# TEORIA 2DA PARCIAL

## ANALISIS DE ALGORITMOS

Nos permite comparar algoritmos en forma independiente de una plataforma en particular.
Mide la eficiencia de un algoritmo, dependiendo del tamaño de entrada.

- Notación BIG-OH:
  Decimos que T(n) = O(f(n)) si existen tales constantes c > 0 y n sub0 tales que: T(n) <= c f(n) para todo n <= n sub0.
  Se lee: T(n) es de orden f(n)
  f(n) representa una cota superior de T(n)
  La tasa de crecimiento de T(n) es menor o igual que la de f(n)
  - Regla de la suma y regla del producto:
    - Si T1(n)=O(f(n)) y T2(n)=O(g(n)), entonces:
    1. T1(n)+T2 (n)=max(O(f(n)), O(g(n)))
    2. T1(n)·T2 (n)=O(f(n)·g(n))
  - Otras reglas:
    - T(n) es un polinomio de grado k → T(n) = O(n elevado k)
    - T(n) = log elevado k(n) → O(n) para cualquier k
      n siempre crece más rápido que cualquier potencia de log(n)
    - T(n) = cte → O(1)
    - T(n) = cte * f(n) → T(n) = O( f(n) )
- OMEGA:
  Decimos que T(n) = Ω(g(n)) si existen constantes c > 0 y n sub0 tales que: T(n) ≥ c g(n) para todo n >= n sub0
  Se lee: T(n) es omega de g(n)
  g(n) representa una cota inferior de T(n)
  La tasa de crecimiento de T(n) es mayor o igual que la de g(n)
- THETA:
  Decimos que T(n) = Θ(h(n)) ↔ T(n) = O(h(n)) y T(n) = Ω(h(n))
  Se lee: T(n) es theta de h(n)
  T(n) y h(n) tienen la misma tasa de crecimiento

- Algunas funciones:

| Ordenadas en forma creciente | Nombre |
|---|---|
| **1** | Constante |
| **log n** | Logaritmo |
| **n** | Lineal |
| **n log n** | n Log n |
| **$n^2$** | Cuadrática |
| **$n^3$** | Cúbica |
| **$c^n$ c>1** | Exponencial |

- Algunos ejemplos:

➢Iteración:

***a) For***

Viene como parámetro

```
int sum = 0;
int [] a = new int [n];
for (int i =1; i<= n ; i++ )
    sum += a[i];
```

$$T(n) = cte_1 + \sum_{i=1}^{n} cte_2 =$$

$$= cte_1 + n * cte_2$$

$$\Rightarrow O(n)$$

**a) For**

Viene como parámetro

```
int sum = 0;
int [] a = new int [n][n];
for (int i =1; i<= n ; i++) {
  for (int j =1; j<= n ; j++)
     sum += a[i][j];
}
```

$$T(n) = cte_1 + \sum_{i=1}^{n}\sum_{j=1}^{n} cte_2 =$$

$$= cte_1 + n*n*cte_2$$

$$\Rightarrow O(n^2)$$

**a) For**

Viene como parámetro

```
int [] a = new int [n];
int [] s = new int [n];
for ( int i =1; i<= n ; i++ )
     s[i] = 0;
for ( int i =1; i<= n ; i++) {
  for (int j =1; j<= i ; j++)
      s[i] += a[j];
}
```

$$T(n) = cte_1 + \sum_{i=1}^{n} cte_2 + \sum_{i=1}^{n}\sum_{j=1}^{i} cte_3 =$$

$$= cte_1 + n * cte_2 + cte_3 * \sum_{i=1}^{n} i = \ldots\ldots$$

$$\Rightarrow O(n^2)$$

**b) While**

```
int x= 0;
int i = 1;
while ( i <= n) {
      x = x + 1;
      i = i + 2;
}
```

$$T(n) = cte_1 + \sum_{i=1}^{(n+1)/2} cte_2 =$$

$$= cte_1 + cte_2/2 * (n+1)$$

$$\Rightarrow O(n)$$

**b) While**

```
int x= 1;
while ( x < n)
   x = 2 *x;
```

$$T(n) = cte_1 + cte_2 * \log(n)$$

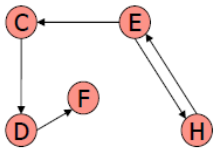
$$\Rightarrow O(\log(n))$$

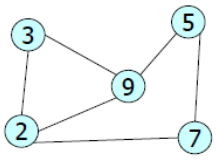
**GRAFOS**

**TERMINOLOGIAS Y EJEMPLOS**

- Grafo: modelo para representar relaciones entre elementos de un conjunto.
- Grafo: (V, E), V es un conjunto de vértices o nodos, con una relación entre ellos; E es un conjunto de pares (u, v), u, v Є V, llamados aristas o arcos.
- Grafo dirigido: la relación sobre V no es simétrica. Arista ≡ par ordenado (u, v).
- Grafo no dirigido: la relación sobre V es simétrica. Arista ≡ par no ordenado {u, v}, u, v Є V y u ≠ v.

