Alicia Magaly Azúa Saucedo

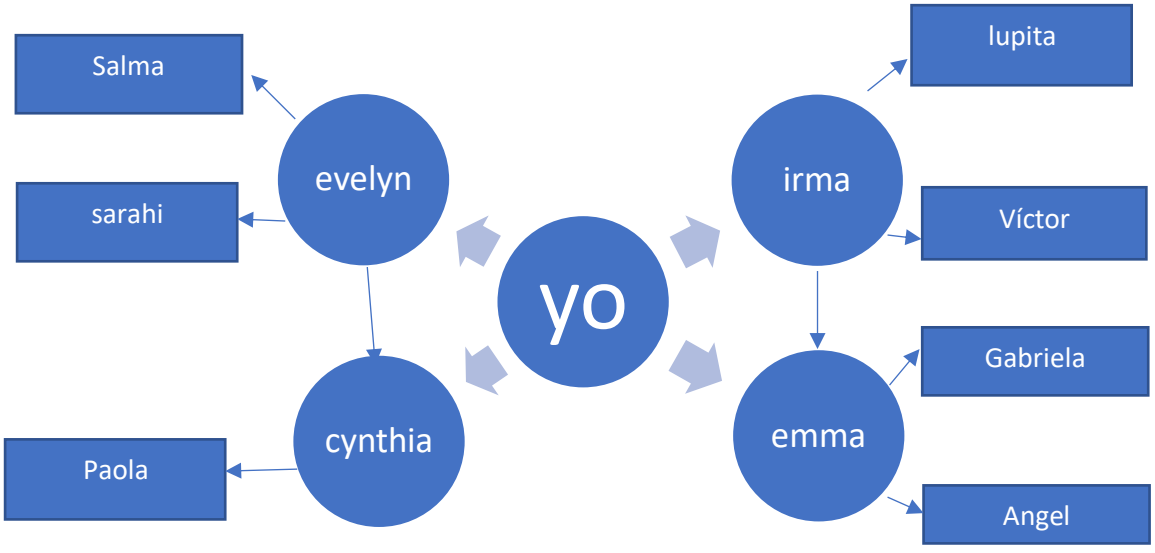
1743217

Reporte de Estructuras de Datos

29 de abril de 2018

Grafos:

Los grafos son un tipo de estructuras de datos que están compuestos de nodos y aristas que se pueden utilizar para diversas cosas, son utilizados en muchas cosas de la vida real muchos de los grafos cuentan con distancias y te puede dar la manera más rápida de cómo se hablan personas o como llegar a lugares o diversos aspectos más; el siguiente grafo representa amigos y quienes hablan a quien y como llegar a hablarse el uno a otro y por medio de quien:



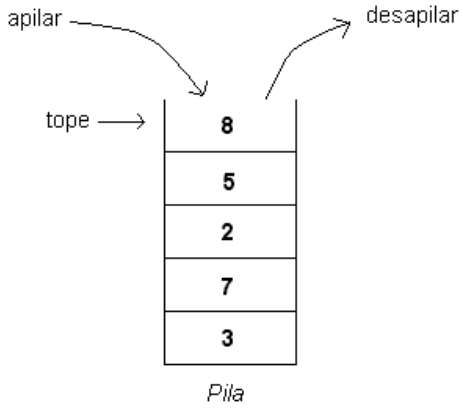
Código:

```
class Grafo:
    def __init__(self):
```

```
        self.V = set() #un conjunto
        self.E = dict() #un mapeo de pesos de aristas
        self.vecinos = dict() # un mapeo
    def agregar(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.agregar(v)
        self.agregar(u)
        self.E[(v, u)] = self.E[(u, v)] = peso  # en ambos sentidos
        self.vecinos[v].add(u)
        self.vecinos[u].add(v)
```

Pila:

una pila es donde el último elemento añadido es el primer elemento retirado ("último en entrar, primero en salir"). Para agregar un ítem a la cima de la pila se utiliza un append (). Para retirar un ítem de la cima de la pila se usa un pop ().



De esa manera trabaja la pila como estructura de datos.

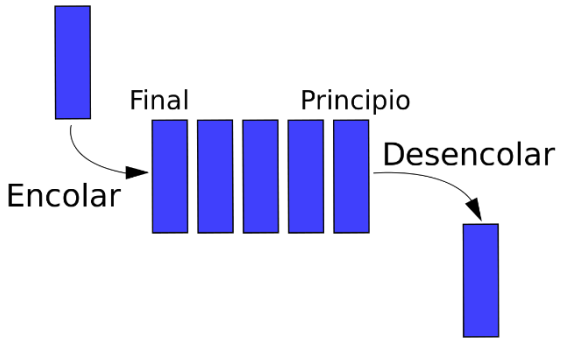
Código:

```
class Pila:
    def __init__(self):
        self.pila = []
    def obtener(self):
```

```
        return self.pila.pop()
    def meter(self,e):
     self.pila.append(e)
        return len(self.pila)
    @property
    def longitud(self):
        return len(self.pila)
```

Fila:

una cola o una fila es donde el primer elemento añadido es el primer elemento retirado (“primero en entrar, primero en salir”); pero las listas no son eficientes para este propósito. Agregar y sacar del final de la lista es rápido, pero insertar o sacar del comienzo de una lista es lento, porque todos los otros elementos tienen que ser desplazados por uno.

