Algoritmos y Estructuras de Datos

Grafos

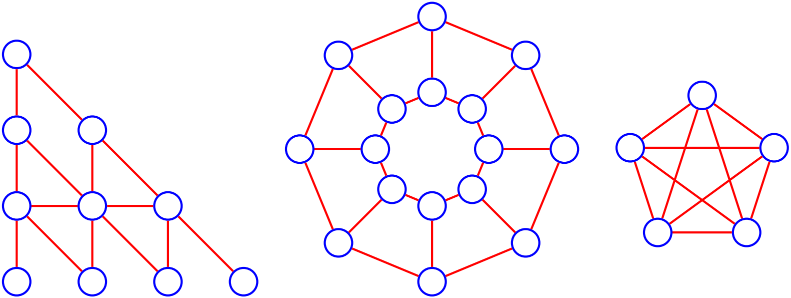
Grafos

Algoritmos sobre grafos

Caso de estudio: grafos de dependencias

Grafos

Un **grafo** es una "red" de vértices conectados por aristas:



- Los vértices también se llaman *nodos*.
- Las aristas también se llaman *ejes* o *arcos*.

Grafos

Sirven para modelar muchos objetos, situaciones y problemas:

- Lugares y rutas (redes de transporte).
- Servidores y conexiones (redes informáticas).
- Personas y relaciones (redes sociales).
- Conceptos y relaciones (redes de conocimiento).
- ...

Diversos tipos de grafos

- **No dirigidos:**

$$\begin{tikzcd}
a \arrow[r, dash] \arrow[d, dash] & b \arrow[r, dash] \arrow[d, dash] & c \arrow[r, dash] \arrow[dl, dash] & d \\
e \arrow[r, dash] \arrow[ur, dash] & f
\end{tikzcd}$$

- **Dirigidos:**

$$\begin{tikzcd}
a \arrow[r] & b \arrow[r, bend right] \arrow[d] & c \arrow[l, bend right] \arrow[dl] & d \arrow[l] \\
e \arrow[u] \arrow[ur] \arrow[r] & f
\end{tikzcd}$$

- **Mixtos:**

$$\begin{tikzcd}
a \arrow[r] & b \arrow[r, bend right] \arrow[d] & c \arrow[l, bend right] \arrow[r, dash] & d \\
e \arrow[u] \arrow[ur, dash] \arrow[r, dash] & f \arrow[ur, dash]
\end{tikzcd}$$

- **Multi-grafos:**

$$\begin{tikzcd}
a \arrow[r, dash, bend left] \arrow[r, dash, bend right] \arrow[d, dash] & b \arrow[r, dash] \arrow[d, dash] & c \arrow[r, dash] \arrow[dl, dash] \arrow[dl, dash, bend right] \arrow[dl, dash, bend left] & d \\
e \arrow[r, dash] \arrow[ur, dash] & f
\end{tikzcd}$$

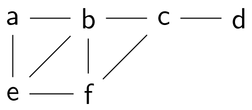
Grafo — Definición formal

Definición

Un **grafo** (no dirigido) es un par (V, E), donde:

- V es un conjunto finito de vértices,
- E es un conjunto finito de aristas.
  Cada arista es un par no ordenado de vértices distintos $\{v, w\} \subseteq V$.

Ejemplo



$G = (V, E)$
$V = \{a, b, c, d, e, f\}$
$E = \{\{a, b\}, \{a, e\}, \{b, c\}, \{b, e\}, \{b, f\}, \{c, d\}, \{c, f\}, \{e, f\}\}$

Nociones básicas

Sea $G = (V, E)$ un grafo.

- Generalmente notamos:
  - n al número total de vértices de G.
  - m al número total de aristas de G.
- Si $\{v, w\} \in E$, decimos que v, w son *adyacentes* o *vecinos*.
- Notamos $d(v)$ al número de vecinos de $v \in V$.

Observación

Un grafo de n vértices tiene a lo sumo $\frac{n^2-n}{2}$ aristas.
(Es decir, $m \in O(n^2)$).

Demostración.

- Sea $V = \{v_0, \ldots, v_{n-1}\}$.
- Observemos que $d(v_i) \leq n - 1$ para todo $i \in \{0, \ldots, n-1\}$.
- Por lo tanto el número total de aristas es:

$$\frac{1}{2} \cdot \sum_{i=0}^{n-1} d(v_i) \quad \leq \quad \frac{1}{2} \cdot \sum_{i=0}^{n-1} n - 1 \quad = \quad \frac{1}{2}\, n\,(n-1)$$

Nociones básicas

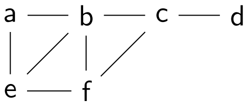
Sea $G = (V, E)$ un grafo.

- Si $v, w \in V$ son vértices, un *camino* de v a w es una secuencia de ℓ aristas que conectan v con w pasando por $\ell + 1$ nodos en total:

$$v = v_0 - v_1 - v_2 \ldots - v_\ell = w$$

  $\ell \geq 0$ se llama la *longitud* del camino.
- Decimos que G es *conexo* si para todo par de vértices $v, w \in V$ hay un camino entre ellos.
- Un *ciclo* es un camino de longitud $\ell > 0$ que va de un vértice v a sí mismo.