- Ejemplos:

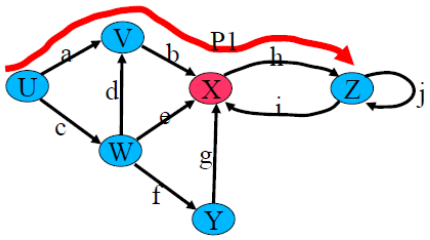


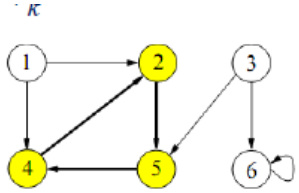

*Grafo dirigido G(V,E).*
*V = {C,D,E,F,H}*
*E={(C,D),(D,F),(E,C),(E,H),*
*(H,E)}*

*Grafo no dirigido G(V,E).*
*V = {2,3,5,7,9}*
*E= {{2,3},{2,7},{2,9},{3,9},*
*{5,7},{5,9}}*

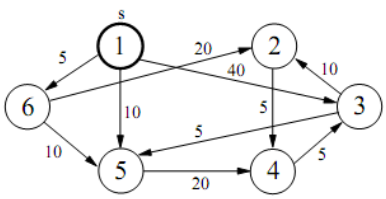
  - v es adyacente a u si existe una arista (u, v) Є E.
  - en un grafo no dirigido, (u, v) Є E incide en los nodos u, v.
  - en un grafo dirigido, (u, v) Є E incide en v, y parte de u.
  - En grafos no dirigidos:
    - El grado de un nodo: número de arcos que inciden en él.
  - En grafos dirigidos:
    - existen el grado de salida (grado_out) y el grado de entrada (grado_in).
      - el grado_out es el número de arcos que parten de él.
      - el grado_in es el número de arcos que inciden en él.
    - El grado del vértice será la suma de los grados de entrada y de salida.
- Grado de un grafo: máximo grado de sus vértices.
- Camino desde u Є V a v Є V: secuencia v1, v2, ..., vk tal que u=v1, v=v k, y (vi−1,vi) Є E, para i = 2, ... ,k.
- Longitud de un camino: número de arcos del camino.
- Camino simple: camino en el que todos sus vértices, excepto, tal vez, el primero y el último, son distintos. P1 es un camino simple desde U a Z.



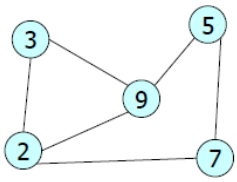

- Ciclo: camino desde v1, v2, ..., vk tal que v1=vk. Ej: <2,5,4,2> es un ciclo de longitud 3.
El ciclo es simple si el camino es simple.



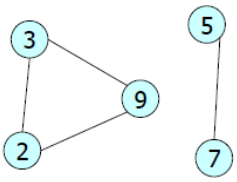

- Bucle: ciclo de longitud 1. Por ejemplo, el vértice 6 con la arista apuntadose a si mismo.
- Grafo acíclico: grafo sin ciclos.
- Un grafo ponderado, pesado o con costos: cada arco o arista tiene asociado un valor o etiqueta.




- Conectividad en grafos no dirigidos: Un grafo no dirigido es conexo si hay un camino entre cada par de vértices.

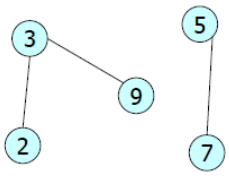



Conexo

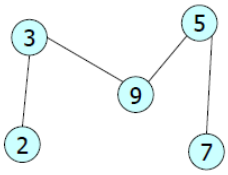



No Conexo

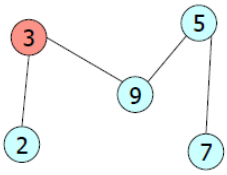
- Bosque: Un bosque es un grafo sin ciclos.
- Árbol libre: Un árbol libre es un bosque conexo.
- Árbol: Un árbol es un árbol libre en el que un nodo se ha designado como raíz.




Bosque

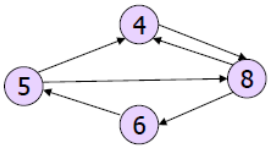



Árbol libre

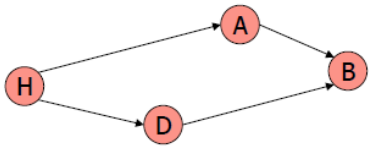



Árbol

- Conectividad en grafos dirigidos:
  - v es alcanzable desde u, si existe un camino de u a v.
  - Un grafo dirigido se denomina fuertemente conexo si existe un camino desde cualquier vértice a cualquier otro vértice
  - Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin sentido en los arcos) es conexo, el grafo es débilmente conexo.




