Representaciones de grafos y recorridos

Cómo se guarda un grafo y cómo se visita

Carlos A Delgado S

Pontificia Universidad Javeriana Cali

Septiembre de 2026

Objetivos de la sesión

Al terminar la clase, usted podrá

- Escribir un grafo en las cuatro representaciones y decir cuánta memoria ocupa cada una.
- Escoger la representación que le sirve a un algoritmo, con el costo de cada consulta en la mano.
- Recorrer un grafo a mano en profundidad y en amplitud, y decir en qué se diferencian los dos órdenes de visita.
- Escribir los dos recorridos en pseudocódigo y llevarlos a Python sobre lista de adyacencia.
- Justificar por qué ambos cuestan $\Theta(V + E)$ sobre listas y $\Theta(V^2)$ sobre matriz.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

Lo que quedó de la sesión anterior

En una línea cada cosa

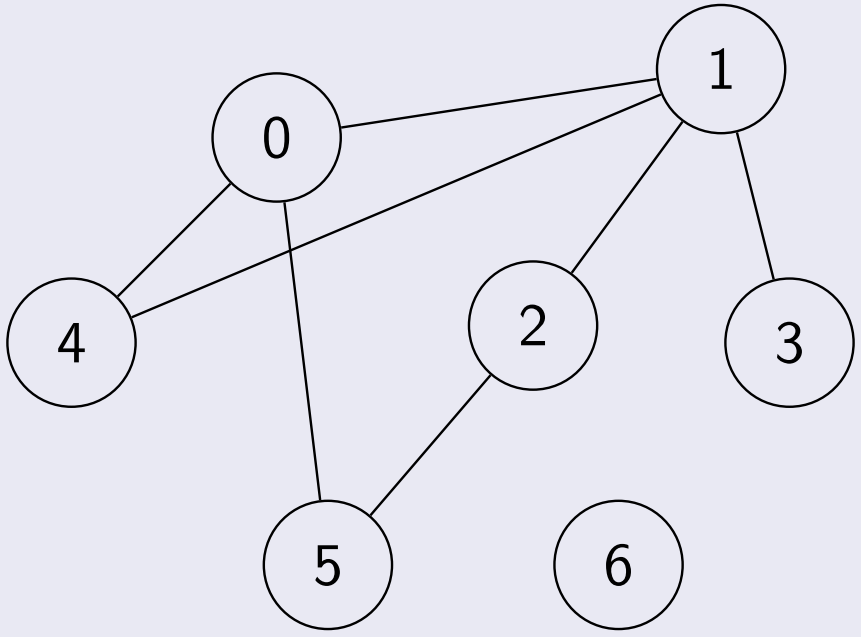
- Un grafo es un par $G = (V, E)$: un conjunto de vértices y un conjunto de aristas que los unen de a dos.
- En un grafo **no dirigido** la arista $\{u, v\}$ se recorre en los dos sentidos; en uno **dirigido**, (u, v) va de u a v y nada más.
- u y v son **adyacentes** si hay una arista entre ellos. $N(u)$ son los vecinos de u y $\delta(u)$ es su **grado**: cuántas aristas lo tocan.
- Sumar los grados cuenta cada arista dos veces: $\sum_{v \in V} \delta(v) = 2|E|$.

En el deck de repaso

Las definiciones completas, los cinco tipos de grafo, las familias (K_n, C_n, W_n, bipartitos), los subgrafos y los complementos.

Los dos grafos de hoy

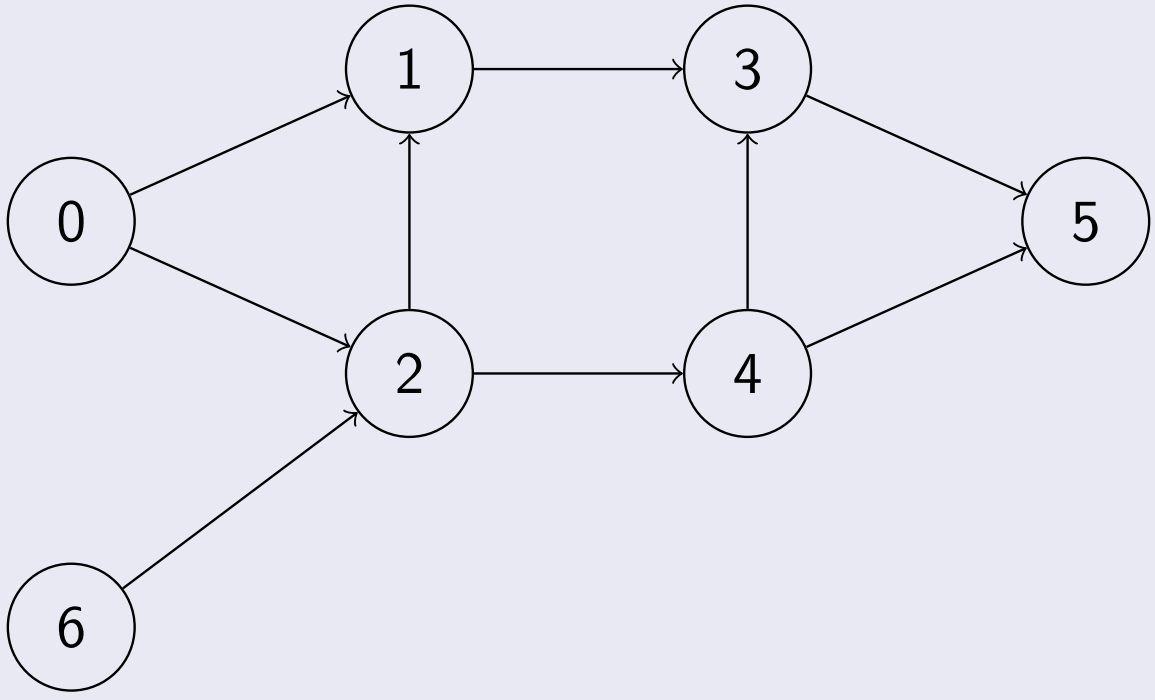
G_1: siete vértices, siete aristas, no dirigido



