CURSO DE PROGRAMACIÓN COMPETITIVA
URJC - 2021

Sesión 5 (8ª Semana)

- David Morán (david.moran@urjc.es)
- Sergio Pérez (sergio.perez.pelo@urjc.es)
- Jesús Sánchez-Oro (jesus.sanchezoro@urjc.es)
- Isaac Lozano (isaac.lozano@urjc.es)
- Raúl Martín (raul.martin@urjc.es)
- Jakub Jan (jakubjanluczyn@gmail.com)
- Antonio Gonzalez (antonio.gpardo@urjc.es)
- Iván Martín (ivan.martin@urjc.es)
- Leonardo Antonio Santella (leocaracas2010@gmail.com)

Contenidos

- Grafos (Ponderados)
  - Introducción y algoritmos de camino más corto
    1. Floyd Warshall
    2. Colas de Prioridad
    3. Dijkstra
  - Árboles de recubrimiento
    1. Prim
    2. Union-Find
    3. Kruskal

Grafos Ponderados

- En la vida real, el camino más corto entre dos puntos no es siempre el número de aristas por las que se pasa de un vértice i a un j cualquiera
- Los grafos muchas veces son ponderados

Grafos Ponderados

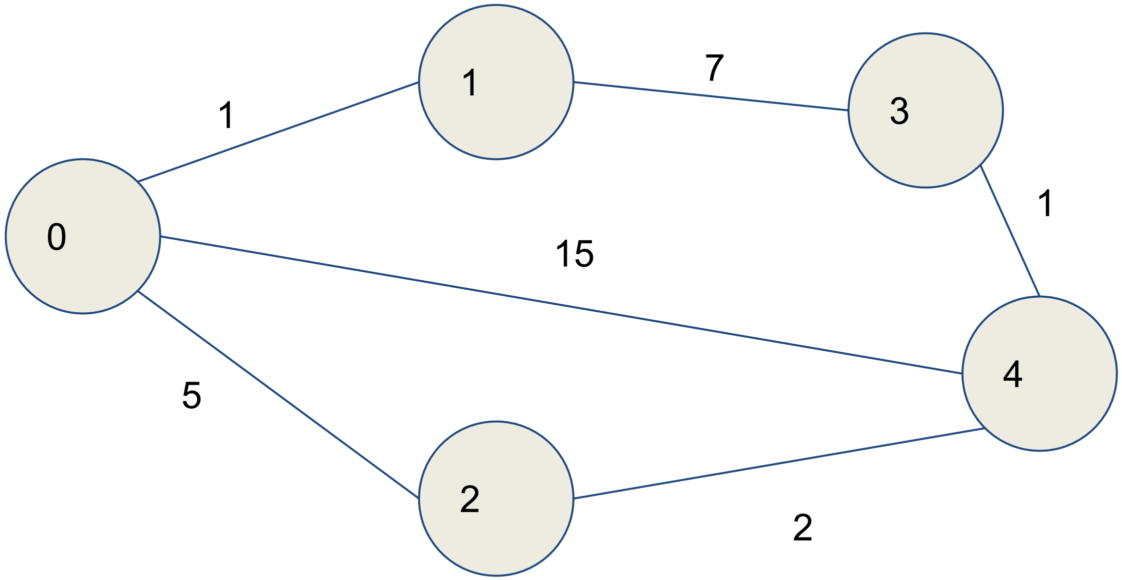
- Para representar que de u a v existe una arista con peso w
- Matriz de adyacencia:
  - mat[u][v] = w
- Lista de adyacencia:
  - edges.push(edge(u, v, w)) // clase edge

Grafos Ponderados

```
class edge{
   from, to, weight
   edge(f, t, w):
      from = f
      to = t
      weight = w
}
arraylist<edge> edges[N]
```