Fuertemente Conexo



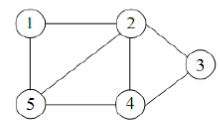

No Fuertemente Conexo
Débilmente Conexo

- Componentes conexas:
  - En un grafo no dirigido, una componente conexa es un subgrafo conexo tal que no existe otra componente conexa que lo contenga.
  - Es un subgrafo conexo maximal.
  - Un grafo no dirigido es no conexo si está formado por varias componentes conexas.
- Componentes fuertemente conexas:
  - En un grafo dirigido, una componente fuertemente conexa, es el máximo subgrafo fuertemente conexo.
  - Un grafo dirigido es no fuertemente conexo si está formado por varias componentes fuertemente conexas.

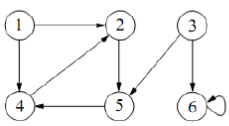
**REPRESENTACIONES**

- Matriz de adyacencias:
  - G = (V, E): matriz A de dimensión |V |×|V |.
  - Valor a sub ij de la matriz:

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



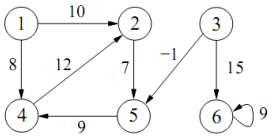

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

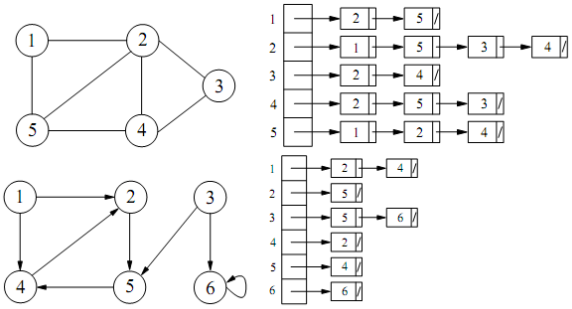
- Costo espacial: O (|V|elevado 2)
- Representación es útil para grafos con número de vértices pequeño, o grafos densos (|E|≈|V|×|V|)
- Comprobar si una arista (u,v) pertenece a E → consultar posición A(u,v)
  - Costo de tiempo T(|V|,|E|) = O(1)
- Representación aplicada a Grafos pesados:
  - El peso de (i,j) se almacena en A (i, j)

$$a_{ij} = \begin{cases} w(i,j) & \text{si } (i,j) \in E \\ 0 \ o \ \infty & \text{en cualquier otro caso} \end{cases}$$

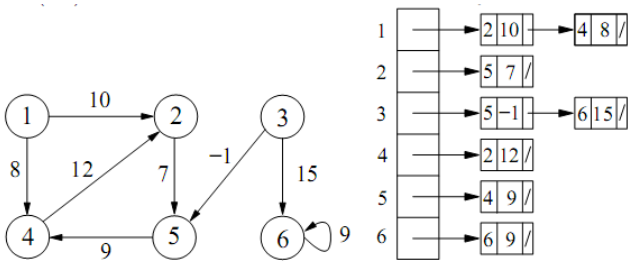



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 10 | 0 | 8 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 7 | 0 |
| 3 | 0 | 0 | 0 | 0 | −1 | 15 |
| 4 | 0 | 12 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 9 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 9 |

- <u>Lista de adyacencias:</u>
  - G = (V, E): vector de tamaño |V|.
  - Posición i → puntero a una lista enlazada de elementos (lista de adyacencia). Los elementos de la lista son los vértices adyacentes a i




  - Si G es dirigido, la suma de las longitudes de las listas de adyacencia será |E|.
  - Si G es no dirigido, la suma de las longitudes de las listas de adyacencia será 2|E|.
  - Costo espacial, sea dirigido o no: O (|V|+|E|).
  - Representación apropiada para grafos con |E| menor que |V|2.
  - Desventaja: si se quiere comprobar si una arista (u, v) pertenece a E → buscar v en la lista de adyacencia de u.
  - Costo temporal T (|V|, |E|) será O (Grado G) C O (|V|).
  - Representación aplicada a Grafos pesados:
    - El peso de (u, v) se almacena en el nodo de v de la lista de adyacencia de u.




**<u>RECORRIDOS</u>**

- <u>En profundidad (**DFS**):</u> Generalización del recorrido preorden de un árbol.
  - Estrategia:
    - Partir de un vértice determinado v.
    - Cuando se visita un nuevo vértice, explorar cada camino que salga de él.
    - Hasta que no se haya finalizado de explorar uno de los caminos no se comienza con el siguiente.
    - Un camino deja de explorarse cuando se llega a un vértice ya visitado.

- Si existían vértices no alcanzables desde v el recorrido queda incompleto; entonces, se debe seleccionar algún vértice como nuevo vértice de partida, y repetir el proceso.