Aristas: $\{0,1\}, \{0,4\}, \{0,5\}, \{1,2\}, \{1,3\}, \{1,4\}, \{2,5\}$. El vértice 6 queda aislado.

Los dos grafos de hoy

G_2: siete vértices, nueve aristas, dirigido



Aristas: $(0,1), (0,2), (1,3), (2,1), (2,4), (3,5), (4,3), (4,5), (6,2)$. Al 6 no llega ninguna arista.

Los dos grafos de hoy

Los vértices se numeran desde cero

Los enunciados suelen numerar de 1 a n y Python indexa desde 0. Se corren los índices al leer, una sola vez, y de ahí en adelante todo el programa trabaja en $0..n-1$.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

Lista de adyacencia

Definición (Lista de adyacencia, CLRS Sección 22.1, p. 589)

El grafo se representa como una lista G de listas, donde $G[u]$ contiene los vértices adyacentes a u.

G_1 y G_2

```
G1 = [[1, 4, 5],      # 0      G2 = [[1, 2],      # 0
      [0, 2, 3, 4],   # 1            [3],         # 1
      [1, 5],         # 2            [1, 4],      # 2
      [1],            # 3            [5],         # 3
      [0, 1],         # 4            [3, 5],      # 4
      [0, 2],         # 5            [],          # 5
      []]             # 6            [2]]         # 6
```

Lista de adyacencia

Cuánto ocupa

Una entrada por vértice y, dentro de ellas, una por cada extremo de cada arista. En el grafo no dirigido cada arista aparece dos veces y en el dirigido una sola, de modo que el espacio es $\Theta(V + E)$ en los dos casos.

Matriz de adyacencia

Definición (Matriz de adyacencia, CLRS Sección 22.1, p. 589)

Una matriz m de $|V| \times |V|$ donde m_{uv} vale 1 si existe la arista de u a v, y 0 si no. En el grafo no dirigido la matriz es simétrica.

G_1 en matriz

	0	1	2	3	4	5	6
0	0	1	0	0	1	1	0
1	1	0	1	1	1	0	0
2	0	1	0	0	0	1	0
3	0	1	0	0	0	0	0
4	1	1	0	0	0	0	0
5	1	0	1	0	0	0	0
6	0	0	0	0	0	0	0

Matriz de adyacencia

Cuánto ocupa

$\Theta(V^2)$, sin importar cuántas aristas haya. La fila del vértice 6 son siete ceros y ocupa lo mismo que cualquier otra.

Lista de aristas

Definición (Lista de aristas)

El grafo se representa como una lista cuyos elementos son las aristas. En un grafo con peso cada elemento es una tripla $(u, v, peso)$.

G_2 en lista de aristas

```
G2 = [(0, 1), (0, 2), (1, 3), (2, 1), (2, 4),
      (3, 5), (4, 3), (4, 5), (6, 2)]
```

Cuánto ocupa

$\Theta(E)$, y nada más. Es la más compacta y la más parecida a como llega la entrada de un problema: casi siempre hay que convertirla antes de trabajar.

Matriz de incidencia

Definición (Matriz de incidencia)

Una matriz M de $|V| \times |E|$ donde M_{ij} vale 1 si la arista j es incidente con el vértice i, y 0 en caso contrario.

El grado y los extremos, leídos de la matriz

La suma de la fila i es el grado del vértice i. La columna de una arista tiene exactamente dos unos, uno por cada extremo. En grafos dirigidos se usa 1 para el origen y -1 para el destino.

Casi nunca se usa para programar

Ocupa $\Theta(V \cdot E)$ y no contesta rápido ninguna de las dos preguntas que hacen los algoritmos. Aparece cuando la matriz misma es el dato de entrada del problema.

