Algoritmos sobre grafos: BFS, DFS.

Tecnología Digital V: Diseño de Algoritmos

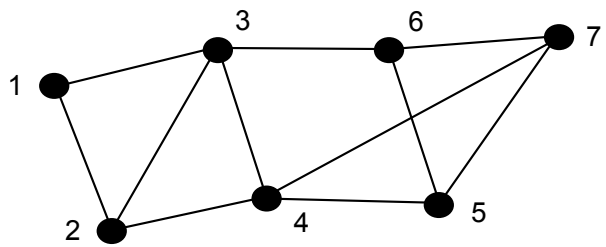
Universidad Torcuato Di Tella

UNIVERSIDAD
TORCUATO DI TELLA

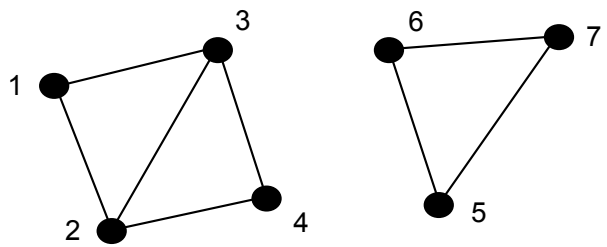
Motivación

Definiciones

- Un grafo es conexo si existe un camino entre todo par de vértices.
- Una componente conexa es un subconjunto de vértices conexo, maximal con esta propiedad.
- Un vértice aislado es un vértice i con $d(i) = 0$ (que conforma una componente conexa de tamaño 1).



Conexo



No conexo

Preguntas

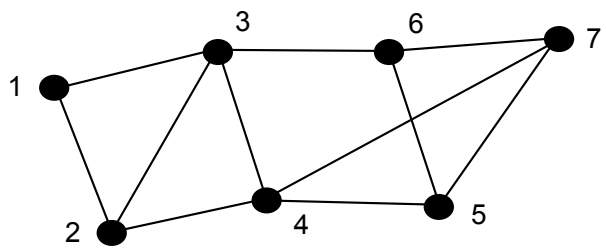
- Ideas para diseñar un algoritmo que verifique si un grafo es conexo?
- Podremos adaptarlo para obtener las componentes conexas de un grafo?

Recorrido de un grafo

Idea intuitiva

Dado un grafo $G = (V, E)$

1. partimos de un vértice arbitrario $s \in V$;
2. idenficamos todos los vecinos que se puede alcanzar desde s;
3. si G es conexo existe un camino entre s y todos los demás vértices;
4. verificamos si los vértices alcanzables son todos los vértices (i.e., V).



Recorrer los nodos iterativamente, chequeando sus vecinos.

Supongamos que iniciamos con $s = 1$

- Cómo podemos encontrar **todos** los vértices alcanzables desde 1 de forma sistemática?
- Qué situaciones particulares se pueden dar durante el proceso?

Podemos "descubrir" nodos más de una vez. Hay nodos aislados.

Recorrido de un grafo

```
RECORRIDO(G = (V, E), s ∈ V)
estado = vector(|V|)
for v ∈ V do
    estado[v] = no descubierto
end for
estado[s] = descubierto
L = [s]                          ▷ L tiene los vertices a procesar
while L ≠ ∅ do
    Sea v el siguiente elemento en L
    for w ∈ N(v) do
        if estado[w] = no descubierto then
            estado[w] = descubierto
            Agregar w a L
        end if
    end for
    estado[v] = procesado
    Eliminar a v de L
end while
return estado
```

En cada iteración, un vértice puede tener 1 de 3 posibles estados:

- `no descubierto`: todavía no ha sido visitado por el algoritmo. No llego
- `descubierto`: visitado por ser vecino de otro vértice, pero no procesado. Llego
- `procesado`: sus vecinos han sido descubiertos. Me paro

Al terminar el algoritmo, verificamos que estado[v] == `procesado` $\forall v \in V$.

Con este algoritmo, se verifica que el grafo es conexo.

Representación alterna de grafos: matriz de adyacencia, lista de vecinos

Algunas propiedades

- En el ciclo **while**, L contiene solamente vértices $v \in V$ tales que estado$[v]$ = `descubierto`.
- Al finalizar el algoritmo, estado$[v] \in \{$`no descubierto`, `procesado`$\}$.
- Los vértices $v \in V$ tales que estado$[v]$ = `procesado` conforman una componente conexa.

Preguntas

- Cuántas veces se ejecuta el ciclo **while** exterior en peor caso?
- Cuál es la complejidad de obtener los vecinos de v, $N(v)$?

Recorrido de un grafo: complejidad computacional

Sea $|V| = n$ y $|E| = m$.

RECORRIDO($G = (V, E)$, $s \in V$)

```
estado = vector(|V|)
for v ∈ V do                                   ▷ Incialización.
    estado[v] = no descubierto
end for
estado[s] = descubierto
L = [s]
while L ≠ ∅ do                                 ▷ Vértices descubiertos a procesar.
    Sea v el siguiente elemento en L           ▷ Procesamos el siguiente vértice.
    for w ∈ N(v) do                            ▷ Descubrimos los vecinos de v.
        if estado[w] = no descubierto then
            estado[w] = descubierto
            Agregar w a L
        end if
    end for
    estado[v] = procesado                      ▷ Marcamos v como procesado.
    Eliminar a v de L
end while
```

Recorrido de un grafo: complejidad computacional

Sea $|V| = n$ y $|E| = m$. G implementado sobre matriz de adyacencia.

