Árboles y Recorridos en grafos

Eric Brandwein Manuel Nores

Mayo 2024

Recordando...

- **Árbol**:

Recordando...

- **Árbol**: grafo conexo y sin ciclos.
- **Árbol enraizado**:

Recordando...

- **Árbol**: grafo conexo y sin ciclos.
- **Árbol enraizado**: un árbol con un vértice que designamos como raíz. Cada vértice de un árbol enraizado tiene un **padre** (excepto la raíz) y algunos **hijos**.
- **Subgrafo generador**:

Recordando...

- **Árbol**: grafo conexo y sin ciclos.
- **Árbol enraizado**: un árbol con un vértice que designamos como raíz. Cada vértice de un árbol enraizado tiene un **padre** (excepto la raíz) y algunos **hijos**.
- **Subgrafo generador**: subgrafo que contiene todos los vértices del grafo original.
- **Arista puente**:

Recordando...

- **Árbol**: grafo conexo y sin ciclos.
- **Árbol enraizado**: un árbol con un vértice que designamos como raíz. Cada vértice de un árbol enraizado tiene un **padre** (excepto la raíz) y algunos **hijos**.
- **Subgrafo generador**: subgrafo que contiene todos los vértices del grafo original.
- **Arista puente**: arista que al removerla se incrementa la cantidad de componentes conexas.
- **Punto de corte o de articulación**:

Recordando...

- **Árbol**: grafo conexo y sin ciclos.
- **Árbol enraizado**: un árbol con un vértice que designamos como raíz. Cada vértice de un árbol enraizado tiene un **padre** (excepto la raíz) y algunos **hijos**.
- **Subgrafo generador**: subgrafo que contiene todos los vértices del grafo original.
- **Arista puente**: arista que al removerla se incrementa la cantidad de componentes conexas.
- **Punto de corte o de articulación**: vértice que al removerlo se incrementa la cantidad de componentes conexas.

Breadth First Search (BFS)

Breadth First Search (BFS)

Recorrer primero los vértices vecinos, después los vecinos de los vecinos, etc.

Breadth First Search (BFS)

Recorrer primero los vértices vecinos, después los vecinos de los vecinos, etc.

```
1  from queue import Queue
2
3  def bfs(lista_de_adyacencias, raiz):
4      a_visitar = Queue()
5      visitados = [False] * len(lista_de_adyacencias)
6      a_visitar.put(raiz)
7      visitados[raiz] = True
8      while not a_visitar.empty():
9          actual = a_visitar.get()
10         # Hacer algo con el vértice actual
11         for vecino in lista_de_adyacencias[actual]:
12             if not visitados[vecino]:
13                 a_visitar.put(vecino)
14                 visitados[vecino] = True
```

Breadth First Search (BFS)

Recorrer primero los vértices vecinos, después los vecinos de los vecinos, etc.

```
1  from queue import Queue
2
3  def bfs(lista_de_adyacencias, raiz):
4      a_visitar = Queue()
5      visitados = [False] * len(lista_de_adyacencias)
6      a_visitar.put(raiz)
7      visitados[raiz] = True
8      while not a_visitar.empty():
9          actual = a_visitar.get()
10         # Hacer algo con el vértice actual
11         for vecino in lista_de_adyacencias[actual]:
12             if not visitados[vecino]:
13                 a_visitar.put(vecino)
14                 visitados[vecino] = True
```

- Complejidad temporal:

Breadth First Search (BFS)

Recorrer primero los vértices vecinos, después los vecinos de los vecinos, etc.

```python
from queue import Queue

def bfs(lista_de_adyacencias, raiz):
    a_visitar = Queue()
    visitados = [False] * len(lista_de_adyacencias)
    a_visitar.put(raiz)
    visitados[raiz] = True
    while not a_visitar.empty():
        actual = a_visitar.get()
        # Hacer algo con el vértice actual
        for vecino in lista_de_adyacencias[actual]:
            if not visitados[vecino]:
                a_visitar.put(vecino)
                visitados[vecino] = True
```

- Complejidad temporal: $\Theta(n + m)$
- Complejidad espacial:

Breadth First Search (BFS)

Recorrer primero los vértices vecinos, después los vecinos de los vecinos, etc.

```python
from queue import Queue

def bfs(lista_de_adyacencias, raiz):
    a_visitar = Queue()
    visitados = [False] * len(lista_de_adyacencias)
    a_visitar.put(raiz)
    visitados[raiz] = True
    while not a_visitar.empty():
        actual = a_visitar.get()
        # Hacer algo con el vértice actual
        for vecino in lista_de_adyacencias[actual]:
            if not visitados[vecino]:
                a_visitar.put(vecino)
                visitados[vecino] = True
```

- Complejidad temporal: $\Theta(n + m)$
- Complejidad espacial: $O(n)$

Depth First Search (DFS)

Depth First Search (DFS)

Recorrer un vecino, después un vecino de ese vecino, etc. hasta no poder más, y después volver y seguir por otro vecino, etc.