Construir las tres desde la lista de aristas

Lo que llega y lo que se arma

La entrada de un problema casi siempre trae n y la lista de aristas. Las tres construcciones recorren esa lista una vez; la del grafo no dirigido escribe cada arista en los dos sentidos.

```
def lista_de_adyacencia(n, aristas, dirigido):
    # G[u] guarda los vecinos de u: una lista por vertice
    G = []
    u = 0
    while u < n:
        G.append([])
        u = u + 1
    for arista in aristas:
        u = arista[0]
        v = arista[1]
        G[u].append(v)
        if not dirigido:
            G[v].append(u)
    return G
```

Construir las tres desde la lista de aristas

```
def matriz_de_adyacencia(n, aristas, dirigido):
    # m[u][v] vale 1 si la arista existe y 0 si no existe
    m = []
    u = 0
    while u < n:
        fila = []
        v = 0
        while v < n:
            fila.append(0)
            v = v + 1
        m.append(fila)
        u = u + 1
    for arista in aristas:
        u = arista[0]
        v = arista[1]
        m[u][v] = 1
        if not dirigido:
            m[v][u] = 1
    return m
```

Construir las tres desde la lista de aristas

```
def lista_de_aristas(n, aristas, dirigido):
    # El grafo es la lista de sus aristas; sin direccion se guarda el par al
    ↪  reves
    E = []
    for arista in aristas:
        u = arista[0]
        v = arista[1]
        E.append((u, v))
        if not dirigido:
            E.append((v, u))
    return E
```

Lo que cuesta armarlas

La lista de adyacencia y la de aristas cuestan $\Theta(V + E)$: crear las V listas vacías y recorrer las E aristas. La matriz cuesta $\Theta(V^2)$ y no puede costar menos, porque hay que escribir el cero en cada posición que no es arista antes de poner los unos.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

La pregunta que quedó pendiente

Dos criterios

1. **Cuánta memoria** pide cada representación.
2. **Qué tan fácil** es llegar a las conexiones de un vértice.

Con siete vértices las tres sirven igual

La diferencia aparece cuando un algoritmo recorre el grafo entero, y ahí la elección decide si el programa termina o no.

Primero, la memoria

Los tres órdenes

Representación	Espacio
Lista de adyacencia	$\Theta(V + E)$
Matriz de adyacencia	$\Theta(V^2)$
Lista de aristas	$\Theta(E)$

Con números

Una red de $V = 10^5$ vértices y $E = 2 \cdot 10^5$ aristas. La lista de adyacencia guarda $4 \cdot 10^5$ entradas; la matriz guarda 10^{10} posiciones, de las cuales el 99,9998 % son ceros. La matriz no cabe en memoria.

Primero, la memoria

Denso y ralo

Un grafo es **denso** cuando E se acerca a V^2 y **ralo** cuando E es del orden de V. Las redes viales, las redes sociales y casi todo lo que llega de un problema real son ralos.

¿Existe la arista (u, v)?

Lista de adyacencia

```
def hay_arista_en_lista(G, u, v):
    # Recorre los vecinos de u buscando a v
    encontrada = False
    for w in G[u]:
        if w == v:
            encontrada = True
    return encontrada
```

Matriz de adyacencia

```
def hay_arista_en_matriz(m, u, v):
    # Una sola consulta: la posicion ya dice la respuesta
    return m[u][v] == 1
```

Lista de aristas

```
def hay_arista_en_lista_de_aristas(E, u, v):
    # Recorre todas las aristas del grafo, no solo las de u
    encontrada = False
    for arista in E:
        if arista[0] == u and arista[1] == v:
            encontrada = True
    return encontrada
```

¿Existe la arista (u, v)?

Lo que cuesta cada una

La matriz contesta en $O(1)$. La lista de adyacencia recorre los vecinos de u, que en el peor caso son $V - 1$. La lista de aristas recorre las E aristas del grafo.

¿Cuáles son los vecinos de u?

Lista de adyacencia

```
def vecinos_en_lista(G, u):
    # Los vecinos ya estan juntos: la lista de u es la respuesta
    return G[u]
```

Matriz de adyacencia

```
def vecinos_en_matriz(m, u):
    # Hay que revisar la fila completa, incluidos los ceros
    resultado = []
    v = 0
    while v < len(m):
        if m[u][v] == 1:
            resultado.append(v)
        v = v + 1
    return resultado
```

Lista de aristas

```
def vecinos_en_lista_de_aristas(E, u):
    # Hay que revisar todas las aristas del grafo
    resultado = []
    for arista in E:
        if arista[0] == u:
            resultado.append(arista[1])
    return resultado
```

¿Cuáles son los vecinos de u?

La pregunta que hacen los recorridos

Un recorrido pregunta por los vecinos de cada vértice, una vez por vértice. Sobre listas la suma de todas esas consultas es $\Theta(V + E)$; sobre la matriz es $\Theta(V^2)$ aunque el grafo tenga tres aristas.

Las tres, lado a lado

	Lista de ady.	**Matriz de ady.**	**Lista de aristas**
Espacio	$\Theta(V+E)$	$\Theta(V^2)$	$\Theta(E)$
¿Existe (u,v)?	$O(\delta(u))$	$O(1)$	$O(E)$
Vecinos de u	$\Theta(\delta(u))$	$\Theta(V)$	$\Theta(E)$
Recorrer todo	$\Theta(V+E)$	$\Theta(V^2)$	$\Theta(V \cdot E)$
Agregar arista	$O(1)$	$O(1)$	$O(1)$

La regla corta

Si el grafo es ralo, lista de adyacencia. La matriz se justifica cuando V es pequeño, cuando el grafo es denso, o cuando el algoritmo hace muchas consultas de adyacencia sueltas y ningún recorrido.

