Estructuras de datos nodales: Parte 2

Semana 10 - Jueves 17 de octubre de 2019

Anuncios

1. No olviden contestar la encuesta de carga académica: ¡incentivos!
2. Hoy tenemos la última actividad **formativa**.

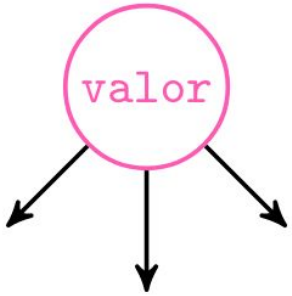
Nodo

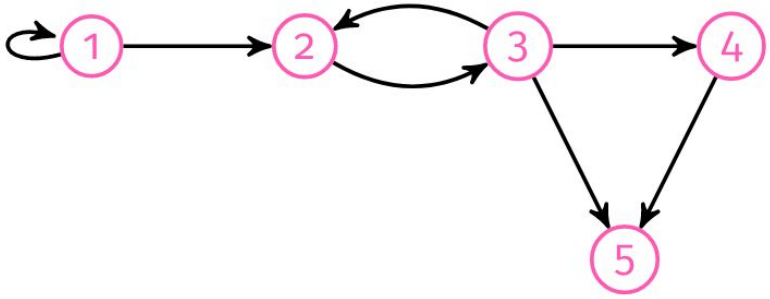
Unidad básica de datos

- Datos
- Vecinos

```python
class Nodo:

    def __init__(self, valor=None):
        """Inicializa la estructura del nodo"""
        self.valor = valor
        self.vecinos = []
```





Grafos

- ¿Posibles usos?

Grafos

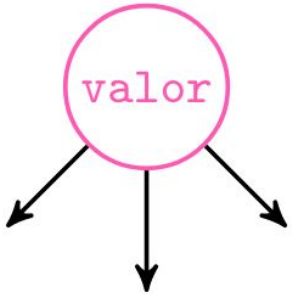
- Posibles formas de representar un grafo

Representación basada en nodos

```
class Nodo:

    def __init__(self, valor=None):
        self.valor = valor
        self.vecinos = []
    def agregar_vecino(self, vecino):
        self.vecinos.append(vecino)

nodo_1 = Nodo(1)
nodo_2 = Nodo(2)
nodo_3 = Nodo(3)
nodo_1.agregar_vecino(nodo_2)
```

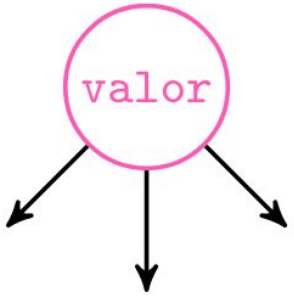


Representación basada en nodos

```
class Nodo:

    def __init__(self, valor=None):
        self.valor = valor
        self.vecinos = []
    def agregar_vecino(self, vecino):
        self.vecinos.append(vecino)

nodo_1 = Nodo(1)
nodo_2 = Nodo(2)
nodo_3 = Nodo(3)
nodo_1.agregar_vecino(nodo_2)
```



¿Cómo represento el grafo completo en un solo objeto?

Representación basada en nodos

```
class Nodo:

    def __init__(self, valor=None):
        self.valor = valor
        self.vecinos = []
    def agregar_vecino(self, vecino):
        self.vecinos.append(vecino)

class Grafo:

    def __init__(self):
        self.nodos = []
```

Representación basada en nodos

```
class Nodo:

    def __init__(self, valor=None):
        self.valor = valor
        self.vecinos = []
    def agregar_vecino(self, vecino):
        self.vecinos.append(vecino)

class Grafo:

    def __init__(self):
        self.nodos = dict()
```

Listas de adyacencia

```
grafo = {
   1: [2, 3],
   2: [4, 5],
   3: [],
   4: [5],
   5: [4]
}

nodos = list(grafo.keys())
```

Listas de adyacencia

```
class Grafo:
    def __init__(self):
        self.listas_de_adyacencia = dict()

    def agregar_vertice(self, valor):
        if valor not in self.listas_de_adyacencia:
            self.listas_de_adyacencia[valor] = list()

    def agregar_arista(self, valor_1, valor_2):
        self.agregar_vertice(valor_1)
        self.agregar_vertice(valor_2)
        self.listas_de_adyacencia[valor_1].append(valor_2)
```