Depth First Search (DFS)

Recorrer un vecino, después un vecino de ese vecino, etc. hasta no poder más, y después volver y seguir por otro vecino, etc.

```
1  def dfs(lista_de_adyacencias, vertice, visitados=None):
2      if visitados is None:
3          visitados = [False] * len(lista_de_adyacencias)
4
5      visitados[vertice] = True
6      # Hacer algo con el vértice
7      for hijo in lista_de_adyacencias[vertice]:
8          if not visitados[hijo]:
9              dfs(lista_de_adyacencias, hijo, visitados)
```

Depth First Search (DFS)

Recorrer un vecino, después un vecino de ese vecino, etc. hasta no poder más, y después volver y seguir por otro vecino, etc.

```
1  def dfs(lista_de_adyacencias, vertice, visitados=None):
2      if visitados is None:
3          visitados = [False] * len(lista_de_adyacencias)
4
5      visitados[vertice] = True
6      # Hacer algo con el vértice
7      for hijo in lista_de_adyacencias[vertice]:
8          if not visitados[hijo]:
9              dfs(lista_de_adyacencias, hijo, visitados)
```

- Complejidad temporal:

Depth First Search (DFS)

Recorrer un vecino, después un vecino de ese vecino, etc. hasta no poder más, y después volver y seguir por otro vecino, etc.

```
1  def dfs(lista_de_adyacencias, vertice, visitados=None):
2      if visitados is None:
3          visitados = [False] * len(lista_de_adyacencias)
4
5      visitados[vertice] = True
6      # Hacer algo con el vértice
7      for hijo in lista_de_adyacencias[vertice]:
8          if not visitados[hijo]:
9              dfs(lista_de_adyacencias, hijo, visitados)
```

- Complejidad temporal: $\Theta(n + m)$
- Complejidad espacial:

Depth First Search (DFS)

Recorrer un vecino, después un vecino de ese vecino, etc. hasta no poder más, y después volver y seguir por otro vecino, etc.

```
1  def dfs(lista_de_adyacencias, vertice, visitados=None):
2      if visitados is None:
3          visitados = [False] * len(lista_de_adyacencias)
4
5      visitados[vertice] = True
6      # Hacer algo con el vértice
7      for hijo in lista_de_adyacencias[vertice]:
8          if not visitados[hijo]:
9              dfs(lista_de_adyacencias, hijo, visitados)
```

- Complejidad temporal: $\Theta(n + m)$
- Complejidad espacial: $O(n)$

BFS - Padres

```
1  from queue import Queue
2
3  def bfs(lista_de_adyacencias, raiz):
4      a_visitar = Queue()
5      visitados = [False] * len(lista_de_adyacencias)
6      a_visitar.put(raiz)
7      visitados[raiz] = True
8      while not a_visitar.empty():
9          actual = a_visitar.get()
10         # Hacer algo con el vértice actual
11         for vecino in lista_de_adyacencias[actual]:
12             if not visitados[vecino]:
13                 a_visitar.put(vecino)
14                 visitados[vecino] = True
```

BFS - Padres

```
1  from queue import Queue
2
3  def bfs(lista_de_adyacencias, raiz):
4      a_visitar = Queue()
5      visitados = [False] * len(lista_de_adyacencias)
6      a_visitar.put(raiz)
7      visitados[raiz] = True
8      padres = [None] * len(lista_de_adyacencias)
9      while not a_visitar.empty():
10         actual = a_visitar.get()
11         for vecino in lista_de_adyacencias[actual]:
12             if not visitados[vecino]:
13                 a_visitar.put(vecino)
14                 visitados[vecino] = True
15                 padres[vecino] = actual
16     return padres
```

¿Cambia la complejidad?