Las tres, lado a lado

Se puede cambiar de representación

Pasar de lista de aristas a lista de adyacencia cuesta $\Theta(V + E)$, una sola pasada. Si el algoritmo va a hacer más trabajo que eso, conviene construir la representación que le sirve en lugar de forzarlo sobre la que llegó.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

¿Por dónde se empieza?

Con las estructuras lineales no había pregunta

Una lista se recorre del primero al último. Una pila y una cola se vacían sacando de a uno. En los tres casos el orden viene dado y no hay nada que decidir.

Un grafo no trae orden

No hay primero ni último. Se puede empezar por cualquier vértice: el arranque cambia el orden de visita, no el conjunto de vértices que se alcanzan. Lo que sí hay que decidir es cómo seguir desde cada uno, porque sus vecinos son varios.

¿Por dónde se empieza?

Desde un solo vértice no se llega a todos

En G_1, desde el 0 no hay forma de tocar el 6. En G_2, desde el 0 no se llega al 6 porque no hay ninguna arista que entre a él. Recorrer el grafo *entero* necesita un ciclo que arranque de nuevo en cada vértice que quede sin visitar.

Camino y alcance

Definición (Camino)

Sea $G = (V, E)$ un grafo. Un ***camino*** *de u a v es una secuencia de vértices $u = x_0, x_1, \ldots, x_k = v$ tal que $(x_{i-1}, x_i) \in E$ para todo i con $1 \leq i \leq k$. El número k de aristas es su* ***longitud****.*

Definición (Alcanzable)

v es ***alcanzable*** *desde u si existe un camino de u a v. Se escribe $u \rightsquigarrow v$. Todo vértice es alcanzable desde sí mismo por el camino de longitud 0.*

Camino y alcance

Lo que hace un recorrido

Dado $s \in V$, un recorrido visita exactamente el conjunto $\{ v \in V : s \rightsquigarrow v \}$, cada vértice una sola vez. En un grafo no dirigido la relación es simétrica; en uno dirigido no, y por eso al 6 de G_2 no se llega desde el 0 aunque exista la arista $(6, 2)$.

Lo que hay que decidir, y lo que no

Tres decisiones

1. **Dónde empezar.** Cualquier vértice sirve.
2. **Qué hacer con los repetidos.** A un vértice se puede llegar por varios caminos, y hay ciclos. Sin una marca de visitado el recorrido no termina.
3. **Por cuál vecino seguir.** Aquí se separan los dos recorridos: uno se va tan lejos como pueda, el otro avanza por anillos.

La marca es lo que hace terminar

Cada vértice se marca la primera vez que se toca y no se vuelve a entrar en él. Como hay V vértices y cada uno se marca una sola vez, el recorrido no puede dar vueltas para siempre.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

La idea

Tan lejos como se pueda, y después devolverse

Desde el vértice actual se toma el primer vecino sin visitar y se sigue por él, sin mirar los demás. Cuando ya no queda ningún vecino sin visitar, se devuelve al vértice anterior y prueba con el siguiente de los suyos. Es exactamente el comportamiento de una pila de llamadas.

A mano sobre G_1

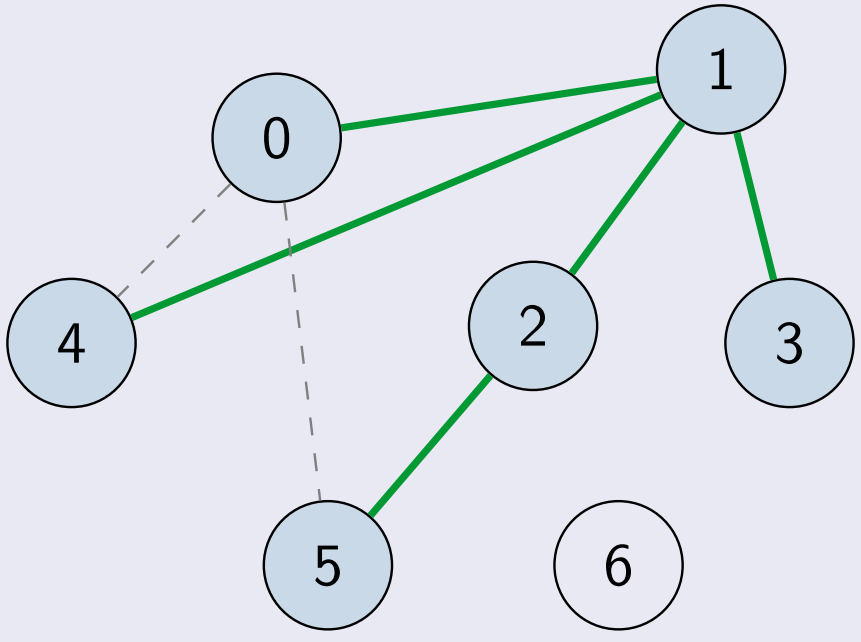
Paso	En	Vecinos	Qué hace
1	0	$1, 4, 5$	1 sin visitar: baja a 1
2	1	$0, 2, 3, 4$	0 visitado; 2 no: baja a 2
3	2	$1, 5$	1 visitado; 5 no: baja a 5
4	5	$0, 2$	los dos visitados: vuelve a 2
5	2	—	no le quedan: vuelve a 1
6	1	$3, 4$	3 sin visitar: baja a 3
7	3	1	visitado: vuelve a 1
8	1	4	4 sin visitar: baja a 4
9	4	$0, 1$	visitados: vuelve
10	—	—	la pila se vacía y termina

Orden de visita

$0, 1, 2, 5, 3, 4$. El 6 no aparece: no se llega a él desde el 0.