Matriz de adyacencia

```
grafo = [
   [0, 1, 1, 0, 0],
   [0, 0, 0, 1, 1],
   [0, 0, 0, 0, 0],
   [0, 0, 0, 0, 1],
   [0, 0, 0, 1, 0]
]
```

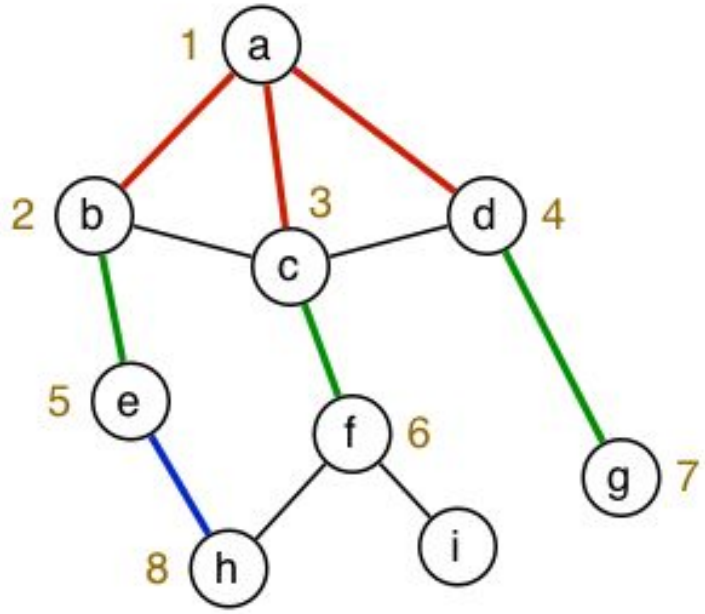
Matriz de adyacencia

```
  grafo = [
1   [0, 1, 1, 0, 0],
2   [0, 0, 0, 1, 1],
3   [0, 0, 0, 0, 0],
4   [0, 0, 0, 0, 1],
5   [0, 0, 0, 1, 0]
  ]   1  2  3  4  5
```

