IIC-2133 — Estructuras de Datos y Algoritmos
Grafos

Jorge A. Baier

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

Santiago, Chile

Representando Grafos en Memoria

Dado un grafo $G = (V, E)$ estas son posibles representaciones:

1. *Listas de adyacencia*. Para cada $u \in V$, $Adj[u]$ es una lista con todos los v tales que $(u, v) \in E$.

Representando Grafos en Memoria

Dado un grafo $G = (V, E)$ estas son posibles representaciones:

1. *Listas de adyacencia*. Para cada $u \in V$, $Adj[u]$ es una lista con todos los v tales que $(u, v) \in E$.

2. *Matriz de adyacencia*. Definimos una matriz A, tal que

$$a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{en caso contrario} \end{cases}$$

¿Cuánta memoria se require usando estas representaciones?

Recorriendo el Grafo

Definiremos un algoritmo (BFS) para recorrer un grafo a partir de un nodo s.

- $color[u]$ puede ser alguno de estos:
  - $blanco$ si no hemos visto nunca u.
  - $gris$ hemos encontrado un camino hasta u.
  - $negro$ hemos terminado de generar los nodos adyacentes a u.
- $\pi[u]$ el predecesor de u.
- $d[u]$ número de aristas en el camino descubierto hasta u
- Q estructura de datos que contiene a los nodos grises.

Búsqueda en Amplitud (Breadth-First Search)

```
function BFS(G, s)
    for each u ∈ V[G] do
        color[u] ← blanco; d[u] ← ∞; π[u] ← nil
    Inserte s a Q
    color[s] ← gris; d[s] ← 0
    while Q no está vacía do
        Extraiga un elemento u desde Q
        for each v ∈ Adj[u] do
            if color[v] = blanco then
                π[v] ← u
                d[v] ← d[u] + 1
                Inserte v a Q
                color[v] ← gris
        color[u] ← negro
```

Propiedades

Definicion: Decimos que v es alcanzable desde u, o, simplemente, $u \rightsquigarrow v$, si existe una camino que comienza en u y termina en v.

Teorema: $s \rightsquigarrow t$ ssi después de una llamada a BFS(G, s) se cumple que $color[t] = negro$.

Propiedades

Definicion: Decimos que v es alcanzable desde u, o, simplemente, $u \rightsquigarrow v$, si existe una camino que comienza en u y termina en v.

Teorema: $s \rightsquigarrow t$ ssi después de una llamada a BFS(G, s) se cumple que $color[t] = negro$.

Teorema: El tiempo de ejecución de BFS(G, s) es $O(|E|)$.

Más Propiedades de BFS

Theorem

Dado un grafo $G = (V, E)$ y una función de peso unitaria ($w(e) = 1$, para todo $e \in E$), luego de una llamada a BFS(G, s), $d[t] = \delta(s, t)$ para todo $t \in V$.

Definimos V_k como el conjunto de vértices a distancia k desde u.
Luego se procede por inducción en k.
Antes demostramos:

Lemma

Si durante una ejecución de BFS(G, s) la cola contiene los elementos $\langle v_1, v_2, \ldots, v_n \rangle$, entonces $d[v_n] \leq d[v_1] + 1$ y $d[v_i] \leq d[v_{i+1}]$ para cualquier $i \in \{1, \ldots, n - 1\}$.

Búsqueda en Profundidad (Depth-First Search)

```
procedure Init-DFS(G)
    t ← 0
    for each u ∈ V[G] do
        color[u] ← blanco; d[u] ← ∞; π[u] ← nil
procedure DFS(G, s)
    Init-DFS(G)
    DFS-visit(G, s)
```

DFS (parte 2)

procedure *DFS-visit*(G, s)
  $color[s] \leftarrow gris$
  $t \leftarrow t + 1$
  $d[s] \leftarrow t$
  for each $u \in Adj[s]$ **do**
    if $color[u] = blanco$ **then**
      $\pi[u] \leftarrow s$
      DFS-visit(G, u)
  $color[s] \leftarrow negro$
  $t \leftarrow t + 1$
  $f[s] \leftarrow t$

Observación: podemos interpretar a $d[s]$ y $f[s]$ como los tiempos de "inicio" y "finalización" de s.

