Grafos
Búsqueda

MSc Edson Ticona Zegarra

Taller avanzado 2025

Contenido

Contenido

Grafo

- Un grafo es un par $G = (V, E)$ de conjuntos tal que $E \subset V^2$

Grafo

- Un grafo es un par $G = (V, E)$ de conjuntos tal que $E \subset V^2$
- Así, todo elemento de E es un subconjunto de 2 elementos de V. Los elementos de V son los vértices y los elementos de E son sus aristas.

Grafo

- Un grafo es un par $G = (V, E)$ de conjuntos tal que $E \subset V^2$
- Así, todo elemento de E es un subconjunto de 2 elementos de V. Los elementos de V son los vértices y los elementos de E son sus aristas.
- Los vértices de G son denotados por $V(G)$ y sus aristas por $E(G)$

Grafo

- Un grafo es un par $G = (V, E)$ de conjuntos tal que $E \subset V^2$
- Así, todo elemento de E es un subconjunto de 2 elementos de V. Los elementos de V son los vértices y los elementos de E son sus aristas.
- Los vértices de G son denotados por $V(G)$ y sus aristas por $E(G)$
- Un vértice v es incidente con una arista e si $v \in e$

Grafo

- Un grafo es un par $G = (V, E)$ de conjuntos tal que $E \subset V^2$
- Así, todo elemento de E es un subconjunto de 2 elementos de V. Los elementos de V son los vértices y los elementos de E son sus aristas.
- Los vértices de G son denotados por $V(G)$ y sus aristas por $E(G)$
- Un vértice v es incidente con una arista e si $v \in e$
- El conjunto de vértices adyacentes a v se denota por $E(v)$

Grafo

- Un grafo es un par $G = (V, E)$ de conjuntos tal que $E \subset V^2$
- Así, todo elemento de E es un subconjunto de 2 elementos de V. Los elementos de V son los vértices y los elementos de E son sus aristas.
- Los vértices de G son denotados por $V(G)$ y sus aristas por $E(G)$
- Un vértice v es incidente con una arista e si $v \in e$
- El conjunto de vértices adyacentes a v se denota por $E(v)$
- El grado $d_G(v)$ o $d(v)$ de un vértice es el número $|E(v)|$ de aristas en v

Grafo

- Un grafo es un par $G = (V, E)$ de conjuntos tal que $E \subset V^2$
- Así, todo elemento de E es un subconjunto de 2 elementos de V. Los elementos de V son los vértices y los elementos de E son sus aristas.
- Los vértices de G son denotados por $V(G)$ y sus aristas por $E(G)$
- Un vértice v es incidente con una arista e si $v \in e$
- El conjunto de vértices adyacentes a v se denota por $E(v)$
- El grado $d_G(v)$ o $d(v)$ de un vértice es el número $|E(v)|$ de aristas en v
- Dos vértices u, v son adyacentes si existe una arista $e = \{u, v\}$

Caminos y ciclos

- Un *camino* es un grafo no vacio $P = (V, E)$, tal que $V = \{x_0, x_1, ..., x_k\}$ y $E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\}$

Caminos y ciclos

- Un *camino* es un grafo no vacio $P = (V, E)$, tal que $V = \{x_0, x_1, ..., x_k\}$ y $E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\}$
- Donde todo x_i es diferente. El número de aristas de un camino es su tamaño o longitud

Caminos y ciclos

- Un *camino* es un grafo no vacio $P = (V, E)$, tal que $V = \{x_0, x_1, ..., x_k\}$ y $E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\}$
- Donde todo x_i es diferente. El número de aristas de un camino es su tamaño o longitud
- Si $P = x_0, ..., x_{k-1}$ es un camino y $k \geq 3$, entonce el grafo $C = P + x_{k-1}x_0$ es llamado de *ciclo*

Conectividad

- Se dice que un grafo G está *conectado* si existe un camino entre cualquier par de vértices

Conectividad

- Se dice que un grafo G está *conectado* si existe un camino entre cualquier par de vértices
- Un subgrafo maximal conectado de G es llamado de *componente* de G

Árboles y Florestas

- Un grafo acíclico es llamado de *floresta*

Árboles y Florestas

- Un grafo acíclico es llamado de *floresta*
- Un floresta conectada es llamada de *árbol*

Árboles y Florestas

- Un grafo acíclico es llamado de *floresta*
- Un floresta conectada es llamada de *árbol*
- Es decir, una floresta es un grafo cuyos componentes son árboles.