- Esquema recursivo: dado G = (V, E)

1. Marcar todos los vértices como no visitados.
2. Elegir vértice u como punto de partida.
3. Marcar u como visitado.
4. Para todo v adyacente a u, (u, v) Є E, si v no ha sido visitado, repetir recursivamente (3) y (4) para v.

• Finalizar cuando se hayan visitado todos los nodos alcanzables desde u.

• Si desde u no fueran alcanzables todos los nodos del grafo: volver a (2), elegir un nuevo vértice de partida v no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

```
dfs (v: vértice)
   marca[v]:= visitado;
   para cada nodo w adyacente a v
         si w no está visitado
               dfs(w);


main:dfs (grafo)
      inicializar marca en false (arreglo de booleanos);
      para cada vértice v del grafo
          si w no está visitado
             dfs(v);
```

- Tiempo de ejecución de un BFS:
  - G (V, E) se representa mediante listas de adyacencia.
  - El método dfs(v) se aplica únicamente sobre vértices no visitados
    → sólo una vez sobre cada vértice.
  - dfs(v) depende del número de vértices adyacentes que tenga (longitud de la lista de adyacencia).
    → el tiempo de todas las llamadas a dfs(v): O(|E|)
  - añadir el tiempo asociado al bucle de main_dfs(grafo): O (|V|)). Tiempo del recorrido en profundidad es O(|V|+|E|).

- <u>En amplitud (**BFS**):</u> Generalización del recorrido por niveles de un árbol.
  - Estrategia:
    - Partir de algún vértice u, visitar u y, después, visitar cada uno de los vértices adyacentes a u.
    - Repetir el proceso para cada nodo adyacente a u, siguiendo el orden en que fueron visitados.
  - Esquema iterativo: dado G = (V, E)

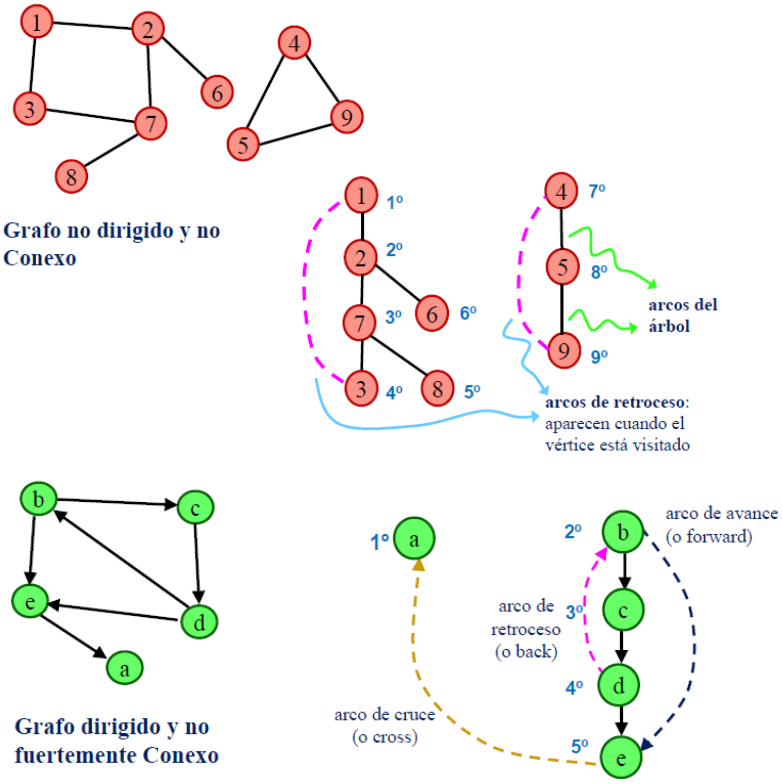
1. Encolar el vértice origen u.
2. Marcar el vértice u como visitado.
3. Procesar la cola.
4. Desencolar u de la cola
5. Para todo adyacente a u, (u, v) Є E,
6. si v no ha sido visitado
7. encolar y visitar v

• Si desde u no fueran alcanzables todos los nodos del grafo: volver a (1), elegir un nuevo vértice de partida no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices

  - Costo T(|V|,|E|) es de O(|V|+|E|)

- <u>Bosque de expansión del DFS:</u>
  - El recorrido no es único: depende del nodo inicial y del orden de visita de los adyacentes.

- El orden de visita de unos nodos a partir de otros puede ser visto como un árbol: árbol de expansión (o abarcador) en profundidad asociado al grafo.
- Si aparecen varios árboles: bosque de expansión (o abarcador) en profundidad.