El Bosque DFS

Cuando ejecutarmos DFS completamente sobre un grafo, generamos lo que se conoce como un *bosque DFS*.

```
procedure DFS(G)
    Init-DFS(G)
    for each s ∈ V[G] do
        if color[s] = blanco then
            DFS-visit(G, s)
```

¿Qué sucede cuando?

Si u, v son dos vértices en $G = (V, E)$, ¿es posible que:

1. $[d[u], f[u]]$ y $[d[v], f[v]]$ sean intervalos disjuntos?
2. $[d[u], f[u]]$ esté contenido en $[d[v], f[v]]$ o $[d[v], f[v]]$ esté contenido en $[d[u], f[u]]$?
3. $[d[u], f[u]]$ y $[d[v], f[v]]$ tengan intersección no vacía y no están contenidos el uno en el otro?

¿Qué sucede cuando?

Si u, v son dos vértices en $G = (V, E)$, ¿es posible que:

1. $[d[u], f[u]]$ y $[d[v], f[v]]$ sean intervalos disjuntos?
2. $[d[u], f[u]]$ esté contenido en $[d[v], f[v]]$ o $[d[v], f[v]]$ esté contenido en $[d[u], f[u]]$?
3. $[d[u], f[u]]$ y $[d[v], f[v]]$ tengan intersección no vacía y no están contenidos el uno en el otro?

Teorema del paréntesis: Sólo 1) y 2) se pueden dar.

White Path Theorem

Teorema: En un bosque DFS para un grafo G, v es descendiente de u ssi en tiempo $d[u]$, v es alcanzable desde u (mediante aristas de G) mediante un camino de nodos blancos.

Tree/Back Edges

Después de ejecutar DFS(G)

1. Decimos que (u, v) es un *tree edge* si v fue descubierto por primera vez desde u.
2. Decimos que (u, v) es un *back edge* si v es un ancestro de u en un árbol depth-first.

Propiedad: (u, v) es un *back edge* si v es gris cuando u es expandido. (Es decir, DFS se puede modificar para encontrar back-edges eficientemente)

Teorema: G tiene un ciclo ssi DFS(G) descubre un *back edge*

Ejercicios

Diga cómo usar o modificar DFS para:

1. Decidir si un grafo G tiene un ciclo.
2. Generar un orden topológico de $G = (V, E)$. (Definición: $<\subseteq V \times V$ es un orden topológico ssi $<$ es un orden total y para todo $(u, v) \in E$ se tiene $u < v$.)

Componentes Fuertemente Conexas

Definicion: Una componente fuertemente conexa (CFC) de un grafo $G = (V, E)$ es un subconjunto C maximal de V tal que para todo $u, v \in C$ $u \rightsquigarrow v$ (y $v \rightsquigarrow u$).

Componentes Fuertemente Conexas

Definicion: Una componente fuertemente conexa (CFC) de un grafo $G = (V, E)$ es un subconjunto C maximal de V tal que para todo $u, v \in C$ $u \leadsto v$ (y $v \leadsto u$).

Algoritmo de Kosarju (para calcular todas las CFC)

1. Llame a DFS(G) para computar los tiempos de término $f[u]$ para cada vértice u.
2. Compute G^T.
3. Llame a DFS(G^T), ordenando los nodos decrecientemente según f en el loop principal (línea 3, del pseudocódigo de DFS(G)).
4. Imprimir cada árbol DFS encontrado como un CFC.

Correctitud del Alg. de Kosarju

Lo discutimos ampliamente en la pizarra...