Árboles y Florestas

- Un grafo acíclico es llamado de *floresta*
- Un floresta conectada es llamada de *árbol*
- Es decir, una floresta es un grafo cuyos componentes son árboles.
- Los vértices de grado 1 de un árbol son *hojas (leaves)*

Árboles y Florestas: definiciones

- Todo par de vértices de un árbol T está conectado por un camino único

Árboles y Florestas: definiciones

- Todo par de vértices de un árbol T está conectado por un camino único
- T está minimamente conectado, es decir, T está conectado pero $T - e$ esta desconectado para cualquier $e \in T$

Árboles y Florestas: definiciones

- Todo par de vértices de un árbol T está conectado por un camino único
- T está minimamente conectado, es decir, T está conectado pero $T - e$ esta desconectado para cualquier $e \in T$
- T is maximalmente acíclico, es decir, T **no** tiene ciclos pero $T + xy$ tiene ciclos para cualquier par de vértices no adyacentes $x, y \in T$

Árboles y Florestas: definiciones

- Sea T un árbol enraizado en r y sea x un vértice cualquier. Se llama de *ancestro* a todo vértice y tal que y se encuentra en el camino entre r y x

Árboles y Florestas: definiciones

- Sea T un árbol enraizado en r y sea x un vértice cualquier. Se llama de *ancestro* a todo vértice y tal que y se encuentra en el camino entre r y x
- Así mismo, se llama a x de *decendiente* de y

Cortes

- Dado un conjunto V, hacer una *partición* de V es agrupar sus elementos en subconjuntos no vacios tal que todo elemento está incluido exactamente en un subconjunto

Cortes

- Dado un conjunto V, hacer una *partición* de V es agrupar sus elementos en subconjuntos no vacios tal que todo elemento está incluido exactamente en un subconjunto
- Si V_1 y V_2 es una partición de V, el conjunto de aristas que van de V_1 a V_2 se conoce como un *corte*

Cortes

- Dado un conjunto V, hacer una *partición* de V es agrupar sus elementos en subconjuntos no vacios tal que todo elemento está incluido exactamente en un subconjunto
- Si V_1 y V_2 es una partición de V, el conjunto de aristas que van de V_1 a V_2 se conoce como un *corte*
- Note que si $V_1 = v$, entonces el corte es denotado por $E(v)$

Representación de grafos

- Matriz de adyacencia: `int AM[n][n]`

Representación de grafos

- Matriz de adyacencia: `int AM[n][n]`
- Lista de adyacencia: `vector<vector<pair<int,int>> al`. Cada par guarda el índice y el peso de la arista

Representación de grafos

- Matriz de adyacencia: `int AM[n][n]`
- Lista de adyacencia: `vector<vector<pair<int,int>> al`. Cada par guarda el índice y el peso de la arista
- Lista de aristas: `vector<tuple<int, int, int>> el`. Cada tupla guarda el peso, y las aristas

Grafos implícitos

- Algunos grafos no son almacenados directamene, sino que son generados explícitamente en alguna operación

Grafos implícitos

- Algunos grafos no son almacenados directamene, sino que son generados explícitamente en alguna operación
- Estos grafos se conocen como grafos *implícitos*

Grafos implícitos

- Algunos grafos no son almacenados directamene, sino que son generados explícitamente en alguna operación
- Estos grafos se conocen como grafos *implícitos*
- Por ejemplo, navegar una grilla 2D, un tablero de ajedrez, etc

Breadth First Search: Búsqueda en amplitud

```
toVisit.push(u) ;            // toVisit es una cola, nodo u arbitrario
while !toVisit.empty() do
    u = toVisit.front() ;
    for v ∈ E(u) do
        if !visited(v) then
            toVisit.push(v) ;
            dist[v] = dist [u] + 1 ;
            parent[v] = u ;
            mark u as visited ;
        end
    end
    toVisit.pop()
end
```