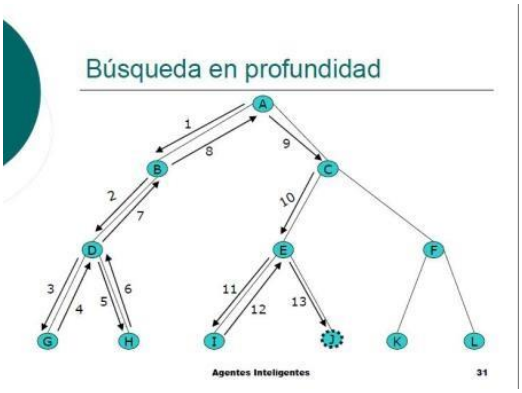


Código:

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

Búsqueda de profundidad (DFS):

Una Búsqueda en profundidad es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa, de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.



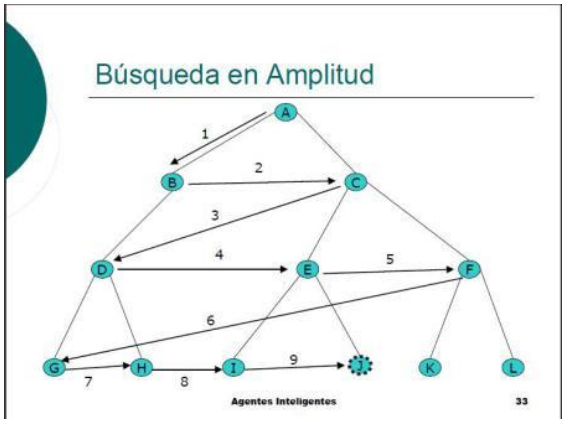
Código:

```
def DFS(g, ni):
    visitados =[]
    f= Pila()
    f.meter(ni)
    while(f.longitud > 0):
          na = f.obtener()
       if na not in visitados:
          visitados.append(na)
          ln = g.vecinos[na]
          for nodo in ln:
             if nodo not in visitados:
                f.meter(nodo)
    return visitados
```

búsqueda de amplitud (BFS):

Búsqueda de amplitud es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo. Intuitivamente, se comienza en la raíz y se exploran todos los vecinos de este nodo. Después, para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución.



Código:

```
def BFS(g, ni):
    visitados = []
    f= Fila()
    f.meter(ni)
    while (f.longitud > 0):
        na = f.obtener()
        if na not in visitados:
            visitados.append(na)
            ln = g.vecinos[na]
            for nodo in ln:
                if nodo not in visitados:
                    f.meter(nodo)
    return visitados
```

densidad:

Sea $G = (V, E)$ un grafo simple con $n=|V|$, $m=|E|$. Definiremos la **densidad del grafo** como

$$d = \frac{2m}{n(n-1)}$$

Notar que $0 < d < 1$, donde $d=0$ si todos los vértices son aislados y $d=1$ si el grafo es completo. Si d es cercano a cero se dice que el grafo es disperso y si d es cercano a 1 se dice que el grafo es denso.

Código:

```
def densidad(self):

    cantidadAristas = len(self.E)

    cantidadVertices = len(self.V)

    cantidadMaximaAristas = cantidadVertices * (cantidadVertices - 1)

    if cantidadMaximaAristas < 1:

       cantidadMaximaAristas = 1

    dens=cantidadAristas/cantidadMaximaAristas

    return dens
```

al momento de ejecutar mi código con el grafo me salió un valor de 0.2 ya que se sustituyen el numero de aristas y el número de vértices en la ecuación

Dijkstra:

Encuentra caminos más cortos desde el vértice de inicio a todos los vértices más cercanos igual o igual al final.

Se supone que el gráfico de entrada G tiene la siguiente representación: A Vértice puede ser cualquier objeto que pueda usarse como un índice en un diccionario. G es un diccionario, indexado por vértices. Para cualquier vértice v, G [v] es en sí mismo un diccionario, indexado por los vecinos de v. Para cualquier borde v-> w, G [v] [w] es la longitud del borde.

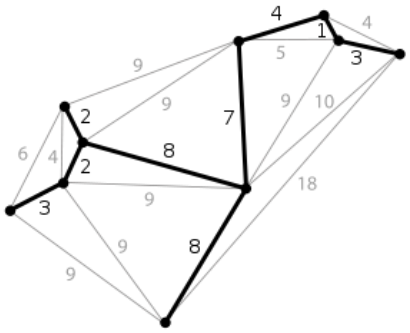
El algoritmo de Dijkstra solo está garantizado que funciona correctamente cuando todas las longitudes de los bordes son positivas.

Código:

```
DIJKSTRA (Grafo G, source_node s)

    for u ∈ V [G] to do

       distance [u] = INFINITY

       father [u] = NULL

       seen [u] = false

    distance [s] = 0

    add (tail, (s, distance [s]))

    while queue is not empty to do

       u = minimum_extract (tail)

       seen [u] = true

       for all v ∈ adjacency [u] to do

          if distance [v]> distance [u] + weight (u, v) do

             distance [v] = distance [u] + weight (u, v)

             father [v] = u

             add (tail, (v, distance [v]))
```

Kruskal:

El algoritmo Kruskal genera el Árbol de Cobertura Mínima de un gráfico conexo. ¿Pero qué es un árbol de cobertura mínima? Supongamos que tenemos un grafo, y queremos reducir su complejidad para poder trabajar con él. Este grafo tiene pesos, por lo tanto, tratamos de encontrar un árbol que pueda cubrir todos sus nodos, pero que tenga como característica utilizar los mínimos pesos para todos los arcos. La siguiente gráfica mostrará la idea general de este algoritmo:



El grafo es conexo y tiene pesos. Un MST es un árbol generado a partir de este grafo que utilice todos los nodos, pero que se conecte por medio de los arcos que tengan el mínimo peso.