# Análisis y Diseño de Algoritmos.

**Sesión 8.** 9 de Octubre de 2014.

Maestría en Sistemas Computacionales.

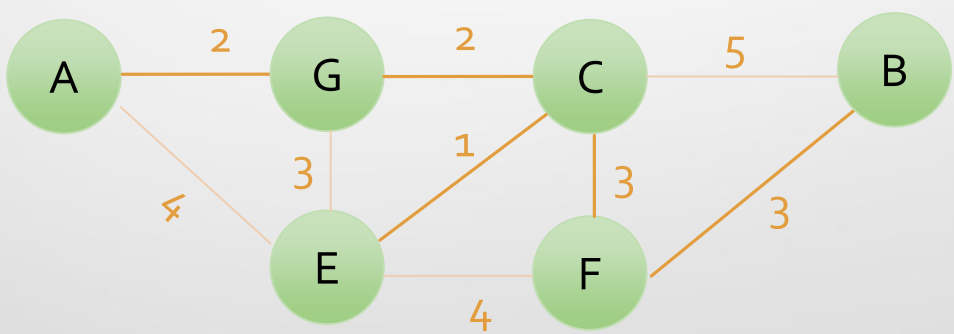
Por: Hugo Iván Piza Dávila.

# Árbol recubridor mínimo

- Un árbol recubridor de un grafo conexo ponderado G es un subgrafo acíclico que une todos los vértices de G.
- Pueden existir muchos árboles recubridores de G.
- Un árbol recubridor mínimo (ARM) es aquél cuya suma de los pesos de sus aristas es la menor (Minimum Spanning Tree).
- A una compañía de cable le interesaría obtener un ARM que conecte a todos sus clientes con el mínimo uso de cobre.
- Minimizar el peso total considerando la mejor opción conocida localmente: algoritmo voraz.

# Árbol recubridor mínimo

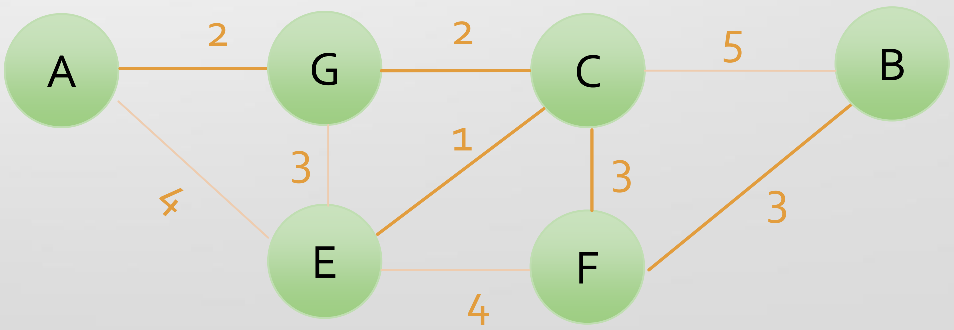
- En el ejemplo, el ARM está compuesto por las aristas: AG, GC, CE, CF, FB



# Algoritmo de Prim

- Sea $n_1$ el nodo de partida
- Sea ARM = $\{n_1\}$ un conjunto de nodos que ya forman parte del *árbol recubridor mínimo*.
- Desde $k$ = 2 hasta el número de nodos del grafo, hacer:
  - Encontrar el nodo $n_k$ que esté unido a ARM con el menor peso
  - Ojo: $n_k \notin$ ARM, $n_k$ es vecino de algún nodo $n \in$ ARM
  - Agregar $n_k$ a ARM

1) ARM = {A}
2) ARM = {A, G}
   a) Peso(AE) = 4
   b) Peso(AG) = 2
3) ARM = {A, G, C}
   a) Peso(AE) = 4
   b) Peso(GC) = 2
   c) Peso(GE) = 3