BFS - Padres

```
1  from queue import Queue
2
3  def bfs(lista_de_adyacencias, raiz):
4      a_visitar = Queue()
5      visitados = [False] * len(lista_de_adyacencias)
6      a_visitar.put(raiz)
7      visitados[raiz] = True
8      padres = [None] * len(lista_de_adyacencias)
9      while not a_visitar.empty():
10         actual = a_visitar.get()
11         for vecino in lista_de_adyacencias[actual]:
12             if not visitados[vecino]:
13                 a_visitar.put(vecino)
14                 visitados[vecino] = True
15                 padres[vecino] = actual
16     return padres
```

¿Cambia la complejidad? **No!**

BFS - Padres

```python
from queue import Queue

def bfs(lista_de_adyacencias, raiz):
    a_visitar = Queue()
    visitados = [False] * len(lista_de_adyacencias)
    a_visitar.put(raiz)
    visitados[raiz] = True
    padres = [None] * len(lista_de_adyacencias)
    while not a_visitar.empty():
        actual = a_visitar.get()
        for vecino in lista_de_adyacencias[actual]:
            if not visitados[vecino]:
                a_visitar.put(vecino)
                visitados[vecino] = True
                padres[vecino] = actual
    return padres
```

¿Cambia la complejidad? **No!**

¿Se puede hacer lo mismo con DFS?

BFS - Padres

```python
from queue import Queue

def bfs(lista_de_adyacencias, raiz):
    a_visitar = Queue()
    visitados = [False] * len(lista_de_adyacencias)
    a_visitar.put(raiz)
    visitados[raiz] = True
    padres = [None] * len(lista_de_adyacencias)
    while not a_visitar.empty():
        actual = a_visitar.get()
        for vecino in lista_de_adyacencias[actual]:
            if not visitados[vecino]:
                a_visitar.put(vecino)
                visitados[vecino] = True
                padres[vecino] = actual
    return padres
```

¿Cambia la complejidad? **No!**

¿Se puede hacer lo mismo con DFS? **Sí!**

BFS - Distancia a un vértice

```python
from queue import Queue

def bfs(lista_de_adyacencias, raiz):
    a_visitar = Queue()
    visitados = [False] * len(lista_de_adyacencias)
    a_visitar.put(raiz)
    visitados[raiz] = True
    while not a_visitar.empty():
        actual = a_visitar.get()
        # Hacer algo con el vértice actual
        for vecino in lista_de_adyacencias[actual]:
            if not visitados[vecino]:
                a_visitar.put(vecino)
                visitados[vecino] = True
```

BFS - Distancia a un vértice

```
1  from queue import Queue
2
3  def bfs(lista_de_adyacencias, raiz):
4      a_visitar = Queue()
5      distancias = [-1] * len(lista_de_adyacencias)
6      a_visitar.put(raiz)
7      distancias[raiz] = 0
8      while not a_visitar.empty():
9          actual = a_visitar.get()
10         for vecino in lista_de_adyacencias[actual]:
11             if distancias[vecino] == -1:
12                 a_visitar.put(vecino)
13                 distancias[vecino] = distancias[actual] + 1
14     return distancias
```

¿Cambia la complejidad?

BFS - Distancia a un vértice

```
1  from queue import Queue
2
3  def bfs(lista_de_adyacencias, raiz):
4      a_visitar = Queue()
5      distancias = [-1] * len(lista_de_adyacencias)
6      a_visitar.put(raiz)
7      distancias[raiz] = 0
8      while not a_visitar.empty():
9          actual = a_visitar.get()
10         for vecino in lista_de_adyacencias[actual]:
11             if distancias[vecino] == -1:
12                 a_visitar.put(vecino)
13                 distancias[vecino] = distancias[actual] + 1
14     return distancias
```

¿Cambia la complejidad? **No!**

BFS - Distancia a un vértice

```
1  from queue import Queue
2
3  def bfs(lista_de_adyacencias, raiz):
4      a_visitar = Queue()
5      distancias = [-1] * len(lista_de_adyacencias)
6      a_visitar.put(raiz)
7      distancias[raiz] = 0
8      while not a_visitar.empty():
9          actual = a_visitar.get()
10         for vecino in lista_de_adyacencias[actual]:
11             if distancias[vecino] == -1:
12                 a_visitar.put(vecino)
13                 distancias[vecino] = distancias[actual] + 1
14     return distancias
```

¿Cambia la complejidad? **No!**

¿Se puede hacer lo mismo con DFS?