Bosque de expansión, empezando el recorrido en el vértice **a**

- Clasificación de los arcos de un grafo dirigido en el bosque de expansión de un DFS.
  - Arcos tree (del árbol): son los arcos en el bosque depth-first-search, arcos que conducen a vértices no visitados durante la búsqueda.
  - Arcos forward: son los arcos u → v que no están en el bosque, donde v es descendiente, pero no es hijo en el árbol.
  - Arcos backward: son los arcos u → v, donde v es antecesor en el árbol. Un arco de un vértice a sí mismo es considerado un arco back.
  - Arcos cross: son todos los otros arcos u → v, donde v no es ni antecesor ni descendiente de u. Son arcos que pueden ir entre vértices del mismo árbol o entre vértices de diferentes árboles en el bosque depth-first-search

- <u>Algoritmo de Kosaraju</u>: Pasos:
  1. Aplicar DFS(G) rotulando los vértices de G en post-orden (apilar).
  2. Construir el grafo reverso de G, es decir GR (invertir los arcos).
  3. Aplicar DFS (GR) comenzando por los vértices de mayor rótulo (tope de la pila).
  4. Cada árbol de expansión resultante del paso 3 es una componente fuertemente conexa.
     - Si resulta un único árbol entonces el dígrafo es fuertemente conexo.
  - Complejidad:
    - Se realizan dos DFS
    - Se recorren todas las aristas una vez para crear el grafo reverso
    - O (|V| + |E|)

**<u>ORDENACION TOPOLOGICA</u>**

- La ordenación topológica es una permutación:
- v1, v2, v3, ..., v sub|V|de los vértices, tal que si (vi, vj) Є E, vi ≠ vj, entonces vi precede a vj en la permutación.
- La ordenación no es posible si G es cíclico.
- La ordenación topológica no es única.

- Una ordenación topológica es como una ordenación de los vértices a lo largo de una línea horizontal, con los arcos de izquierda a derecha.

**ALGORITMOS**

- Con complejidad O(|V|2): Implementación con Arreglo
  - En esta versión el algoritmo utiliza un arreglo grado_in en el que se almacenan los grados de entradas de los vértices y en cada paso se toma de allí un vértice con grado_in = 0.
  - Pasos generales:
    1. Seleccionar un vértice v con grado de entrada cero
    2. Visitar v
    3. "Eliminar" v, junto con sus aristas salientes
    4. Repetir el paso 1 hasta seleccionar todos los vértices

```
int sortTopologico( ){
    int numVerticesVisitados = 0;
    while(haya vertices para visitar){
        if(no existe vertice con grado_in = 0)
            break;
        else{
         seleccionar un vertice v con grado_in = 0;
         visitar v; //mandar a la salida
         numVerticesVisitados++;
         borrar v y todas sus aristas salientes;
        }
    }

   return numVerticesVisitados;
}
```

Búsqueda secuencial en el arreglo

Decrementar el grado de entrada de los adyacentes de v

El tiempo total del algoritmo es de O (|V| elevado a 2 +|E|)

- Con complejidad O (|V| + |A|): Implementación con Pila o Cola
  - En esta versión el algoritmo utiliza un arreglo Grado_in en el que se almacenan los grados de entradas de los vértices y una pila P (o una cola Q) en donde se almacenan los vértices con grados de entrada igual a cero.

```
int sortTopologico( ){
    int numVerticesVisitados = 0;
    while(haya vertices para visitar){
        if(no existe vertice con grado_in = 0)
            break;
        else{
         seleccionar un vertice v con grado_in = 0;
         visitar v; //mandar a la salida
         numVerticesVisitados++;
         borrar v y todas sus aristas salientes;
        }
    }

   return numVerticesVisitados;
}
```

Tomar el vértice de la cola

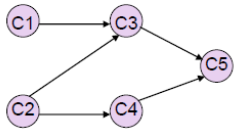
Decrementar el grado de entrada de los adyacentes de v. Si llegó a 0, encolarlo

El tiempo total del algoritmo es de O (|V| +|E|)

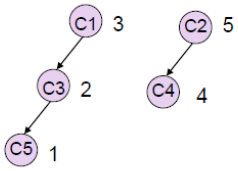
- Con complejidad O (|V| + |A|): DFS
  - Se realiza un recorrido DFS, marcando cada vértice en post-orden, es decir, una vez visitados todos los vértices a partir de uno dado, el marcado de los vértices en post-orden puede implementarse según una de las sig. opciones:
    a) numerándolos antes de retroceder en el recorrido; luego se listan los vértices según sus números de post-orden de mayor a menor.
    b) colocándolos en una pila P, luego se listan empezando por el tope.