Breadth First Search: Búsqueda en amplitud

- La complejidad temporal es $O(|V| + |E|)$

Breadth First Search: Búsqueda en amplitud

- La complejidad temporal es $O(|V| + |E|)$
- La complejidad espacial es $O(|V|)$

Caminos mínimos

- Para grafos sin pesos, BFS encuentra la distancia mínima $\delta(s, v)$ entre s y v, definiendo la distancia como el tamaño del camino entre s y v, $\forall v \in V(G)$

Caminos mínimos

- Para grafos sin pesos, BFS encuentra la distancia mínima $\delta(s, v)$ entre s y v, definiendo la distancia como el tamaño del camino entre s y v, $\forall v \in V(G)$
- En el seudocódigo, $\delta(s, u)$ será $dist[u]$ donde s es el vértice de inicio

Breadth-first Tree

- El órden de visita de los vértices define un *breadth-first tree*.

Breadth-first Tree

- El órden de visita de los vértices define un *breadth-first tree*.
- Definimos el breadth-first tree como $G_\pi = (V_\pi, E_\pi)$ tal que $V_\pi = \{v \in V : v.parent \neq NIL\} \cup \{s\}$ y $E_\pi = \{(v.parent, v) : v \in V_\pi - \{s\}\}$

Breadth-first Tree

- El órden de visita de los vértices define un *breadth-first tree*.
- Definimos el breadth-first tree como $G_\pi = (V_\pi, E_\pi)$ tal que $V_\pi = \{v \in V : v.parent \neq NIL\} \cup \{s\}$ y $E_\pi = \{(v.parent, v) : v \in V_\pi - \{s\}\}$
- La estructura *parent* contiene el breadth-first tree enraizado en s

Breadth-first Tree

- El órden de visita de los vértices define un *breadth-first tree*.
- Definimos el breadth-first tree como $G_\pi = (V_\pi, E_\pi)$ tal que $V_\pi = \{v \in V : v.parent \neq NIL\} \cup \{s\}$ y $E_\pi = \{(v.parent, v) : v \in V_\pi - \{s\}\}$
- La estructura *parent* contiene el breadth-first tree enraizado en s
- G_π consiste de los vértices alcanzables desde s y a su vez contiene un camino único de s a v, siendo el camino mínimo

Breadth-first Tree: Imprimir camino

input : G, s, v : G grafo, camino de s a v
if $v == s$ **then**
  print s
end
else
  if $v.parent == NIL$ **then**
    print "no path"
  end
  else
    PrintPath($G, s, v.parent$)
    print v
  end
end

Depth First Search: Búsqueda en profundidad

```
toVisit.push(u) ;    // toVisit es una pila y se comienza
en un vértice arbitrario u
while !toVisit.empty() do
    u = toVisit.front() /* Adj(u) adyacentes a u        */
    for v ∈ Adj(u) do
        /* colorear los vértices                         */
        if !visited(v) then
            toVisit.push(v) ;
            parent[v] = u ;
            mark u as visited ;
        end
    end
    toVisit.pop()
end
```

Depth First Search (DFS): Búsqueda en profundidad

mark u as visited
$u.d \leftarrow time++$
for $v \in Adj(u)$ **do**
    if $!visited(v)$ **then**
        $parent[v] = u$
        $DFS(v)$
    end
end
$u.f \leftarrow ++time$

Depth First Search: Búsqueda en amplitud

- $u.d$ es el tiempo de *discovery*

Depth First Search: Búsqueda en amplitud

- $u.d$ es el tiempo de *discovery*
- $u.f$ es el tiempo de *finalización*

Depth First Search: Búsqueda en amplitud

- $u.d$ es el tiempo de *discovery*
- $u.f$ es el tiempo de *finalización*
- La complejidad temporal es $O(|V| + |E|)$

Depth First Search: Búsqueda en amplitud

- $u.d$ es el tiempo de *discovery*
- $u.f$ es el tiempo de *finalización*
- La complejidad temporal es $O(|V| + |E|)$
- La complejidad espacial es $O(|V|)$