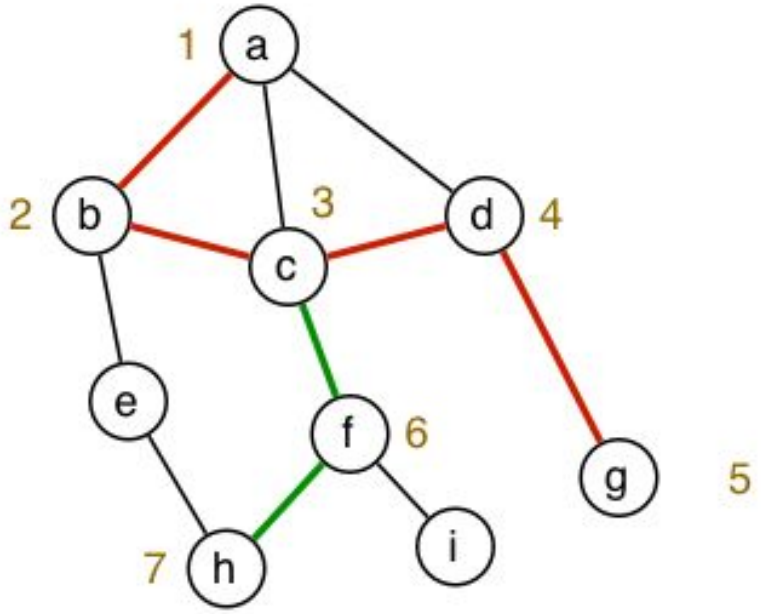
Grafos

- Cómo recorrer un grafo

DFS vs BFS

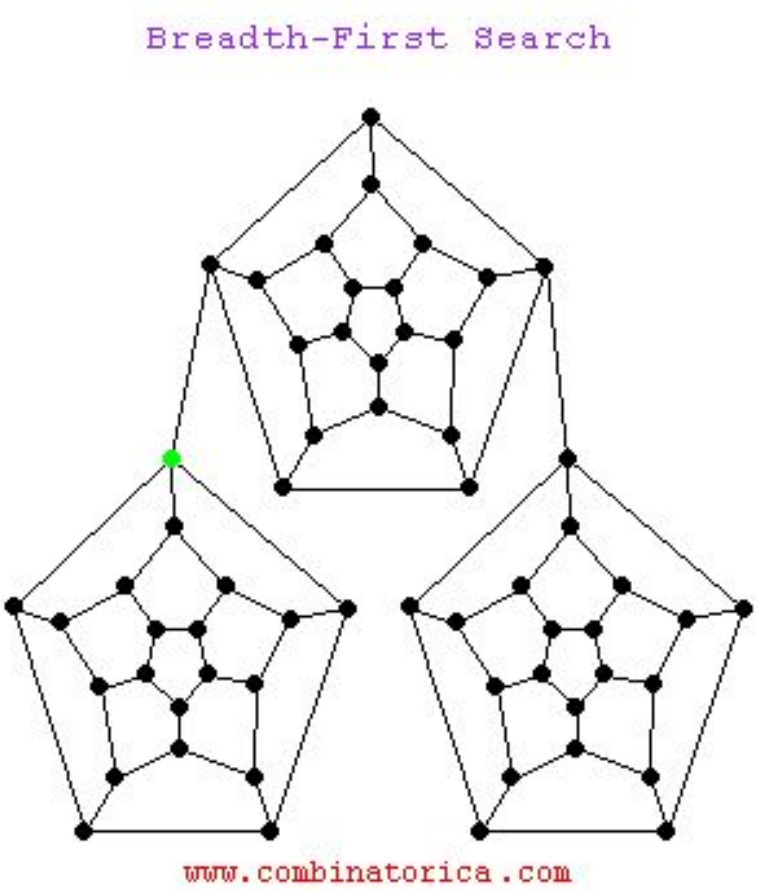


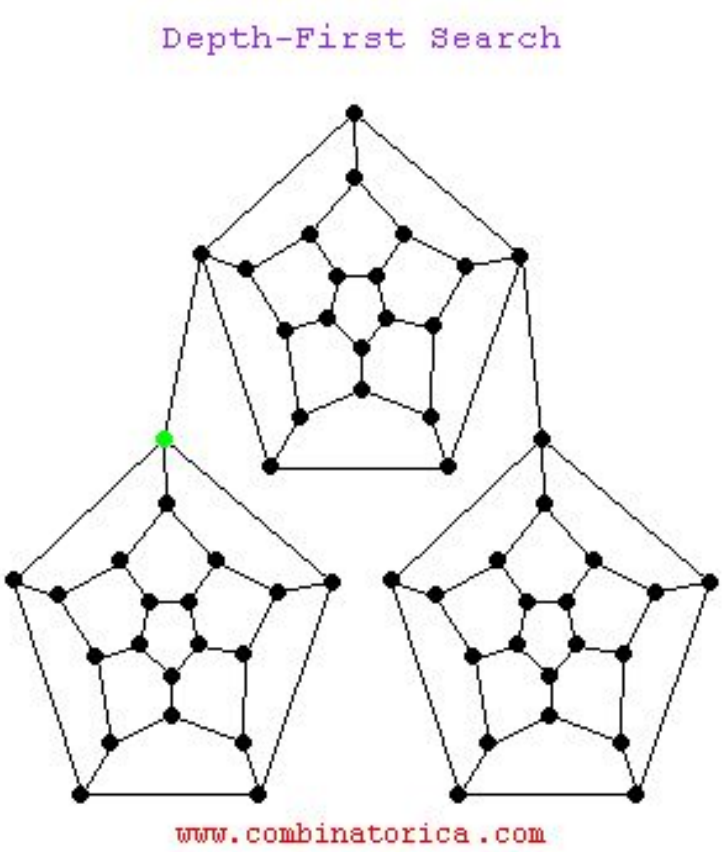
Breadth-first Search



Depth-first Search

DFS vs BFS


Breadth-First Search
www.combinatorica.com


Depth-First Search
www.combinatorica.com

Representación basada en nodos con pesos

```
class Lugar:
    def __init__(self, nombre):
        self.nombre = nombre
        self.calles = []

    def agregar_calle(self, destino):
        self.calles.append(destino)

class Ciudad:
    def __init__(self):
        self.lugares = []
```

Representación basada en nodos con pesos

```
class Lugar:
    def __init__(self, nombre):
        self.nombre = nombre
        self.calles = []

    def agregar_calle(self, distancia, destino):
        self.calles.append((distancia, destino))

class Ciudad:
    def __init__(self):
        self.lugares = []
```

El auto de **Ruz** está malo, y solo es capaz de moverse de un lugar a otro por calles cortas (**menos de 5km**). No puede andar en calles de 5km o más, si no se apaga su auto en la calle.

Escribe un método que desde un **lugar inicial** entregue todos los **lugares alcanzables** por calles de **distancia menor a 5**.