Ejemplo



Es conexo y tiene un ciclo a — b — e — a.

Grafos

Algoritmos sobre grafos

Caso de estudio: grafos de dependencias

Representaciones de grafos

Para representar grafos es necesario elegir una estructura de datos.
No hay una manera única de representarlos.
Cada representación tiene sus ventajas y desventajas.

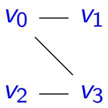
Representaciones de grafos

Sean $G = (V, E)$, $V = \{v_0, \ldots, v_{n-1}\}$, $E = \{e_0, \ldots, e_{m-1}\}$.

Matriz de adyacencia

El grafo se representa con una matriz $\mathcal{A}$ de $n \times n$ booleanos, de tal modo que $\{v_i, v_j\} \in E$ si y sólo si $\mathcal{A}[i][j] = \texttt{True}$.

$G =$



$\mathcal{A} =$

	v_0	v_1	v_2	v_3
v_0	○	●	○	●
v_1	●	○	○	○
v_2	○	○	○	●
v_3	●	○	●	○

Observación: alcanza con guardar la mitad triangular superior.

- Espacio de almacenamiento . . . . $O(n^2)$
- Complejidad de determinar si v, w son vecinos . . . . $O(1)$
- Complejidad de calcular $d(v)$ . . . . $O(n)$

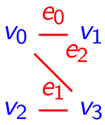
Representaciones de grafos

Sean $G = (V, E)$, $V = \{v_0, \ldots, v_{n-1}\}$, $E = \{e_0, \ldots, e_{m-1}\}$.

Matriz de incidencia

El grafo se representa con una matriz $\mathcal{I}$ de $n \times m$ booleanos, de tal modo que $v_i \in e_j$ si y sólo si $\mathcal{I}[i][j] = \texttt{True}$.

$G =$



$\mathcal{I} =$

	e_0	e_1	e_2
v_0	●	○	●
v_1	●	○	○
v_2	○	●	○
v_3	○	●	●

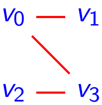
- Espacio de almacenamiento .... $O(n\, m)$
- Complejidad de determinar si v, w son vecinos .... $O(n\, m)$
- Complejidad de calcular $d(v)$ .... $O(n + m)$

Representaciones de grafos

Sean $G = (V, E)$, $V = \{v_0, \ldots, v_{n-1}\}$, $E = \{e_0, \ldots, e_{m-1}\}$.

Lista de adyacencias

El grafo se representa con un arreglo $\mathcal{A}$ de n conjuntos, de tal modo que $\mathcal{A}[v_i]$ es el conjunto de los vecinos de v_i.

$G =$ 

$$\mathcal{A} = \begin{array}{c|l} v_0 & \{v_1, v_3\} \\ v_1 & \{v_0\} \\ v_2 & \{v_3\} \\ v_3 & \{v_0, v_2\} \end{array}$$

- Espacio de almacenamiento .... $O(n^2)$
- Complejidad de determinar si v, w son vecinos ...... $O(\log n)$
  (depende de la representación del conjunto)
- Complejidad de calcular $d(v)$ .... $O(1)$

Problema de alcanzabilidad

Queremos un algoritmo con el siguiente contrato:

- Entrada: un grafo $G = (V, E)$ y dos vértices $v, w \in V$.
- Salida: un booleano que indique si hay un camino de v a w.

Depth-first search

Un método para resolver el problema de alcanzabilidad.

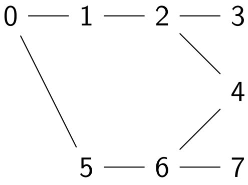
Depth-first search (DFS)

La entrada es $G = (V, E)$ y $v, w \in V$.

- Inicializar las estructuras auxiliares:
  - Un conjunto de vértices **ya visitados**, $W := \{\}$.
  - Una pila de vértices **pendientes por visitar**, $P := [v]$.
- Mientras haya vértices pendientes por visitar en la pila P:
  - Sacar el vértice x del tope de la pila de pendientes.
  - Si $x = w$, terminar: **hay un camino de v a w**.
  - Marcar x como visitado.
  - Para cada vecino y de x que no haya sido visitado:
    - Agregarlo como pendiente en la pila.
- Llegado este punto, **no hay un camino de v a w.**

Depth-first search — ejemplo

Veamos si hay un camino de 0 a 6:



P	W
$[0]$	$\emptyset$
$[1,5]$	$\{0\}$
$[2,5]$	$\{0,1\}$
$[3,4,5]$	$\{0,1,2\}$
$[4,5]$	$\{0,1,2,3\}$
$[6,5]$	$\{0,1,2,3,4\}$

Depth-first search — complejidad temporal

Asumimos que el grafo se representa sobre listas de adyacencia.
El conjunto W se puede representar con un arreglo de tamaño n.