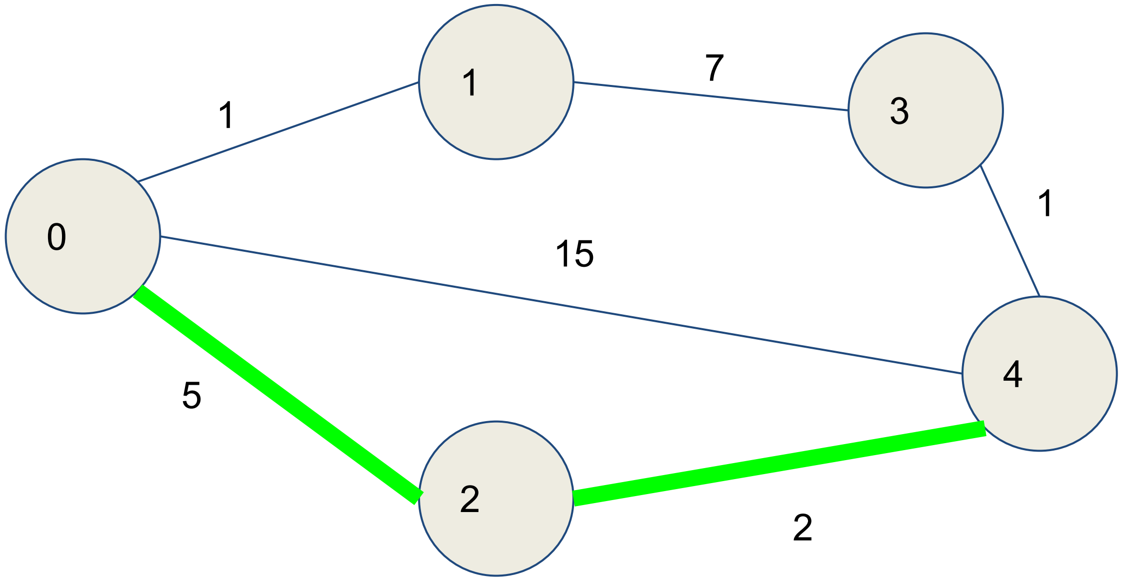
Grafos Ponderados

- ¿Cuál es el camino más corto del siguiente grafo para ir de 0 a 4?



Grafos Ponderados

- ¿Cuál es el camino más corto del siguiente grafo para ir de 0 a 4?



Grafos Ponderados

- Algoritmo de Floyd Warshall
- Compara todos los puntos para ir de cualquier nodo i a cualquier nodo j considerando un posible tercer nodo k
- El mínimo (*) entre todos estos objetivos siempre dará como resultado el camino mínimo entre cualquier par de nodos i,j dentro del grafo

Grafos Ponderados

- Algoritmo de Floyd Warshall
- (Contra) Es N^3 por lo que su utilización está bastante limitada
- (Pro) Funciona sobre cualquier par de vértices
- (Pro) es muy fácil de escribir
- (Pro) suele actuar sobre matrices de adyacencia, más fácil de implementar

Grafos Ponderados

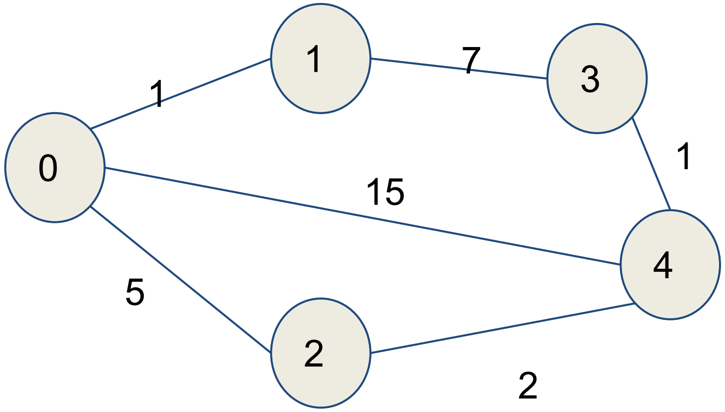
- Algoritmo de Floyd Warshall

```
for k from 0..N
  for i from 0..N
    for j from 0..N
      graph[i][j] = min(
graph[i][j], graph[i][k] +
graph[k][j])
```

Grafos Ponderados

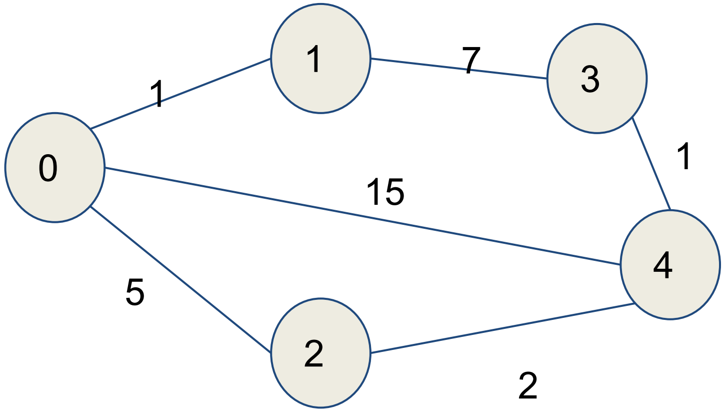
- Algoritmo de Floyd Warshall
- Por cada par de vértices en la matriz de adyacencia i,j veremos si es mejor el camino actual que tenemos ó hacer un camino nuevo pasando por cualquier otro nodo k entre medias
- Si no existe un camino entre i,j; graph[i][j] = INF