A mano sobre G_1

Las aristas que usó el recorrido



En verde, las cinco aristas por las que el recorrido bajó a un vértice nuevo: forman un árbol sobre los seis vértices alcanzados. Punteadas, las dos que llevaban a alguien ya visitado.

A mano sobre G_2, que es dirigido

Paso	En	Sale hacia	Qué hace
1	0	$1, 2$	1 sin visitar: baja a 1
2	1	3	sin visitar: baja a 3
3	3	5	sin visitar: baja a 5
4	5	—	no sale ninguna arista: vuelve
5	0	2	2 sin visitar: baja a 2
6	2	$1, 4$	1 visitado; 4 no: baja a 4
7	4	$3, 5$	los dos visitados: vuelve
8	—	—	termina

Orden de visita

$0, 1, 3, 5, 2, 4$. El 6 queda por fuera, y aquí por una razón distinta: existe la arista $(6, 2)$, pero no hay ninguna que entre al 6. En un grafo dirigido, alcanzar no es recíproco.

El algoritmo

$DFS(G)$

1. Para cada nodo v de G:
   1.1 Marcar v como no visitado
2. Para cada nodo v de G:
   2.1 Si v no ha sido visitado ejecutar $DFSAux(v)$

$DFSAux(v)$

1. Marcar v como visitado
2. Agregar v al orden de visita
3. Para cada nodo u adyacente a v:
   3.1 Si u no ha sido visitado ejecutar $DFSAux(u)$

El algoritmo

Los dos procedimientos

El paso 2 de *DFS* es el ciclo externo, el que no deja a nadie por fuera. *DFSAux* es el recorrido propiamente dicho, y es recursivo: la pila de llamadas es la que guarda el camino de vuelta.

La implementación, sobre lista de adyacencia

```
def dfs(G, u, visitado, orden):
    # Marca u, lo anota y sigue por el primer vecino que falte
    visitado[u] = True
    orden.append(u)
    for v in G[u]:
        if not visitado[v]:
            dfs(G, v, visitado, orden)

def dfs_desde(G, inicio):
    # Los vertices alcanzables desde inicio, en orden de visita
    visitado = [False] * len(G)
    orden = []
    dfs(G, inicio, visitado, orden)
    return orden
```

La implementación, sobre lista de adyacencia

La marca se pone al entrar

Si se pusiera al salir, un ciclo haría que el mismo vértice se visitara dos veces. El `for` recorre los vecinos en el orden en que estén guardados, y ese orden decide la traza.

El ciclo externo, para no dejar a nadie por fuera

```
def dfs_completo(G):
    # Al agotar lo alcanzable, arranca en el primer vertice
    # que siga sin visitar
    n = len(G)
    visitado = [False] * n
    orden = []
    u = 0
    while u < n:
        if not visitado[u]:
            dfs(G, u, visitado, orden)
        u = u + 1
    return orden
```

Sobre los dos grafos

`dfs_completo(G1)` da $0, 1, 2, 5, 3, 4, 6$ y `dfs_completo(G2)` da $0, 1, 3, 5, 2, 4, 6$: en ambos el 6 entra al final, cuando el ciclo externo llega a él y lo encuentra sin marcar.

El ciclo externo, para no dejar a nadie por fuera

Cuidado con `[[]] * n`

`[False] * n` está bien, porque los booleanos no se modifican. `[[]] * n` no: deja *n* referencias a la *misma* lista, y agregar un vecino a una las cambia todas.

La misma idea sin recursión

```
def dfs_con_pila(G, inicio):
    # La pila guarda lo que falta por mirar
    visitado = [False] * len(G)
    orden = []
    pila = [inicio]
    while len(pila) > 0:
        u = pila.pop()
        if not visitado[u]:
            visitado[u] = True
            orden.append(u)
            i = len(G[u]) - 1
            while i >= 0:
                if not visitado[G[u][i]]:
                    pila.append(G[u][i])
                i = i - 1
    return orden
```

La misma idea sin recursión

El orden de apilado y el momento de la marca

Los vecinos se apilan **al revés** para que el primero de la lista quede encima y salga primero: así esta versión produce el mismo orden que la recursiva. Y la marca se pone al **sacar**, no al apilar, porque un vértice puede entrar a la pila varias veces antes de que le toque el turno.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

La idea

Por anillos

Primero el vértice de arranque. Después todos sus vecinos. Después los vecinos de esos que no se hayan visto. El recorrido avanza por capas, y no entra a la capa siguiente hasta terminar la actual.