Depth First Search (DFS): Clasificación de aristas

- **Tree edges**: aristas en el grafo G_π, siendo G_π el *depth-first tree*

Depth First Search (DFS): Clasificación de aristas

- **Tree edges**: aristas en el grafo G_π, siendo G_π el *depth-first tree*
- **Back edges**: aristas conectando un vértice u a un ancestro v en un *depth-first tree*

Depth First Search (DFS): Clasificación de aristas

- **Tree edges**: aristas en el grafo G_π, siendo G_π el *depth-first tree*
- **Back edges**: aristas conectando un vértice u a un ancestro v en un *depth-first tree*
- **Forward edges**: aristas conectando un vértice u a un decendiente v en un *depth-first tree*

Depth First Search (DFS): Clasificación de aristas

- **Tree edges**: aristas en el grafo G_π, siendo G_π el *depth-first tree*
- **Back edges**: aristas conectando un vértice u a un ancestro v en un *depth-first tree*
- **Forward edges**: aristas conectando un vértice u a un decendiente v en un *depth-first tree*
- **Cross edges**: el resto de aristas

Orden topológico

- Un ordenamiento topológico se define para un grafo dirigido acíclico (DAG en inglés)

Orden topológico

- Un ordenamiento topológico se define para un grafo dirigido acíclico (DAG en inglés)
- Es un ordenamiento lineal de los vértices tal que u aparece antes de v si existe una arista dirigida (u, v).

Orden topológico

- Un ordenamiento topológico se define para un grafo dirigido acíclico (DAG en inglés)
- Es un ordenamiento lineal de los vértices tal que u aparece antes de v si existe una arista dirigida (u, v).
- Podemos usar el DFS para hallar los tiempos de finalización y usarlos de referencia para el órden

Ordenamiento topológico

$DFS(G)$
SORT en base a u.f

Ordenamiento topológico: Algoritmo de Kahn

for $u \in V(G)$ **do**
  if $d(u) == 0$ **then**
    $toVisit.push(u)$; // *toVisit* `es una cola`
  end
end
while $!toVisit.empty()$ **do**
  $u \leftarrow toVisit.pop()$
  for $v \in E(u)$ **do**
    $d(v) \leftarrow d(v) - 1$
    if $d(v) > 0$ **then**
      continue
    end
    $toVisit.push(v)$
  end
end

Componentes conectados

- Podemos utilizar el DFS o BFS para hallar los compoenentes conectados

Componentes conectados

- Podemos utilizar el DFS o BFS para hallar los compoenentes conectados
- Ambos van a recorrer cada componente por cada llamada

Verificar grafos bipartitos

- Se dice que un grafo $G = (V, E)$, si V admite una partición en dos tal que toda arista tenga un extremo en cada partición.

Verificar grafos bipartitos

- Se dice que un grafo $G = (V, E)$, si V admite una partición en dos tal que toda arista tenga un extremo en cada partición.
- En general se dice que un grafo es r-partito si admite r particiones tal que toda arista tenga un extremo en alguna partición

Verificar grafos bipartitos

- Se dice que un grafo $G = (V, E)$, si V admite una partición en dos tal que toda arista tenga un extremo en cada partición.
- En general se dice que un grafo es r-partito si admite r particiones tal que toda arista tenga un extremo en alguna partición
- Vértices en la misma partición no pueden ser adyacentes

Verificar grafos bipartitos

- Se dice que un grafo $G = (V, E)$, si V admite una partición en dos tal que toda arista tenga un extremo en cada partición.
- En general se dice que un grafo es r-partito si admite r particiones tal que toda arista tenga un extremo en alguna partición
- Vértices en la misma partición no pueden ser adyacentes
- Podemos usar BFS o DFS para ello, coloreando los vértices de manera alternada

Verificar ciclos

- Podemos utilizar un DFS para verificar existencia de ciclos

Verificar ciclos

- Podemos utilizar un DFS para verificar existencia de ciclos
- Si se encuentra un back edge es una prueba de que existe un ciclo