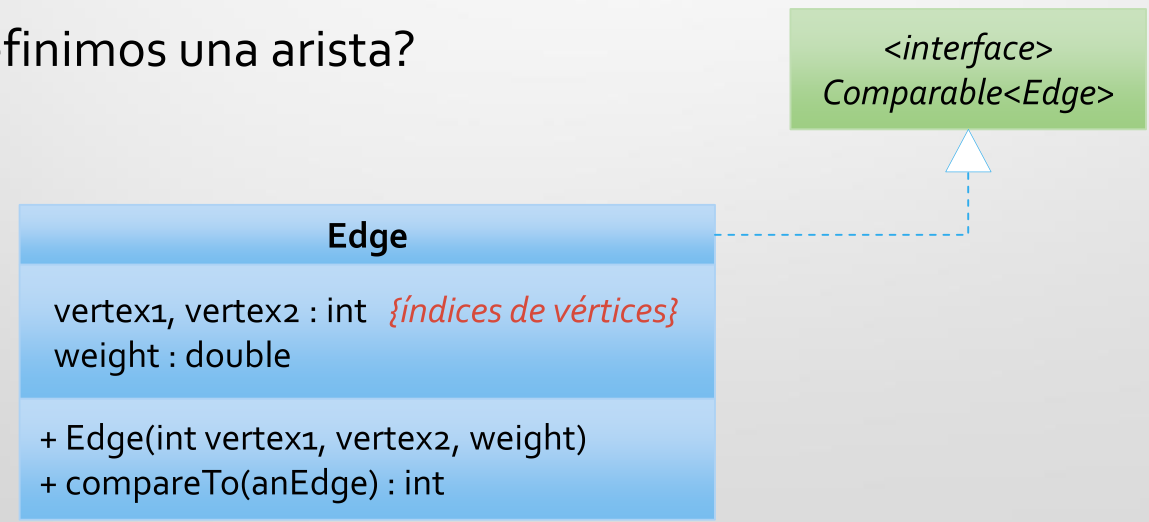


# Implementación de Prim

- Estructuras de datos a utilizar:
  1. Grafo definido por una matriz de adyacencia de números reales.
     - Para aristas inexistentes almacenaremos el máximo valor posible.
  2. Una lista con capacidad para N – 1 aristas que almacenará la solución: el árbol recubridor mínimo (ARM).
  3. Un arreglo que determine si cada vértice ya fue visitado o no.
     - Ya forma parte del ARM una arista que sale del vértice visitado.
  4. Una cola de prioridad que almacene aristas candidatas a formar parte del árbol recubridor mínimo.
     - Va a contener sólo aquellas que sean vecinas de algún nodo ya visitado, es decir, deben ser aristas vecinas del ARM.
     - Las aristas estarán siempre ordenadas por su peso.

# Implementación de Prim

- La lista y la cola almacenan aristas.
- La cola exige que la arista pueda compararse con otra, utilizando el peso, para poderse ordenar.
- ¿Cómo definimos una arista?



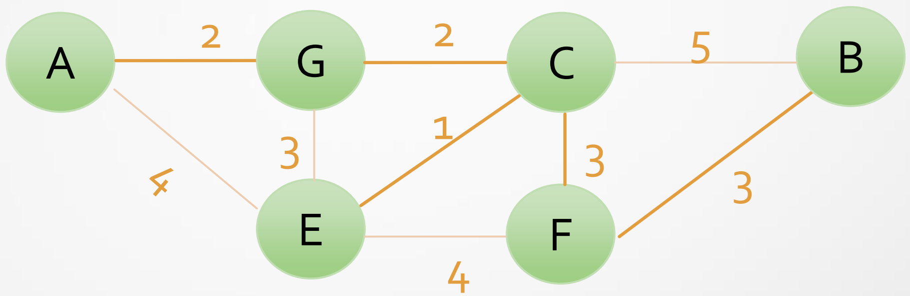
# Implementación de Prim

1. El nodo de inicio será el $0$: marcarlo como visitado.
2. Añadir a la cola de candidatos todas las aristas que partan del nodo de inicio (el peso es menor al máximo posible).
3. Repetir mientras el ARM sea menor a $N - 1$:
   a) Sacar la arista al frente de la cola (la que tiene menor peso).
   b) Si la arista conecta a un nodo del ARM con uno fuera del ARM:
      i. Añadir la arista a la solución.
      ii. Marcar como visitado al nodo que estaba fuera del ARM.
      iii. Añadir todas las aristas que parten del nodo nuevo a la cola de candidatos.

