodo[1]+1))
            #for v in vecinos_d:
                #f.meter(v)
    return visitados
```

Codigo 4.-Busqueda por anchura

La busqueda por profundidad es un proceso analogo a la busqueda por anchura, la unica diferencia es que en la busqueda por profunidad lo que se prioratiza es ir visitando nodos cada vez mas profundamente hasta que un camino quede completamente explorado y no se pueda profundizar mas, en lugar de utilizar una estructura de fila para el algoritmo se utilizara una clase pila, a continuación se presenta el codigo para este algoritmo:

```
def DFS_N(g, ni):
    visitados= dict()    ##dicionario con llaves igual a nodos y valores igual a
    distancia de nodo inicial
    Xvisitar=pila()
    Xvisitar.meter(  (ni,0)   )
    while Xvisitar.longitud > 0: ##mientras haya alguien en fila
        nodo = Xvisitar.obtener()
        if nodo[0] not in visitados:
            visitados[nodo[0]]=nodo[1]
            for vecino in g.vecinos[nodo[0]]:
                #vecinos_d.append( (e,nodo[1]+1) )
                Xvisitar.meter((vecino,nodo[1]+1))
            #for v in vecinos_d:
                #f.meter(v)
    return visitados
```

Codigo 5.-Busqueda por profundidad

Tambien se presentan algoritmos para calcular el diametro y los nodos centrales de un grafo, estos algoritmos utilizan el algoritmo de busqueda por anchura que ya se definio:

```
    @property
    def diametro(self):
        maximo = 0
        for vertice in self.V:
            dic_bfs = BFS_N(self, vertice)
            if max(dic_bfs.values())>maximo:
                maximo = max(dic_bfs.values())
        return maximo

    @property
    def centrales(self):
        di_ma=dict()  #distancias maximas de cada vertice
        nodos_centrales=[]
        for v in self.V:
            diccionario = BFS_N(self, v)
            di_ma[v]= max(diccionario.values()  )
        radio = min(di_ma.values())
        for valor in di_ma:
            if di_ma[valor] == radio:
                nodos_centrales.append(valor)
        return nodos_centrales
```

Codigo 6.-Codigo para calcular diametro y nodos centrales

VI. Conclusión

Para finalizar mi conclusión es que las estructuras de datos son objetos dentro de las ciencias computacionales que ofrecen gran flexibilidad, nos permiten abstraer objectos matemÃąticos con gran flexibilidad y definir los metodos, propiedades y caracteristicas que tendran nuestros objectos plasmados en codigo.
Como futura mejora, queda pendiente el desarrollo de las clases definidad, esto ya que carecen de algunos metodos que pueden ser utilies en ciertas estructura; el desarrollo de algoritmos en los cuales se toma en cuenta que un grafo puede ser ponderado y para finalizar el analisis de la complejidad computacional de estos algoritmos.