→ *Aplicando el recorrido en profundidad.*

*Opción **a)** - numerando los vértices*




Grafo dirigido acíclico

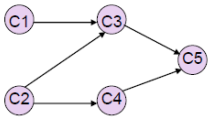



Aplico DFS a partir de un vértice cualquiera, por ejemplo C1

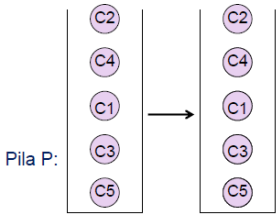
***Ordenación Topológica:*** ***C2 C4 C1 C3 C5***

→ *Aplicando el recorrido en profundidad.*

*Opción **b)** - apilando los vértices*




Grafo dirigido acíclico




1.- Aplico DFS a partir de un vértice cualquiera, por ejemplo C1, y apilo los vértice en post-orden.
2.- Listo los vértices a medida que los desapilo.

***Ordenación Topológica:*** ***C2 C4 C1 C3 C5***

## GRAFOS – CAMINOS DE COSTO MINIMO

- Definición:
  - Sea G=(V,A) un grafo dirigido y pesado, el costo c(i, j) está asociado a la arista v(i, j).
  - Dado un camino: v1, v2, v3,…….vN
    - El costo del camino es:

$$C = \sum_{i=1}^{N-1} c(i, i+1)$$

  - Este valor también se llama longitud del camino pesado. La longitud del camino no pesado es la cantidad de aristas
  - El camino de costo mínimo desde un vértice vi a otro vértice vj es aquel en que la suma de los costos de las aristas es mínima.

$$C = \sum_{i=1}^{N-1} c(i, i+1)$$

    - Esto significa que: es mínima
- Algoritmos de caminos mínimos:
  - Los algoritmos calculan los caminos mínimos desde un vértice origen s a todos los restantes vértices del grafo.
  - Grafos sin peso:
    - Para cada vértice v mantiene la siguiente información:
      - Dv: distancia mínima desde el origen (inicialmente ∞ para todos los vértices excepto el origen con valor 0)
      - Pv: vértice por donde paso para llegar
      - Conocido: dato booleano que me indica si está procesado (inicialmente todos en 0) (este último campo no va a ser necesario para esta clase de grafos)
    - Estrategia: Recorrido en amplitud (BFS)
      - Pasos:
        - Avanzar por niveles a partir del origen, asignando distancias según se avanza (se utiliza una cola)
        - Inicialmente, es Dw=∞. Al inspeccionar w se reduce al valor correcto Dw=Dv+1
        - Desde cada v, visitamos a todos los nodos adyacentes a v
    - Algoritmo:

```
   Camino_min_GrafoNoPesadoG,s) {
(1)    para cada vértice v ∈ V
(2)        Dv = ∞; Pv = 0; Conocv= 0;
(3)    Ds = 0; Encolar (Q,s); Conocv= 1;
(4)    Mientras (not esVacio(Q)) {
(5)        Desencolar (Q,u);
(6)        Marcar u como conocido;
(7)        para c/vértice w ∈ V adyacente a u {
(8)            si (w no es conocido) {
(9)                    Dw = Du + 1;
(10)                   Pw = u;
(11)                   Encolar(Q,w); Conocw= 1;
(12)           }
(13)       }
(14)   }
   }
```

- Grafos con pesos positivos:
  - Algoritmo de Dijkstra:
    - Pasos:
      - Dado un vértice origen s, elegir el vértice v que esté a la menor distancia de s, dentro de los vértices no procesados
      - Marcar v como procesado
      - Actualizar la distancia de w adyacente a v
    - Para cada vértice v mantiene la siguiente información:
      - Dv: distancia mínima desde el origen (inicialmente ∞ para todos lo vértices excepto el origen con valor 0)
      - Pv: vértice por donde paso para llegar
      - Conocido: dato booleano que me indica si está procesado (inicialmente todos en 0)
    - La actualización de la distancia de los adyacentes w se realiza con el siguiente criterio:

➢ Se compara $D_w$ con $D_v + c(v,w)$

Distancia de $s$ a $w$ (sin pasar por $v$)

Distancia de $s$ a $w$, pasando por $v$

➢ Se actualiza si $D_w > D_v + c(v,w)$

- Algoritmo:

```
   Dijkstra(G,w, s){
(1)    para cada vértice v ∈ V
(2)        Dv = ∞;    Pv = 0;
(3)    Ds = 0;
(4)    para cada vértice v ∈ V {
(5)        u = vérticeDesconocidoMenorDist;
(6)        Marcar u como conocido;
(7)        para cada vértice w ∈ V adyacente a u
(8)            si (w no está conocido)
(9)                si (Dw > Du + c(u,w)) {
(10)                   Dw = Du + c(u,w);
(11)                   Pw = u;
(12)               }
(13)       }
(14)   }
   }
```

- Tiempo de ejecución:
  - El bucle para de la línea (4) se ejecuta para todos los vértices
    - ➔ |V| iteraciones
  - La operación vérticeDesconocidoMenorDist es O(|V|) y dado que se realiza |V| veces
    - ➔ el costo total de vérticeDesconocidoMenorDist es O(|V|2)