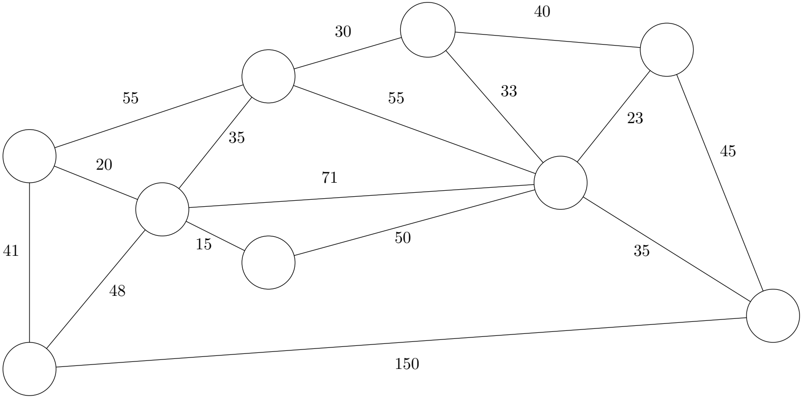
Árboles de Cobertura – Definiciones

1. Dado un grafo no dirigido $G = (V, E)$, se dice que $T \subset E$ define un árbol de cobertura (*spanning tree*) de G si (V, T) es un grafo no dirigido acíclico y conexo.

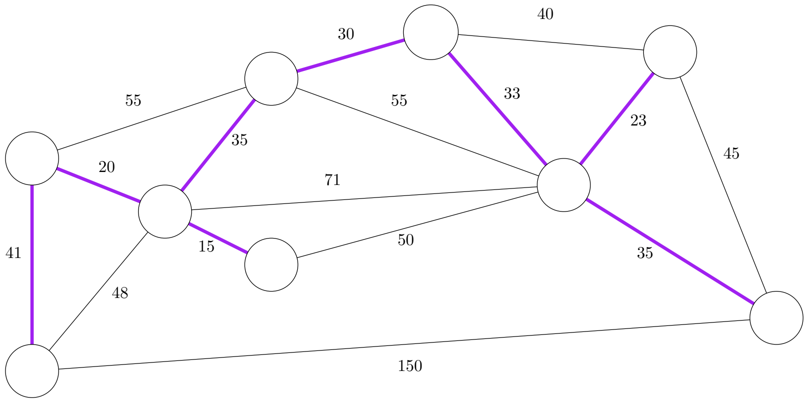
2. Dado un grafo $G = (V, E)$, una función de peso $w : E \to \mathbb{R}$, y un subconjunto T de E, decimos que $w(T) = \sum_{t \in T} w(t)$.

3. Sea $\mathcal{T}$ el conjunto de todos los árboles de cobertura de G. Un árbol de cobertura de costo mínimo (MST, por *minimum spanning tree*) es cualquier elemento $t \in \mathcal{T}$ tal que $w(t) \leq w(t')$, para todo $t' \in \mathcal{T}$.

Un Ejemplo



MST para Nuestro Ejemplo



Un Teorema Importante

Definición: Un *corte* en un grafo $G = (V, E)$ está definido por un subconjunto no vacío S de V. Una arista e *cruza un corte* S si existen $u \in S, v \in V \setminus S$ tales que $e = \{u, v\}$.

Teorema: Sea S un corte en G y e la arista de menor peso que cruza S. Entonces e es una arista del MST de G.

Algoritmo de Prim

Idea: construimos un MST incrementalmente, partiendo por con un nodo, hasta que encontramos el MST completo.

Algoritmo de Prim

Idea: construimos un MST incrementalmente, partiendo por con un nodo, hasta que encontramos el MST completo.

procedure *MST-Prim*(G, w, r)
  $Q \leftarrow$ cola de prioridades vacía (min-heap)
  for each $v \in V[G] \setminus \{r\}$ **do**
    Agregar v a Q con $key[v] \leftarrow \infty$
  Agregar r a Q con $key[r] \leftarrow 0$
  $\pi[r] \leftarrow nil$
  while $Q \neq \emptyset$ **do**
    $u \leftarrow$ Extract-Min(Q)
    for each $v \in Adj[u]$ **do**
      if $v \in Q$ *and* $w(u, v) < key[v]$ **then**
        $key[v] \leftarrow w(u, v)$
        $\pi[v] \leftarrow u$

Propiedades del Algoritmo de Prim

Teorema: Al terminar la ejecución $\{(u, v) \in E : \pi[v] = u\}$ define un MST de G.