Una cola en lugar de una pila

La profundidad atiende lo último que llegó; la amplitud, lo primero. Ese cambio de estructura es la única diferencia entre los dos algoritmos.

Las distancias salen del recorrido

Al avanzar por capas, la capa en la que aparece un vértice es el número mínimo de aristas para llegar a él. La búsqueda en amplitud resuelve el camino más corto cuando todas las aristas valen lo mismo.

A mano sobre G_1

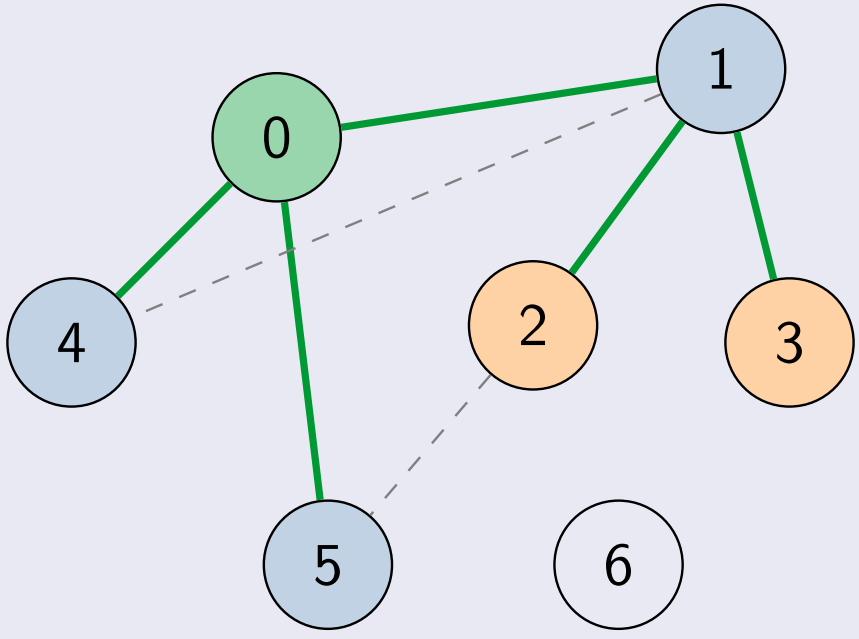
Paso	Sale	Vecinos nuevos	Cola después
1	0	$1, 4, 5$ a distancia 1	$1, 4, 5$
2	1	$2, 3$ a distancia 2	$4, 5, 2, 3$
3	4	—	$5, 2, 3$
4	5	—	$2, 3$
5	2	—	3
6	3	—	vacía: termina

Orden de visita y distancias

Orden: $0, 1, 4, 5, 2, 3$. Distancias desde el 0: $d(0) = 0$;
$d(1) = d(4) = d(5) = 1$; $d(2) = d(3) = 2$; y $d(6)$ no existe.

A mano sobre G_1

Las capas, dibujadas



Verde el arranque, azul la capa 1, naranja la capa 2. Las dos aristas punteadas unen vértices que ya estaban vistos y no aportan un camino más corto.

A mano sobre G_1

El mismo grafo, dos órdenes

Mismo arranque y dos recorridos distintos: $0, 1, 2, 5, 3, 4$ contra $0, 1, 4, 5, 2, 3$. También son distintos los árboles que quedan.

A mano sobre G_2

Paso	Sale	Vecinos nuevos	Cola después
1	0	$1, 2$ a distancia 1	$1, 2$
2	1	3 a distancia 2	$2, 3$
3	2	4 a distancia 2	$3, 4$
4	3	5 a distancia 3	$4, 5$
5	4	—	5
6	5	—	vacía: termina

Orden y distancias

Orden: $0, 1, 2, 3, 4, 5$. Distancias: $0, 1, 1, 2, 2, 3$, y el 6 sin distancia. Al 5 se llega por dos caminos, desde el 3 y desde el 4; se queda con el primero que lo alcanza, que es el más corto.

A mano sobre G_2

En profundidad, otro orden

$0, 1, 3, 5, 2, 4$: el 5 se visitó tercero, aunque esté a tres aristas del arranque. La profundidad no dice nada sobre distancias.

El algoritmo

$BFS(G, s)$

1. Para cada nodo v de G:
    1.1 Hacer $v.distancia = \infty$ y $v.padre = NIL$
2. Hacer $s.distancia = 0$
3. Agregar s a la cola P
4. Mientras la cola P no esté vacía:
    4.1 Retirar el frente w de la cola P
    4.2 Agregar w al orden de visita
    4.3 Para cada nodo u adyacente a w:
        4.3.1 Si $u.distancia = \infty$:
            4.3.1.1 Hacer $u.distancia = w.distancia + 1$
            4.3.1.2 Hacer $u.padre = w$
            4.3.1.3 Agregar u a la cola P

El algoritmo

La distancia hace de marca

$u.distancia = \infty$ dice que u no se ha visto, así que el paso 4.3.1 es a la vez la pregunta por la marca y la asignación de la capa. Y la marca se pone en 4.3.1.1, al encolar, no al retirar.