- El bucle para de la línea (7) se ejecuta para los vértices adyacentes de cada vértice. El número total de iteraciones será la cantidad de aristas del grafo.
  ➔ |E| iteraciones
- El costo total del algoritmo es (|V|2 + |E|) es O(|V|2)
- Optimización: la operación vérticeDesconocidoMenorDist es más eficiente si almacenamos las distancias en una heap.
- La operación vérticeDesconocidoMenorDist es O(log|V|) y dado que se realiza |V| veces
  ➔ el costo total de vérticeDesconocidoMenorDist es O (|V| log |V|)
- El bucle para de la línea (7) supone modificar la prioridad (distancia) y reorganizar la heap. Cada iteración es O (log|V|)
  ➔ realiza|E| iteraciones, O (|E| log|V|)
- El costo total del algoritmo es (|V| log|V|+ |E| log|V|) es O (|E| log|V|)
- Variante: usando heap la actualización de la línea (10) se puede resolver insertando w y su nuevo valor Dw cada vez que éste se modifica.
- El tamaño de la heap puede crecer hasta |E|.
  Dado que |E| ≤ |V|2, log |E| ≤ 2 log |V|, el costo total del algoritmo no varía
- El costo total del algoritmo es O (|E| log|V|)

- Grafos con pesos positivos y negativos:
  - Estrategia: Encolar los vértices
  - Si el grafo tiene aristas negativas, el algoritmo de Dijkstra puede dar un resultado erróneo.
  - Pasos:
    - Encolar el vértice origen s.
    - Procesar la cola:
      - Desencolar un vértice.
    - Actualizar la distancia de los adyacentes Dw siguiendo el mismo criterio de Dijkstra.
    - Si w no está en la cola, encolarlo.
  - El costo total del algoritmo es O (|V| |E|)
  - Algoritmo:

```
   Camino_min_GrafoPesosPositivosyNegativos(G,s) {
(1)    D_s = 0; Encolar (Q,s);
(2)    Mientras (not esVacio(Q)) {
(3)       Desencolar(Q,u);
(4)       para c/vértice w ∈ V adyacente a u {
(5)          si (D_w > D_u + c(u,w)) {
(6)                D_w = D_u + c(u,w);
(7)                P_w = u;
(8)                si (w no está en Q)
(9)                    Encolar(Q,w);
(10)          }
(11)       }
(12)   }
   }
```

- Grafos dirigidos acíclicos:
  - Estrategia: Orden Topológico
    - Optimización del algoritmo de Dijkstra
    - La selección de cada vértice se realiza siguiendo el orden topológico
    - Esta estrategia funciona correctamente, dado que al seleccionar un vértice v, no se va a encontrar una distancia dv menor, porque ya se procesaron todos los caminos que llegan a él.
  - El costo total del algoritmo es O (|V| + |E|)
  - Algoritmo:

```
Camino_min_GrafoDirigidoAcíclico(G,s){

      Ordenar topológicamente los vértices de G;

      Inicializar Tabla de Distancias(G, s);

      para c/vértice u del orden topológico
         para c/vértice w ∈ V adyacente a u
            si (D_w > D_u + c(u,w)) {
                  D_w = D_u + c(u,w);
                  P_w = u;
             }
}

Camino_min_GrafoDirigidoAcíclico(G,s) {
    Calcular el grado_in de todos los vértices;
    Encolar en Q los vértices con grado_in = 0;
    para cada vértice v ∈ V
        D_v = ∞; P_v = 0;
    D_s = 0;
    Mientras (!esVacio(Q)){
        Desencolar(Q,u);
        para c/vértice w ∈ V adyacente a u {
          Decrementar grado de entrada de w
          si (grado_in[w] = 0)
              Encolar(Q,w);
          si (D_u != ∞)
               si D_w > D_u + c(u,w) {
                   D_w = D_u + c(u,w);
                   P_w = u;
               }
        }
    }
}
```

Algoritmos y Estructuras de Datos

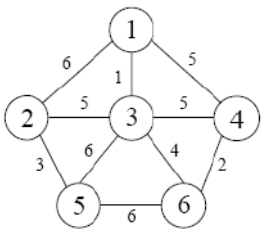
- Tabla de ayuda para los tipos de grafos

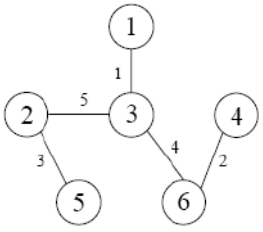
| Grafos | BFS O(V+E) | Dijkstra O(E log V) | Algoritmo modificado (encola vértices) O(V*E) | Optimización de Dijkstra (sort top) O(V+E) |
|---|---|---|---|---|
| **No pesados** | Óptimo | Correcto | Malo | Incorrecto si tiene ciclos |
| **Pesados** | Incorrecto | Óptimo | Malo | Incorrecto si tiene ciclos |
| **Pesos negativos** | Incorrecto | Incorrecto | Óptimo | Incorrecto si tiene ciclos |
| **Grafos pesados acíclicos** | Incorrecto | Correcto | Malo | Óptimo |

Correcto → adecuado pero no es el mejor
Malo → una solución muy lenta

## GRAFOS - ARBOL DE EXPANSION MINIMA

- Dado un grafo G = (V, E) no dirigido y conexo
  - El árbol de expansión mínima es un árbol formado por las aristas de G que conectan todos los vértices con un costo total mínimo.