# Algoritmo de Kruskal

1. Se crea un conjunto *E* que almacena todas las aristas del grafo
2. Se crea un bosque *F* (un conjunto de árboles), donde cada nodo del grafo es un árbol separado
3. Mientras *F* tenga más de un árbol,
   a. Eliminar una arista *a* de peso mínimo de *E*,
   b. Si *a* conecta dos árboles diferentes de *F*, se combinan los dos árboles en uno sólo, y se añade la arista *a* al MST
   c. En caso contrario, se desecha *a*
4. Al finalizar el algoritmo, *F* tiene un solo componente, y MST tendrá todas las aristas que forman el árbol de expansión mínimo

# Algoritmo de Kruskal




Ignorar el orden de los elementos
en E y F de este ejemplo

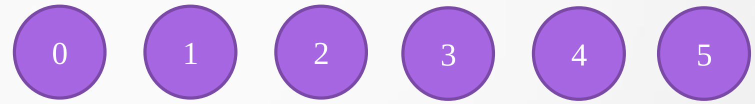
- F = { {A}, {G}, {E}, {C}, {F}, {B} }, MST = {}
- E = { (A, E, 4), (A, G, 2), (G, E, 3), (G, C, 2), (C, E, 1), (E, F, 4), (C, F, 3), (C, B, 5), (B, F, 3)}
- Mínimo = (C, E, 1)
- F = { {A}, {G}, {E, C}, {F}, {B} }, MST = { (C, E, 1) }
- E = { (A, E, 4), (A, G, 2), (G, E, 3), (G, C, 2), (E, F, 4), (C, F, 3), (C, B, 5), (B, F, 3)}
- Mínimo = (A, G, 2)
- F = { {A, G}, {E, C}, {F}, {B} }, MST = { (C, E, 1), (A, G, 2) }
- E = { (A, E, 4), (G, E, 3), (G, C, 2), (E, F, 4), (C, F, 3), (C, B, 5), (B, F, 3)}

# Implementación de Kruskal

- Se crea un conjunto *E* que almacena todas las aristas del grafo
  - Se utiliza la estructura Edge previamente creada.
  - Se crea una arista por cada celda (i, j) de la matriz de adyacencia ≠ MAX.
  - Como es un grafo no dirigido, si se agrega (i, j), no agregar (j, i).
  - Añadir cada arista a una cola de prioridad que ordena de menor a mayor.
- ¿Cómo creamos y actualizamos el bosque F?
  - Utilizamos la técnica *union-find* para unir conjuntos disjuntos.
  - El sello de esta técnica es contar con un arreglo que almacena en la posición *k* el vértice ***padre*** del vértice *k*, en un árbol simulado.
  - El ***padre*** es el que agregó al vértice *k* (y sus descendientes) a su grupo.
  - Utiliza un valor ***rank*** para determinar cuál vértice agrega al otro.
  - Si el ***padre*** es igual a *k*, ese vértice es la raíz del árbol o el **líder del grupo**.
  - Cada arista (i, j) que une dos conjuntos disjuntos se añade al ARM.

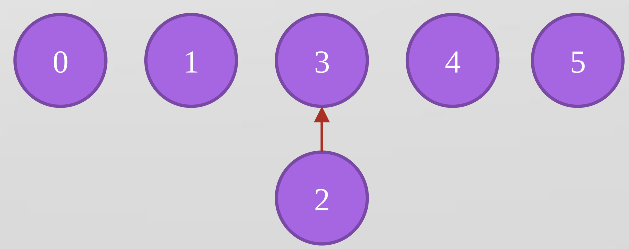
# Técnica *union-find*

- F = { {0}, {1}, {2}, {3}, {4}, {5} }
- Parents = {0, 1, 2, 3, 4, 5}
- Rank = {1, 1, 1, 1, 1, 1}



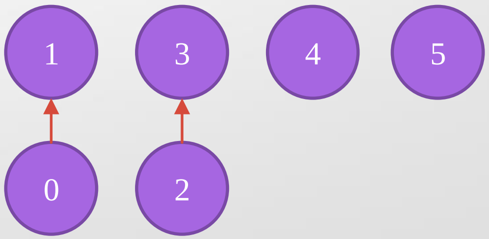

Rank[k] = tamaño del árbol con raíz en *k*.