- Mientras haya vértices pendientes por visitar en la pila P: $O(n)$ iteraciones
  - Sacar el vértice x del tope de la pila de pendientes. $O(1)$
  - Si $x = w$, terminar: hay un camino de v a w. $O(1)$
  - Marcar x como visitado. $O(1)$
  - Para cada vecino y de x que no haya sido visitado: $O(m)$ iteraciones en total
    - Agregarlo como pendiente en la pila. $O(1)$
- Llegado este punto, no hay un camino de v a w. $O(1)$

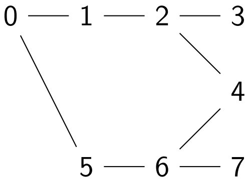
Total: $O(n + m)$.

Depth-first search en Python

Representación del grafo como una lista de adyacencias:

```
class Grafo:
    def __init__(self, vertices, aristas):
        self.vertices = vertices
        self.vecinos = [[] for v in vertices]
        for (v1, v2) in aristas:
            self.vecinos[v1].append(v2)
            self.vecinos[v2].append(v1)
```

Por ejemplo:




```
g = Grafo([0, 1, 2, 3, 4, 5, 6, 7],
          [[0, 1], [1, 2], [2, 3], [2, 4],
           [0, 5], [5, 6], [6, 4], [6, 7]])
```

Depth-first search en Python

```
def es_alcanzable_DFS(grafo, v, w):
    visitados = [False for x in grafo.vertices]
    pila = [v]
    while len(pila) > 0:
        x = pila.pop()
        if x == w:
            return True
        visitados[x] = True
        for y in grafo.vecinos[x]:
            if not visitados[y]:
                pila.append(y)
    return False
```

Breadth-first search

Otro método (emparentado) para el problema de alcanzabilidad.

Breadth-first search (BFS)

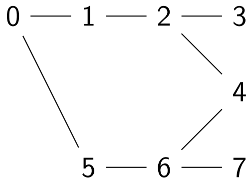
La entrada es $G = (V, E)$ y $v, w \in V$.

- Inicializar las estructuras auxiliares:
  - Un conjunto de vértices **ya visitados**, $W := \emptyset$.
  - Una cola de vértices **pendientes por visitar**, $Q := [v]$.
- Mientras haya vértices pendientes por visitar en la cola Q:
  - Sacar el próximo vértice x de la cola de pendientes.
  - Si $x = w$, terminar: hay un camino de v a w.
  - Marcar x como visitado.
  - Para cada vecino y de x que no haya sido visitado:
    - Agregarlo como pendiente en la cola.
- Llegado este punto, no hay un camino de v a w.

La complejidad también es $O(n + m)$.

Breadth-first search — ejemplo

Veamos si hay un camino de 0 a 7:



Q	W
$[0]$	$\emptyset$
$[1,5]$	$\{0\}$
$[5,2]$	$\{1\}$
$[2,6]$	$\{1,5\}$
$[6,3,4]$	$\{1,5,2\}$
$[3,4,7]$	$\{1,5,2,6\}$
$[4,7]$	$\{1,5,2,6,3\}$
$[7]$	$\{1,5,2,6,3,4\}$

Breadth-first search en Python

```
import queue

def es_alcanzable_BFS(grafo, v, w):
    visitados = [False for x in grafo.vertices]
    cola = queue.Queue()
    cola.put(v)
    while not cola.empty():
        x = cola.get()
        if x == w:
            return True
        visitados[x] = True
        for y in grafo.vecinos[x]:
            if not visitados[y]:
                cola.put(y)
    return False
```

Grafos

Algoritmos sobre grafos

Caso de estudio: grafos de dependencias

Contexto

Un procedimiento tiene varios **pasos**.

Cada paso tiene prerrequisitos (o **dependencias**).

Las dependencias pueden tener otros pasos como dependencia, y así sucesivamente.

Contamos con un diccionario que le asigna a cada paso una lista de sus dependencias directas.

Por ejemplo:

```
dependencias = {
  'MezclarBollo': [],
  'LevarBollo': ['MezclarBollo'],
  'PrepararSalsa': [],
  'ArmarPizza': ['LevarBollo', 'PrepararSalsa'],
  'PrecalentarHorno': [],
  'HornearPizza': ['ArmarPizza', 'PrecalentarHorno'],
  'ServirPizza': ['HornearPizza']
}
```

Problemas

1. Un ciclo en las dependencias está dado por un paso que depende (directa o indirectamente) de sí mismo.
   En este contexto, un ciclo en las dependencias generalmente representa un error.
   Diseñar un algoritmo para determinar si hay algún ciclo en las dependencias.
2. Diseñar un algoritmo que dado un paso indique la lista de todas las dependencias, ya sean directas o indirectas.
3. Suponiendo que cada paso demora una unidad de tiempo, pero que el trabajo de hacer un paso se puede paralelizar, diseñar un algoritmo que dado un paso indique cuántas unidades de tiempo en total se requieren como mínimo para llevarlo a cabo.