Lo que garantiza

Teorema (Correctitud de la búsqueda en amplitud, CLRS Teorema 22.5, p. 600)

Sea $G = (V, E)$ un grafo dirigido o no dirigido y sea $s \in V$. Al terminar $BFS(G, s)$, para todo $v \in V$ se tiene $v.distancia = \delta(s, v)$, donde $\delta(s, v)$ es la longitud del camino más corto de s a v, e ∞ si v no es alcanzable desde s.

De dónde sale

De que la cola nunca guarda vértices de más de dos capas consecutivas: si el frente está en la capa k, todo lo que queda está en la capa k o en la $k + 1$. Las capas salen entonces en orden, y un vértice recibe su distancia la primera vez que alguien lo toca, que es por el camino más corto.

Lo que garantiza

Vale porque todas las aristas cuestan lo mismo

Si las aristas llevaran pesos distintos, la primera vez que se toca un vértice ya no sería por el camino más barato, y este teorema sería falso. Ese caso necesita otro algoritmo.

La implementación, sobre lista de adyacencia

```
from collections import deque

def bfs(G, inicio):
    # distancia[v] == -1 significa que v no se ha visto
    n = len(G)
    distancia = [-1] * n
    padre = [-1] * n
    orden = []
    distancia[inicio] = 0
    cola = deque()
    cola.append(inicio)
    while len(cola) > 0:
        u = cola.popleft()
        orden.append(u)
        for v in G[u]:
            if distancia[v] == -1:
                distancia[v] = distancia[u] + 1
                padre[v] = u
                cola.append(v)
    return (orden, distancia, padre)
```

La implementación, sobre lista de adyacencia

Por qué no hace falta un arreglo de visitados

`distancia[v] == -1` ya distingue lo visto de lo no visto. Y la marca se pone **al encolar**, no al desencolar: si se pusiera al salir, un vértice con dos vecinos en la misma capa entraría dos veces a la cola.

El camino, no solo la distancia

```
def camino_hasta(padre, inicio, destino):
    # Sube por los padres desde destino y despues invierte
    camino = []
    v = destino
    while v != -1:
        camino.append(v)
        v = padre[v]
    camino.reverse()
    if len(camino) == 0 or camino[0] != inicio:
        camino = []
    return camino
```

Sobre G_1

Con `padre` calculado desde el 0, el camino hasta el 3 es $[0, 1, 3]$: dos aristas, que es justo $d(3)$. Hasta el 6 devuelve la lista vacía, porque nunca se le asignó padre.

El camino, no solo la distancia

Por qué el camino sale al revés

Cada vértice guarda de dónde se llegó a él, no hacia dónde va. Subir por los padres reconstruye el camino desde el final, y por eso hay que invertirlo.

Los dos, lado a lado

	Profundidad	**Amplitud**
Estructura	pila (o recursión)	cola
Avanza	tan lejos como pueda	por capas
Marca al	entrar al vértice	encolar el vértice
Da distancias	no	sí, en número de aristas
Orden en G_1	$0, 1, 2, 5, 3, 4$	$0, 1, 4, 5, 2, 3$
Memoria extra	la profundidad del camino	el ancho de una capa

Lo que comparten

Los dos visitan exactamente los vértices alcanzables desde el arranque, cada uno una sola vez, y los dos miran cada arista una vez por cada extremo. Por eso cuestan lo mismo.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

Sobre lista de adyacencia

Teorema

Recorrer un grafo $G = (V, E)$ en profundidad o en amplitud, guardado como lista de adyacencia, cuesta $\Theta(V + E)$.

Sobre lista de adyacencia

Demostración

Se procede contando por separado lo que aporta cada vértice y lo que aporta cada arista.

- Cada vértice se marca una sola vez, porque antes de entrar en él se pregunta por la marca. Entrar, marcar y anotar cuesta $\Theta(1)$, y hay V vértices: en total $\Theta(V)$.
- Estando en u se recorre su lista de vecinos, que tiene $\delta(u)$ entradas. Como cada vértice se visita una vez, esa lista se recorre una vez, y la suma sobre todos los vértices es $\sum_{u \in V} \delta(u) = 2|E|$ en el grafo no dirigido y $|E|$ en el dirigido: en los dos casos $\Theta(E)$.

Sumando las dos partes, el costo total es $\Theta(V + E)$.

Sobre lista de adyacencia

Conclusión

Por lo tanto, se puede concluir que ambos recorridos cuestan $\Theta(V + E)$ sobre lista de adyacencia. El V no se puede quitar: hay que pasar por todos los vértices aunque no tengan aristas, como el 6.

Sobre las otras dos representaciones

Matriz de adyacencia: $\Theta(V^2)$

Preguntar por los vecinos de u obliga a recorrer la fila entera, con sus V posiciones, de las cuales casi todas suelen ser ceros. Se hace una vez por vértice, así que el recorrido cuesta $\Theta(V^2)$ aunque el grafo tenga tres aristas.

Lista de aristas: $\Theta(V \cdot E)$

No hay forma de llegar a los vecinos de u sin recorrer las E aristas del grafo. Se hace una vez por vértice visitado, y el total es $\Theta(V \cdot E)$.