- Arista elegida = (2, 3, 1).
- Unir vértices (2, 3)
  - Líder(2) = 2 ≠ Líder(3) = 3 Pertenecen a diferentes grupos (diferentes líderes).
  - Rank[2] = Rank[3] = 1 Mismo tamaño.
  - Parents[2] = 3 El vértice 3 integra al 2 a su grupo (3 > 2).
  - Rank[3] = Rank[3] + 1 = 2 Al tamaño del grupo 3 se le sumará el del grupo 2.
- Parents = {0, 1, 3, 3, 4, 5}
- Rank = {1, 1, 1, 2, 1, 1}



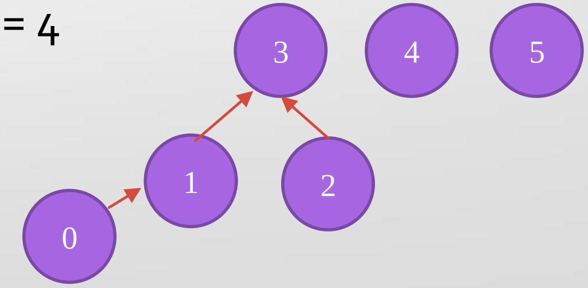
# Técnica *union-find*

- Arista elegida = (0, 1, 2).
  - Líder(0) = 0 ≠ Líder(1) = 1 Pertenecen a diferentes grupos.
  - Rank[0] = Rank[1] = 1 Mismo tamaño: el vértice 1 integra al 0 a su grupo.
  - Parents[0] = 1
  - Rank(1) = Rank(1) + Rank(0) = 1 + 1 = 2
- Parents = {1, 1, 3, 3, 4, 5}
- Rank = {1, 2, 1, 2, 1, 1}



# Técnica *union-find*

- Arista elegida = (1, 2, 2).
  - Líder(1) = 1 ≠ Líder(2) = 3 Pertenecen a diferentes grupos.
  - Rank[1] = Rank[3] = 2 Mismo tamaño: el vértice 3 integra al vértice 1 a su grupo.
  - Parents[1] = 3
  - Rank(3) = Rank(3) + Rank(1) = 2 + 2 = 4
- Parents = {1, 3, 3, 3, 4, 5}
- Rank = {1, 2, 1, 4, 1, 1}



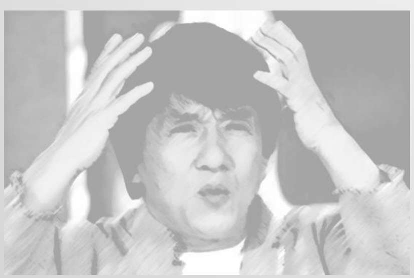
# Técnica *union-find*

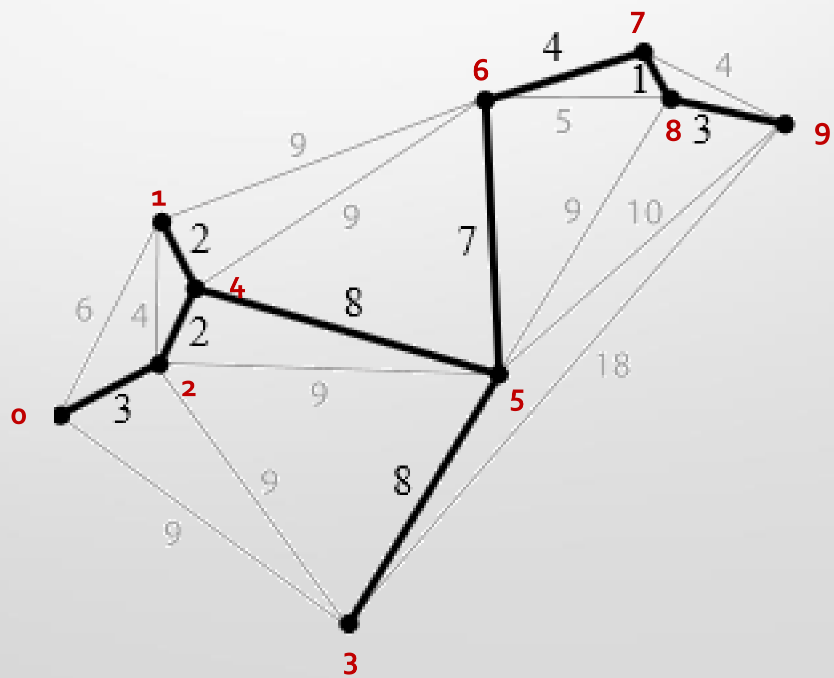
- Métodos adicionales a implementar:
- `int findLeader(int Parents[], int x)`
  - Devuelve el índice del líder del grupo al que pertenece el vértice x.
  - Recibe el arreglo que almacena el padre directo de cada vértice.
  - Se puede implementar de forma recursiva.
- `boolean join(int parents[], int rank[], int i, int j)`
  - Lleva a cabo la unión de los grupos a los que pertenecen los vértices i, j.
  - Si ambos vértices tienen el mismo líder no efectúa la unión y devuelve *falso*.
  - En caso contrario,
    - Sean líder1, líder2, respectivamente, los líderes con menor y mayor rank.
    - El padre de líder1 será lider2
    - Al rank de líder2 se le sumará el rank de líder1.