RECORRIDO($G = (V, E)$, $s \in V$)

```
estado = vector(|V|)
for v ∈ V do                                          ▷ O(n).
    estado[v] = no descubierto
end for
estado[s] = descubierto
L = [s]
while L ≠ ∅ do                                        ▷ O(n).
    Sea v el siguiente elemento en L
    for w ∈ N(v) do                                   ▷ O(n).
        if estado[w] = no descubierto then
            estado[w] = descubierto                   ▷ O(1).
            Agregar w a L                             ▷ O(1).
        end if
    end for
    estado[v] = procesado                             ▷ O(1).
    Eliminar a v de L
end while
```

Recorrido de un grafo: complejidad computacional

Sea $|V| = n$ y $|E| = m$. G implementado sobre lista de vecinos.

RECORRIDO($G = (V, E)$, $s \in V$)

---

estado = vector($|V|$)
for $v \in V$ **do** ▷ $O(n)$.
    estado[v] = `no descubierto`
end for
estado[s] = `descubierto`
$L = [s]$
while $L \neq \emptyset$ **do** ▷ $O(n)$.
    Sea v el siguiente elemento en L
    for $w \in N(v)$ **do** ▷ Obtener $N(v)$ es $O(1)$. Recorrerla, $O(d(v))$.
        if estado[w] = `no descubierto` **then**
            estado[w] = `descubierto` ▷ $O(1)$.
            Agregar w a L ▷ $O(1)$.
        end if
    end for
    estado[v] = `procesado` ▷ $O(1)$.
    Eliminar a v de L
end while

Recorrido de un grafo: complejidad computacional

Matriz de adyacencia

- Obtener los vecinos: $O(n)$
- Recorrer la lista de vecinos de v: $O(d(v))$
- Descubrir / procesar todos los vértices: $O(n^2)$
- Algoritmo completo: $O(n) + O(n^2) = O(n^2)$

Lista de vecinos

- Obtener los vecinos: $O(1)$
- Recorrer la lista de vecinos de v: $O(d(v))$
- Descubrir / procesar todos los vértices: $\sum_{v \in V} d(v)$
- Algoritmo completo: $O(n) + O(\sum_{v \in V} d(v)) = ?$
  n < m => O(m)

Handshaking Lemma

Sea $G = (V, E)$ un grafo con $m = |E|$ la cantidad de aristas. Entonces,

$$\sum_{v \in V} d(v) = 2m$$

Recorrido de un grafo: complejidad computacional

Resumiendo:

1. $O(n^2)$ si el grafo está implementado sobre una matriz de adyacencia.
2. $O(m)$ si el grafo está implementado sobre listas de vecinos.

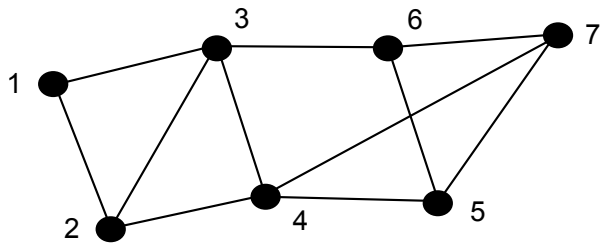
Es habitual escribir la complejidad de algoritmos sobre grafos en función de $n = |V|$ y $m = |E|$.

1. En general, $m = O(n^2)$, con lo cual el peor caso es el mismo para ambas implementaciones.
2. Si $m = O(n)$, decimos que el grafo es poco denso y entonces conviene la segunda implementación.
   Ejemplo: sabemos que cada vértices está conectado con a lo sumo k vecinos, con k acotado.

Ejercicio (6, guía 2)

Cómo re-usamos Recorrido(G, s) para obtener todas las componentes conexas de un grafo?

Tipos de recorridos



- BFS(G,v): Si la lista L se implementa con una cola (queue), entonces el algoritmo recorre el grafo **a lo ancho** y se lo llama (*breadth-first search*).
- DFS(G,v): Si la lista L se implementa con una pila (stack), entonces el algoritmo recorre el grafo **en profundidad** y se lo llama (*depth-first search*).

Pregunta

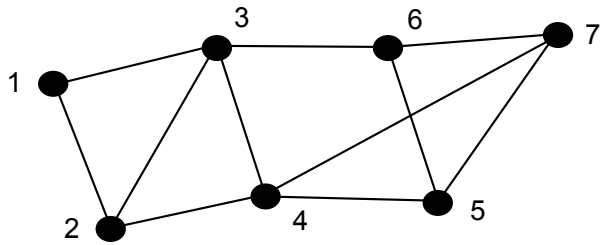
Si quisiéramos calcular la distancia de un vértice v a otro w, que esquema nos convendría usar?

Recorrido de un grafo: complejidad computacional

BFS($G = (V, E)$, $s \in V$)

estado = vector($|V|$), dist = vector($|V|$)
for $v \in V$ **do**
    estado[v] = `no descubierto`
    dist[v] = ∞
end for
estado[s] = `descubierto`, dist[s] = 0
L = Queue(), Agregar s a L.
while $L \neq \emptyset$ **do**
    Sea v el siguiente elemento en L
    for $w \in N(v)$ **do**
        if estado[w] = `no descubierto` **then**
            estado[w] = `descubierto`
            dist[w] = dist[v] + 1
            Agregar w a L
        end if
    end for
    estado[v] = `procesado`
    Eliminar a v de L
end while
return estado, dist

Tipos de recorridos



- BFS(G,v): Si la lista L se implementa con una **cola** (queue), entonces el algoritmo recorre el grafo **a lo ancho** y se lo llama (*breadth-first search*). **Recorre los vértices en orden de distancia creciente desde s.**

- DFS(G,v): Si la lista L se implementa con una **pila** (stack), entonces el algoritmo recorre el grafo **en profundidad** y se lo llama (*depth-first search*). Entre recorrer izquierda o derecha primero es lo mismo **Encuentra primero el vértice más lejano a s en cada camino antes de recorrer otros caminos.**