Representación basada en nodos con pesos

```
class Ciudad:
    ...
    def ruz_alcanza(self, inicio):
        pass
```

Representación basada en nodos con pesos

```
class Ciudad:
    ...
    def ruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]




        return visitados
```

Representación basada en nodos con pesos

```
class Ciudad:
    ...
    def ruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]
        while len(stack) > 0:
            lugar = stack.pop()




        return visitados
```

Representación basada en nodos con pesos

```
class Ciudad:
    ...
    def ruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]
        while len(stack) > 0:
            lugar = stack.pop()
            if lugar not in visitados:
                visitados.add(lugar)



        return visitados
```

Representación basada en nodos con pesos

```
class Ciudad:
    ...
    def ruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]
        while len(stack) > 0:
            lugar = stack.pop()
            if lugar not in visitados:
                visitados.add(lugar)
                for distancia, vecino in lugar.calles:


        return visitados
```

Representación basada en nodos con pesos

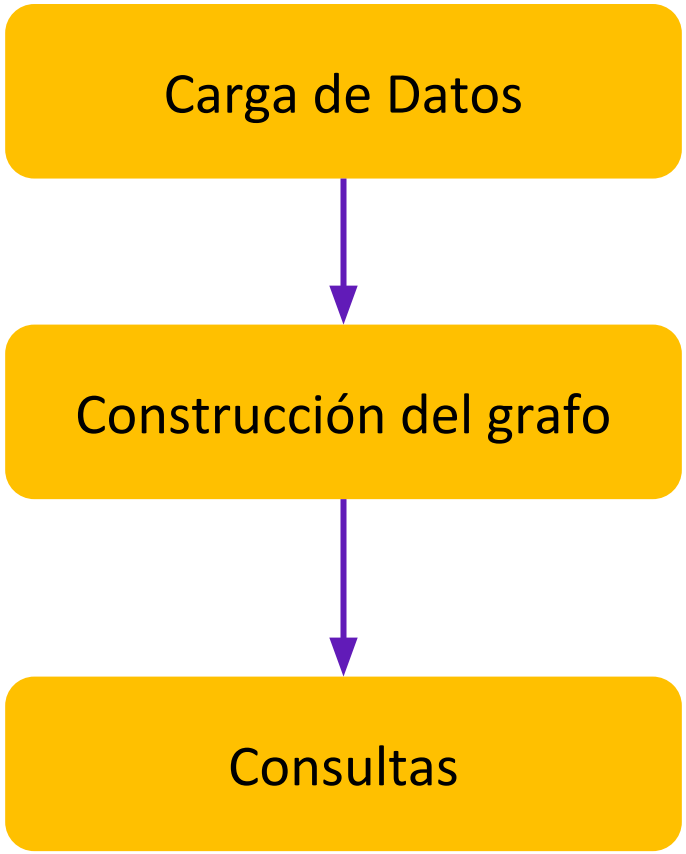
```
class Ciudad:
    ...
    def ruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]
        while len(stack) > 0:
            lugar = stack.pop()
            if lugar not in visitados:
                visitados.add(lugar)
                for distancia, vecino in lugar.calles:
                    if vecino not in visitados and distancia < 5:
                        stack.append(vecino)
        return visitados
```

Actividad

1. En el *syllabus*, vayan a la carpeta “Actividades” y descarguen el enunciado de la actividad 8 (AC08) https://github.com/IIC2233/syllabus
2. Trabajen **individualmente** hasta las 16:30.
3. Recuerden hacer *commit* y *push* cada cierto tiempo.

Cierre

Diagrama de flujo de AC




```
pintogram = Pintogram()
pintogram.cargar_red(path ...)
```

```
def nuevo_usuario(self, id, nombre)
def follow(self, id_1, id_2)
def unfollow(self, id_1, id_2)
```

```
def mis_seguidos(self, id)
```

```
def distancia_social(self, id_1, id_2)
```

Estructura de Grafo

Necesitamos saber cuáles son las aristas de salida de cada nodo

La estructura del grafo nos permite construir métodos eficientes.