- El algoritimo de **kruskal** deberá agregar las primeras N – 1 aristas al ARM tal que el método **join** devuelva verdadero.

# Ejercicio

- Probar que funcionan los algoritmos de Prim y Kruskal con el siguiente grafo:
  - `Prim: (0,2), (2,4), (4,1), (4,5), (5,6), (6,7), (7,8), (8,9), (5,3)`
  - `Kruskal: (7,8), (1,4), (2,4), (0,2), (8,9), (6,7), (5,6), (3,5), (4,5)`



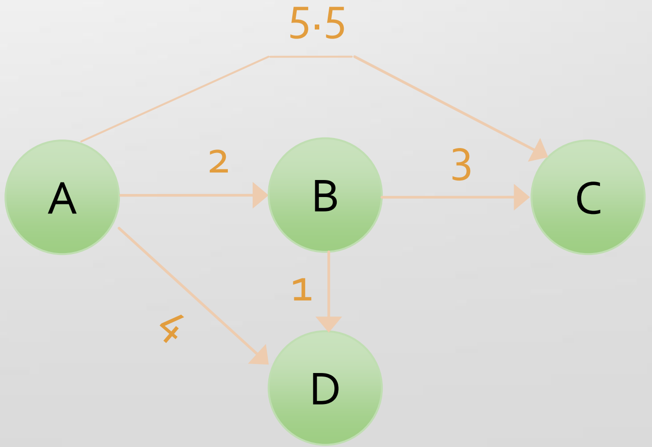


# Algoritmo de Dijkstra

- El algoritmo de Dijkstra calcula la distancia más corta entre un nodo *n* y todos los demás nodos del grafo.
- Puede utilizar búsqueda en profundidad o en anchura.
- Normalmente, el grafo es dirigido y ponderado (Red).

Nodo inicial = A

| A | 0.0 |
|---|---|
| B | 2.0 |
| C | 5.0 |
| D | 3.0 |



# Algoritmo de Dijkstra

- Como búsqueda en profundidad, tendrá una lista de nodos visitados y una pila de nodos por visitar.
- Además, tendrá una lista de la distancia más corta del nodo inicial a cada nodo visitado.
    - `distancias[i]` almacena la distancia más corta encontrada al momento entre el nodo inicial y el nodo visitado número *i*.
- Y una pila de distancias desde el nodo inicial al nodo que se encuentra en la pila.
    - Por cada vecino introducido a la pila de nodos, se introduce también su distancia desde el nodo inicial a la pila de distancias.

# Algoritmo de Dijkstra

1. Meter a la pila el nodo inicial.
2. Meter 0.0 a la pila de distancias.
3. Mientras la pila (que sea) no esté vacía:
   a) Sean: ***n*** el último nodo de la pila, y ***d*** la última distancia de la pila
   b) Si ***n*** ya se visitó y su distancia al nodo inicial registrada es ≤ ***d***, regresar al paso 3 (no se mejoró el camino: no hay nada que actualizar)
   c) Si ***n*** ya se visitó y su distancia registrada es mayor que ***d***, guardar ***d*** en ***distancias*** en la posición en la que ***n*** está en ***nodosVisitados***
   d) Si ***n*** no se ha visitado, agregar ***n*** a ***nodosVisitados***, y ***d*** a ***distancias***
   d) Meter cada vecino $v_k$ de ***n*** a la pila de nodos
   e) Meter cada peso $p_k$ + ***d*** a la pila de distancias