Con los números de antes

Con $V = 10^5$ y $E = 2 \cdot 10^5$: la lista de adyacencia hace unas $3 \cdot 10^5$ operaciones; la matriz, 10^{10}; la lista de aristas, $2 \cdot 10^{10}$. La primera corre en menos de un segundo y las otras dos no terminan.

Sobre las otras dos representaciones

Cuál conviene, entonces

La lista de adyacencia gana porque guarda juntos los vecinos de cada vértice, que es justo lo que los recorridos preguntan. La matriz gana cuando la pregunta es otra: ¿existe esta arista?

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

Los que más cuestan

Al construir el grafo

- Olvidar la arista de vuelta en un grafo no dirigido. El programa corre, no falla, y contesta mal: el grafo quedó dirigido.
- Mezclar la numeración del enunciado con la de Python. Se resta 1 al leer y nunca más.
- Crear las listas con `[[]] * n`, que deja n referencias a la misma lista.

Los que más cuestan

Al recorrer

- No marcar los visitados. Con un ciclo, el recorrido no termina.
- Marcar en el momento equivocado: en profundidad se marca al entrar; en amplitud, al encolar. Marcar al desencolar mete vértices repetidos en la cola.
- Quedarse con un solo arranque cuando el problema pide recorrer todo el grafo. Falta el ciclo externo.
- Recorrer sobre matriz de adyacencia por costumbre, y pasar de $\Theta(V + E)$ a $\Theta(V^2)$ sin darse cuenta.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

Los mismos recorridos, sobre otra representación

Tres conversiones

Las implementaciones de esta clase suponen que el grafo llegó como lista de adyacencia. Reescriba *DFSAux* y *BFS* —primero el pseudocódigo, después el código— para cada una de estas tres formas, sin convertir antes a lista de adyacencia:

1. **Matriz de adyacencia.** La entrada es `m`, una lista de n listas de n ceros y unos. ¿Qué reemplaza a `for v in G[u]`?
2. **Lista de aristas.** La entrada es una lista de pares (u, v) y el número de vértices n.
3. **Diccionario de listas.** Los vértices no son números sino nombres, y el grafo es un `dict` que lleva cada nombre a la lista de sus vecinos. ¿Con qué se reemplaza el arreglo `visitado`?

Los mismos recorridos, sobre otra representación

El análisis de cada una

Para las tres, diga cuánto cuesta el recorrido completo en términos de V y E, y sobre qué línea del código cae ese costo. Compare con el $\Theta(V + E)$ de la lista de adyacencia y explique la diferencia con la operación que cada representación hace cara.

Para practicar en papel

Sobre los dos grafos de la clase

1. Recorra G_1 en profundidad y en amplitud arrancando en el 2 en lugar del 0. ¿Cambian los órdenes de visita? ¿Cambian las distancias?
2. En G_1, ¿desde cuáles vértices se alcanza a todos los demás? ¿Y en G_2?
3. Invierta todas las aristas de G_2 y recorra en profundidad desde el 5. ¿Qué vértices alcanza ahora?
4. Guarde los vecinos de cada vértice de G_1 en orden *decreciente* y vuelva a recorrer en profundidad desde el 0. El orden de visita cambia; el conjunto de vértices alcanzados, no. Explique por qué.
5. Dibuje un grafo de cinco vértices donde la profundidad y la amplitud den el mismo orden de visita desde el mismo arranque.

Para enviar al juez

UVa 11902 — Dominator

Enunciado:
`https://vjudge.net/problem/UVA-11902`
Envío:
`https://vjudge.net/problem/UVA-11902`

Cómo atacarlo

La entrada llega como matriz de adyacencia de un grafo dirigido. Se recorre desde el 0 una vez para saber a quién se alcanza, y después una vez por cada vértice v, borrándolo y volviendo a recorrer: los que dejaron de alcanzarse están dominados por v. Con $n \leq 100$ los n recorridos sobre matriz caben de sobra.

Lo que sigue

Lo que ya se puede contestar

Si dos vértices están conectados, cuántas piezas separadas tiene el grafo, y cuál es el camino más corto cuando todas las aristas valen lo mismo.

Para investigar

Busque qué es un **grafo implícito**: un grafo cuyos vértices y aristas no están guardados en ninguna lista, sino que se calculan a medida que el recorrido los pide. El tablero de un juego y el laberinto de una cuadrícula son los ejemplos de siempre.

Contenido

1 Recuento

2 Las cuatro representaciones

3 Cuál conviene

4 Recorrer un grafo

5 Búsqueda en profundidad

6 Búsqueda en amplitud

7 El costo

8 Errores comunes

9 Ejercicios

10 Referencias

Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*, 3rd ed. MIT Press, 2009. Sección 22.1 (pp. 589–593), Sección 22.2 (pp. 594–602) y Sección 22.3 (pp. 603–612).
- Rosen, K. H. *Discrete Mathematics and Its Applications*, 7th ed. McGraw-Hill, 2012. Capítulo 10.
- van Steen, M. *Graph Theory and Complex Networks: An Introduction*. 2010.
- Halim, S., Halim, F., Effendy, S. *Competitive Programming 4*. Lulu, 2020. Secciones 2.4.1 y 4.2.