No necesitamos iterar por cada nodo del grafo. Podemos aprovechar las relaciones entre nodos.

cargar_red(self, path)

Usamos el **generador** `cargar_archivos` para iterar y nuestras otras funciones para construir el grafo

```
def cargar_red(self, ruta_red):
    conexiones = dict()
    # Primero necesitamos crear todos los usuarios
    for usuario in cargar_archivos(ruta_red):
        id_usuario, nombre, seguidos = usuario
        self.nuevo_usuario(id_usuario, nombre)
        conexiones[id_usuario] = seguidos
    # Después generamos las conexiones
    for id_seguidor, seguidos in conexiones.items():
        for id_seguido in seguidos:
            self.follow(id_seguidor, id_seguido)
```

nuevo_usuario(self, id, nombre)

`nuevo_usuario` debe: (1) crear el nuevo "nodo" del grafo y (2) agregarlo a la lista de nodos. Las conexiones las manejamos con `follow`

```
def nuevo_usuario(self, id, nombre):
    # Creamos el nuevo "nodo"
    usuario = Usuario(id, nombre)
    # Lo agregamos a la lista de nodos
    self.usuarios[usuario.id] = nuevo_usuario
```

follow(self, id1, id2)

La lista de seguidos de cada nodo permite agregar una arista al grafo fácilmente.
Si estamos mantienendo la lista de seguidores, también debemos actualizarla

```
def follow(self, id_seguidor, id_seguido):
    # Buscamos a los nodos en el diccionario
    seguidor = self.usuarios.get(id_seguidor)
    seguido = self.usuarios.get(id_seguido)
    # ¿Por qué no self.usuarios[id]?

    # Supongamos que ambos existen
    if seguidor not in seguido.seguidores:
        seguido.seguidos.append(seguidor)
    # ¿Y si no existiera alguno?
```

unfollow(self, id1, id2)

Similarmente a `follow`, solo debemos manipular la lista de seguidos. Si estamos manteniendo los seguidores, debemos actualizarla también

```
def unfollow(self, id_seguidor, id_seguido):
    # Buscamos a los nodos en el diccionario
    seguidor = self.usuarios.get(id_seguidor)
    seguido = self.usuarios.get(id_seguido)

    # Supongamos que ambos existen
    if seguidor in seguido.seguidores:
        seguido.seguidores.remove(seguidor)
        seguidor.seguidos.remove(seguido)
```

mis_seguidos(self, id)

Con la información que tenemos (lista de adyacencia) es fácil obtener esto. No necesitamos recorrer el grafo porque hemos obtenido la información al construirlo

```
def mis_seguidos(self, id_usuario):
    # Buscamos al nodo en el diccionario
    usuario = self.usuarios.get(id_usuario)
    return len(usuario.seguidos)
```

distancia_social(self, id1, id2)

¿Qué algoritmo usamos? ¿DFS o BFS?

```
def distancia_social(self, id_usuario_1, id_usuario_2):
```

distancia_social(self, id1, id2)

Primero sacamos del camino los casos más simples

```
def distancia_social(self, id_usuario_1, id_usuario_2):
    # Caso trivial
    if usuario_1 == usuario_2:
        return 0
    # Buscamos a los nodos en el diccionario
    usuario_1 = self.usuarios.get(usuario_1)
    usuario_2 = self.usuarios.get(usuario_2)
    if usuario_1 is None or usuario_2 is None:
        return 0
    # ...
```

distancia_social(self, id1, id2)

BFS nos permite recorrer en amplitud. Encontraremos primero los más cercanos.

Pero, **¿cómo obtenemos la distancia?**

```
def distancia_social(self, id_usuario_1, id_usuario_2):
    # ...
    por_visitar = deque()
    por_visitar.append(usuario_1)
    visitados = list()
    while len(por_visitar) > 0:
        usuario_actual = por_visitar.popleft()
        if usuario_actual == usuario_2:
            # Lo encontramos... ¿ahora qué?
        if usuario_actual not in visitados:
            visitados.append(usuario_actual)
            for vecino in usuario_actual.seguidos:
                por_visitar.append(vecino)
    return float("inf")
```

distancia_social(self, id1, id2)

Podemos almacenar **tuplas** que mantengan el nodo y la distancia desde el origen

```
def distancia_social(self, id_usuario_1, id_usuario_2):
    # ...
    por_visitar = deque()
    por_visitar.append((usuario_1, 0))
    visitados = list()
    while len(por_visitar) > 0:
        usuario_actual, distancia = por_visitar.popleft()
        if usuario_actual == usuario_2:
            return distancia
        if usuario_actual not in visitados:
            visitados.append(usuario_actual)
            for vecino in usuario_actual.seguidos:
                por_visitar.append((vecino, distancia+1))
    return float("inf")
```

Próxima semana

1. Hay actividad **sumativa**.
2. Se publicará el material de estudio **mañana**.
3. Tendrán ayudantía el martes.