- Algoritmo de Prim:
  - Construye el árbol haciéndolo crecer por etapas.

- Se elige un vértice como raíz del árbol.
- En las siguientes etapas:

  a) se selecciona la arista (u, v) de mínimo costo que cumpla: u pertenece árbol y v no pertenece árbol

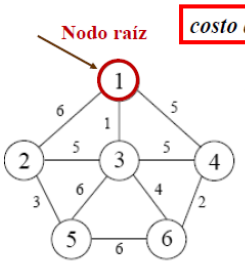
  b) se agrega al árbol la arista seleccionada en a) (es decir, ahora el vértice v pertenece árbol)

  c) se repite a) y b) hasta que se hayan tomado todos los vértices del grafo.

- Ejemplo:

➢ Construye el árbol haciéndolo crecer por etapas

Ejemplo:




1° Paso

costo de la arista (v,w)

Vértice inicial

Vértice elegido

| V | Costo | W | Conoc. |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 2 | ∞ | 0 | 0 |
| 3 | ∞ | 0 | 0 |
| 4 | ∞ | 0 | 0 |
| 5 | ∞ | 0 | 0 |
| 6 | ∞ | 0 | 0 |

1° Paso

Vértice elegido

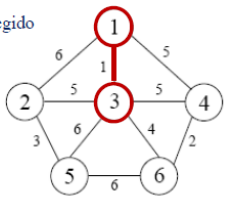
Vértices actualizados

| V | Costo | W | Conoc. |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 2 | 6 | 1 | 0 |
| 3 | 1 | 1 | 0 |
| 4 | 5 | 1 | 0 |
| 5 | ∞ | 0 | 0 |
| 6 | ∞ | 0 | 0 |

| V | Costo | W | Conoc. |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 2 | 6 | 1 | 0 |
| 3 | 1 | 1 | 1 |
| 4 | 5 | 1 | 0 |
| 5 | ∞ | 0 | 0 |
| 6 | ∞ | 0 | 0 |

Vértice elegido

2° Paso




Se agrega la arista (1,3) y el vértice 3

- Resumiendo:
  - Para la implementación se usa una tabla (similar a la utilizada en la implementación del algoritmo de Dijkstra).
  - La dinámica del algoritmo consiste en, una vez seleccionado una arista (u,v) de costo mínimo tq u pertenece árbol y v no pertenece árbol:
    - se agrega la arista seleccionada al árbol
    - se actualizan los costos a los adyacentes del vértice v de la sig. manera:

✓ se compara $Costo_w$ con c(v,w)

Costo mínimo a $w$ (costo de la arista entre un vértice perteneciente al árbol y vértice $w$)

Costo de la arista $(v,w)$

➢ se actualiza si $Costo_w > c(v,w)$

- Tiempo de ejecución:
  - Se hacen las mismas consideraciones que para el algoritmo de Dijkstra
    - Si se implementa con una tabla secuencial:

      ➔ El costo total del algoritmo es O(|V|2)
    - Si se implementa con heap:

      ➔ El costo total del algoritmo es O(|E| log|V|)

- Algoritmo de Kruskal:
  - Selecciona las aristas en orden creciente según su peso y las acepta si no originan un ciclo.
  - El invariante que usa me indica que, en cada punto del proceso, dos vértices pertenecen al mismo
  - conjunto si y sólo sí están conectados.
  - Si dos vértices u y v están en el mismo conjunto, la arista (u, v) es rechazada porque al aceptarla forma un ciclo.

- Inicialmente cada vértice pertenece a su propio conjunto
    - ➔ |V| conjuntos con un único elemento
- Al aceptar una arista se realiza la Unión de dos conjuntos
- Las aristas se organizan en una heap, para ir recuperando la de mínimo costo en cada paso
- Tiempo de ejecución:
    - Se organizan las aristas en una heap, para optimizar la recuperación de la arista de mínimo costo
    - El tamaño de la heap es |E|, y extraer cada arista lleva O (log |E|)
    - El tiempo de ejecución es O (|E |log|E|)
    - Dado que |E| ≤ |V|2, log |E| ≤ 2 log |V|,
        - ➔ el costo total del algoritmo es O (|E |log|V|)