Teorema: Si usamos un heap binario para Q, el algoritmo de Prim ejecuta en $O((|V| + |E|) \log |V|)$.

Otra Opción (Algoritmo de Kruskal)

Idea: Iterar por las aristas, partiendo desde la más pequeña.

procedure *MST-Kruskal*(G, w)
  $T \leftarrow \emptyset$
  for each $v \in V[G]$ **do**
    Crear una componente con v
  $L \leftarrow E[G]$ ordenado por w ascendentemente
  for each $\{u, v\} \in L$ **do**
    if u *y* v *pertenecen a componentes distintas* **then**
      $T \leftarrow T \cup \{\{u, v\}\}$
      Unir las componentes de u y v

¿Cuál es el tiempo de ejecución de este algoritmo?

Conjuntos Disjuntos

- Estructura para una colección de conjuntos $S_1, \ldots, S_n$.
- Conjuntos son disjuntos.
- Se da soporte a las siguientes operaciones:
  1. Make-Set(x): Construye un conjunto con un elemento x.
  2. Find-Set(x): Retorna el conjunto S_j tal que $x \in S_j$
  3. Union(x, y): Une los conjuntos en donde están x e y.

Bosques de Conjuntos Disjuntos

- Cada conjunto puede ser representado por un árbol con una raíz identificada.
- Cada elemento del conjunto contiene un puntero a otro elemento del conjunto, o a sí mismo si el elemento es la raíz.
- El representante del conjunto es la raíz.

Las operaciones se pueden implementar así:

1. Make-Set(x): crea un árbol con un elemento que apunta a sí mismo.
2. Find-Set(x): retorna al representante del árbol donde está x. Requiere "encontrar" la raíz del árbol de x.
3. Union(x, y): hace apuntar al representante de x hacia el representante de y. Requiere "encontrar" ambos representantes.

Pseudocódigo de una implementación básica

```
procedure Make-Set(x)
    padre[x] ← x
procedure Find-Set(x)
    if x ≠ padre[x] then
        return Find-Set(padre[x])
    return padre[x]
procedure Union(x, y)
    padre[Find-Set(x)] ← Find-Set(y)
```

Haciendo las operaciones eficientes (Idea 1)

Find-Set y Make-Set podrían funcionar mejor usando esta idea:

Cuando se encuentra al representante de x_0 siguiendo los punteros de los elementos $x_0, x_1, \ldots, x_m$, hacemos que $padre[x_i] = x_m$.

- El efecto es que los árboles quedan "planos"
- A la idea se le llama **compresión de caminos**.

Pseudocódigo con Compresión de Caminos

procedure *Make-Set*(x)
    $padre[x] \leftarrow x$

procedure *Find-Set*(x)
    if $x \neq padre[x]$ **then**
        $padre[x] \leftarrow$Find-Set$(padre[x])$
    return $padre[x]$

procedure *Union*(x, y)
    $padre[$Find-Set$(x)] \leftarrow$ Find-Set(y)

Haciendo las operaciones eficientes (Idea 2)

Como la operación Union "cuelga" un árbol a otro, conviene producir un árbol menos profundo; es decir,

Al unir x con y colgamos el Find-Set(x) como hijo del Find-Set(y) si y es más profundo que x y vice versa.

- Se usa $rango[x]$ como **estimador** de la profundidad de x.
- Esta heurística se conoce como **unión por rango**.