# Algoritmo de Dijkstra

**Pilas**

| Nodos | Distancias |
|---|---|
| **A** | **0.0** |

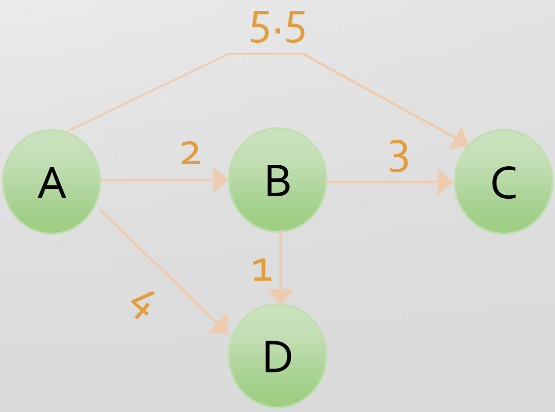
A no ha sido visitado.
Agregamos A, 0.0 a las listas respectivas.
Agregamos sus vecinos y pesos a las pilas.

**Nodos visitados** | |
**Distancias** | |

| **Nodos visitados** | **A** |
|---|---|
| **Distancias** | **0.0** |

| Nodos | Distancias |
|---|---|
| **D** | **4.0** |
| C | 5.5 |
| B | 2.0 |

| **Nodos visitados** | **A** | **D** |
|---|---|---|
| **Distancias** | **0.0** | **4.0** |




D no ha sido visitado.
Agregamos D, 4.0 a las listas respectivas.
D no tiene vecinos.

# Algoritmo de Dijkstra

| Nodos | Distancias |
|---|---|
| **C** | **5.5** |
| B | 2.0 |

**Nodos visitados**

| A | D | C |
|---|---|---|

**Distancias**

| 0.0 | 4.0 | 5.5 |
|---|---|---|

C no ha sido visitado.
Agregamos C, 5.5 a las listas respectivas. C No tiene vecinos.

| Nodos | Distancias |
|---|---|
| **B** | **2.0** |

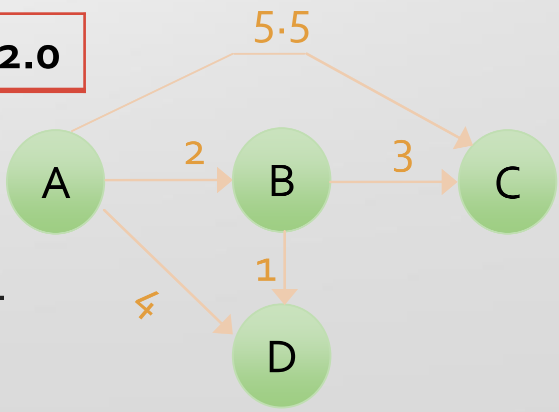
**Nodos visitados**

| A | D | C | B |
|---|---|---|---|

**Distancias**

| 0.0 | 4.0 | 5.5 | 2.0 |
|---|---|---|---|

B no ha sido visitado.
Agregamos B, 2.0 a las listas respectivas.
Agregamos sus vecinos (C, D) y pesos (2 + 3, 2 + 1) a las pilas.



# Algoritmo de Dijkstra

| Nodos | Distancias |
|---|---|
| D | 3.0 |
| C | 5.0 |

**Nodos visitados**

| A | D | C | B |
|---|---|---|---|

**Distancias**

| 0.0 | 3.0 | 5.5 | 2.0 |
|---|---|---|---|

D ya ha sido visitado. Su distancia registrada (4) es mayor que la actual (3). Reemplazamos 4.0 por 3.0 en la lista de distancias. D no tiene vecinos.

| Nodos | Distancias |
|---|---|
| C | 5.0 |

**Nodos visitados**

| A | D | C | B |
|---|---|---|---|

**Distancias**

| 0.0 | 3.0 | 5.0 | 2.0 |
|---|---|---|---|

C ya ha sido visitado.
Su distancia registrada (5.5) es mayor que la actual (5.0).
Reemplazamos 5.5 por 5.0 en la lista de distancias.
C no tiene vecinos.