BFS - Distancia a un vértice

```python
from queue import Queue

def bfs(lista_de_adyacencias, raiz):
    a_visitar = Queue()
    distancias = [-1] * len(lista_de_adyacencias)
    a_visitar.put(raiz)
    distancias[raiz] = 0
    while not a_visitar.empty():
        actual = a_visitar.get()
        for vecino in lista_de_adyacencias[actual]:
            if distancias[vecino] == -1:
                a_visitar.put(vecino)
                distancias[vecino] = distancias[actual] + 1
    return distancias
```

¿Cambia la complejidad? **No!**

¿Se puede hacer lo mismo con DFS? **No!**

Menú BFS

- Encontrar componentes (fuertemente) conexas.

Menú BFS

- Encontrar componentes (fuertemente) conexas. Por ejemplo (Sharir, 1981).

Menú BFS

- Encontrar componentes (fuertemente) conexas. Por ejemplo (Sharir, 1981).
- Ver si un grafo es conexo.

Menú BFS

- Encontrar componentes (fuertemente) conexas. Por ejemplo (Sharir, 1981).
- Ver si un grafo es conexo.
- Encontrar distancia de aristas uno a todos.

Menú BFS

- Encontrar componentes (fuertemente) conexas. Por ejemplo (Sharir, 1981).
- Ver si un grafo es conexo.
- Encontrar distancia de aristas uno a todos.
- Ver si un (di)grafo tiene ciclos.
- etc.

Tipos de aristas en DFS

Tipos de aristas en DFS

- **Tree edges**: las usadas para descubrir nuevos vértices.
- **Backward edges**: las que van de un vértice a un ancestro.
- **Forward edges**: las que van de un vértice a un descendiente que no sea un hijo.
- **Cross edges**: las que van de un vértice a otro vértice ya visitado que no es ni descendiente ni ancestro.

Tipos de aristas en DFS

- **Tree edges**: las usadas para descubrir nuevos vértices.
- **Backward edges**: las que van de un vértice a un ancestro.
- **Forward edges**: las que van de un vértice a un descendiente que no sea un hijo.
- **Cross edges**: las que van de un vértice a otro vértice ya visitado que no es ni descendiente ni ancestro.

En grafos simples hay sólo **tree edges** y **backward edges**.

DFS - Tiempos

Para cada vértice queremos guardar un par (`tiempo_desde`, `tiempo_hasta`).

```
1  def dfs(lista_de_adyacencias, vertice, visitados=None):
2      if visitados is None:
3          visitados = [False] * len(lista_de_adyacencias)
4
5      visitados[vertice] = True
6      # Hacer algo con el vértice
7      for hijo in lista_de_adyacencias[vertice]:
8          if not visitados[hijo]:
9              dfs(lista_de_adyacencias, hijo, visitados)
```

DFS - Tiempos

Para cada vértice queremos guardar un par (`tiempo_desde`, `tiempo_hasta`).

```
1  def dfs(lista_de_adyacencias, vertice, tiempo_actual=0, tiempos=None):
2      if tiempos is None:
3          tiempos = [(-1, -1) for _ in range(len(lista_de_adyacencias))]
4
5      tiempos[vertice] = (tiempo_actual, -1)
6      nuevo_tiempo = tiempo_actual + 1
7      for hijo in lista_de_adyacencias[vertice]:
8          if tiempos[hijo][0] == -1:
9              nuevo_tiempo = dfs(
10                 lista_de_adyacencias, hijo, nuevo_tiempo, tiempos
11             ) + 1
12
13     tiempos[vertice] = (tiempo_actual, nuevo_tiempo)
14     return nuevo_tiempo
```

Menú DFS

- Encontrar componentes (fuertemente) conexas.

Menú DFS

- Encontrar componentes (fuertemente) conexas. (Tarjan, 1972)

Menú DFS

- Encontrar componentes (fuertemente) conexas. (Tarjan, 1972)
- Ver si un grafo es conexo.

Menú DFS

- Encontrar componentes (fuertemente) conexas. (Tarjan, 1972)
- Ver si un grafo es conexo.
- Ver si un (di)grafo tiene ciclos.

Menú DFS

- Encontrar componentes (fuertemente) conexas. (Tarjan, 1972)
- Ver si un grafo es conexo.
- Ver si un (di)grafo tiene ciclos.
- Encontrar puentes y puntos de articulación.
- etc.

Puentes

Guía 4 - Problema 2

Dado un grafo G, enumerar las aristas puente de G.

Gracias!