Pseudocódigo con Comp. de Caminos + Unión por Rango

```
procedure Make-Set(x)
    padre[x] ← x
procedure Find-Set(x)
    if x ≠ padre[x] then
        padre[x] ← Find-Set(padre[x])
    return padre[x]
procedure Union(x, y)
    Link(Find-Set(x), Find-Set(y))
```

Pseudocódigo con Comp. de Caminos + Unión por Rango

procedure *Make-Set*(x)
    $padre[x] \leftarrow x$

procedure *Find-Set*(x)
    if $x \neq padre[x]$ **then**
        $padre[x] \leftarrow \text{Find-Set}(padre[x])$
    return $padre[x]$

procedure *Union*(x, y)
    $\text{Link}(\text{Find-Set}(x), \text{Find-Set}(y))$

procedure *Link*(x, y)
    if $rango[x] > rango[y]$ **then**
        $padre[y] \leftarrow x$
    else
        $padre[x] \leftarrow y$
        if $rango[x] = rango[y]$ **then**
            $rango[y] \leftarrow rango[y] + 1$

Propiedad Bosques+Compresión+Union p. Rango

Theorem

Una secuencia de m operaciones de Make-Set, Link, y Find-Set, n de las cuales son Make-Set, es $O(m \log^* n)$ para el peor caso

Con:

- $\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$
- $\log^{(i)} n$: i aplicaciones de la operación $\log$ sobre n.

Caminos Más Cortos Cuando Hay Costos

- **Escenario:** Cuando existe una función de peso/costo $w : E \to \mathbb{R}$, definimos

$$\delta(u, v) = \min\{w(p) : u \overset{p}{\rightsquigarrow} v\},$$

donde $\min$ se considera ∞ si el conjunto es vacío.

- **Problemas:**
  1. Dado u, computar $\delta(u, v)$, para todo $v \in V$.
  2. Dados u y v, computar $\delta(u, v)$.

procedure *Init*()
    for each $u \in V[G]$ **do**
        $d[u] \leftarrow \infty$;$\pi[u] \leftarrow nil$

procedure *CaminosMasCortos(G,s)*
    Init()
    $d[s] \leftarrow 0$
    for $i \leftarrow 1$ **to** $|V| - 1$ **do**
        for each $(u, v) \in E$ **do**
            $costo \leftarrow d[u] + w(u, v)$
            if $costo < d[v]$ **then**
                $d[v] \leftarrow costo$
                $\pi[v] \leftarrow u$

Propiedades de CaminosMasCortos

Teorema: Si $w \geq 0$, al terminar una ejecución de CaminosMasCortos(G, s), $d[u] = \delta(s, u)$, para todo $u \in V$.

Demostración: pizarra.

Propiedad: El tiempo de ejecución de CaminosMasCortos es

Propiedades de CaminosMasCortos

Teorema: Si $w \geq 0$, al terminar una ejecución de CaminosMasCortos(G, s), $d[u] = \delta(s, u)$, para todo $u \in V$.

Demostración: pizarra.

Propiedad: El tiempo de ejecución de CaminosMasCortos es $O(|V| \cdot |E|)$.

Costos Negativos

- La definición admite que $w(u, v) < 0$ para algunos $(u, v) \in E$.
- Esto podría implicar $\delta(s, u) = -\infty$, para algún $u \in V$.
- Es posible modificar nuestro anterior algoritmo para detectar esta situación.

El Algoritmo de Bellman-Ford

```
function BellmanFord(G, s)
    Init()
    d[s] ← 0
    for i ← 1 to |V| − 1 do
        for each (u, v) ∈ E do
            costo ← d[u] + w(u, v)
            if costo < d[v] then
                d[v] ← costo
                π[v] ← u
    for each (u, v) ∈ E do
        if d[v] > d[u] + w(u, v) then
            return false
    return true
```

Propiedades de Bellman-Ford

Teorema: Si BellmanFord(G, s) retorna true, $d[u] = \delta(s, u)$, para todo $u \in V$.

Teorema: El tiempo de ejecución de Bellman-Ford es

Propiedades de Bellman-Ford

Teorema: Si BellmanFord(G, s) retorna true, $d[u] = \delta(s, u)$, para todo $u \in V$.

Teorema: El tiempo de ejecución de Bellman-Ford es $O(|V| \cdot |E|)$.