Grafos Ponderados



0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

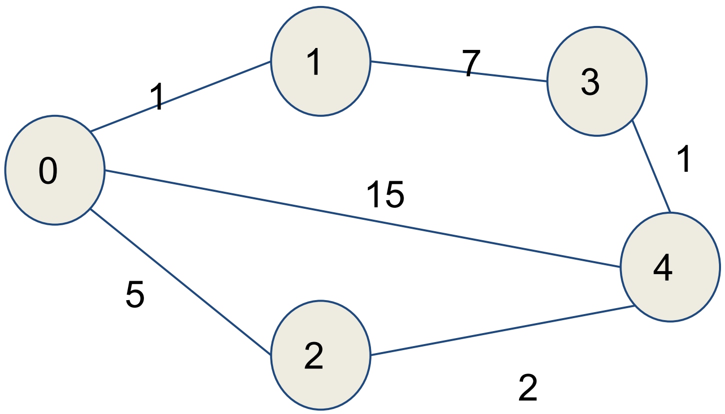
Grafos Ponderados



MIN({0,4},
{0,2}+{2,4})
MIN(15, 5+2) = 7

0	1	5	INF	15 (7)
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15 (7)	INF	2	1	0

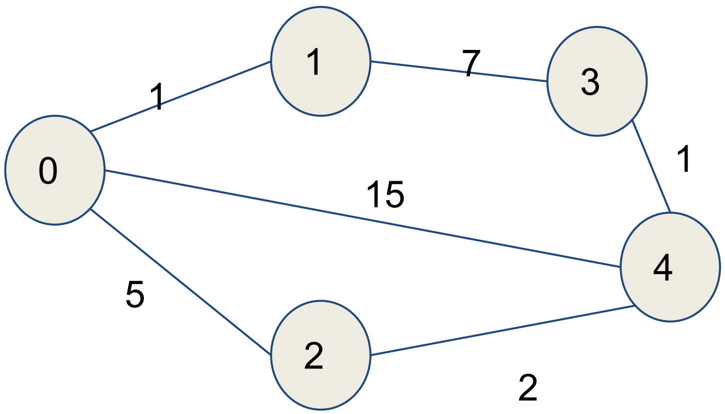
Grafos Ponderados



MIN({0,3},
{0,1}+{1,3})
MIN(INF, 1+7) = 8

0	1	5	INF (8)	15 (7)
1	0	INF	7	INF
5	INF	0	INF	2
INF (8)	7	INF	0	1
15 (7)	INF	2	1	0

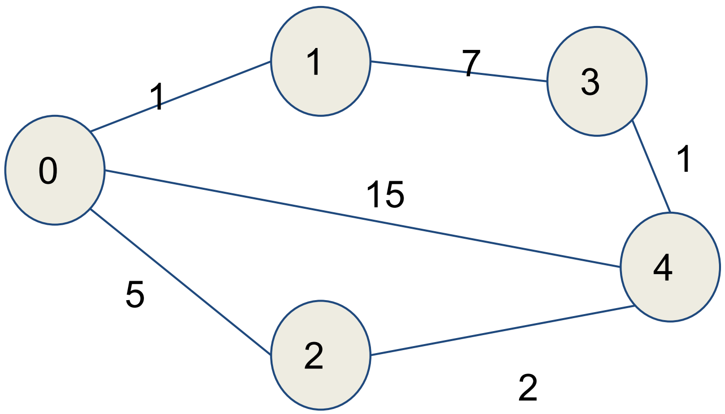
Grafos Ponderados



MIN({0,4},
{0,3}+{3,4})
MIN(7, 8+1) = 7

0	1	5	INF (8)	15 (**7**)
1	0	INF	7	INF
5	INF	0	INF	2
INF (8)	7	INF	0	1
15 (**7**)	INF	2	1	0

Grafos Ponderados



MUCHAS ITERACIONES DESPUÉS...

0	1	5	8	7
1	0	6	7	8
5	6	0	3	2
8	7	3	0	1
7	8	2	1	0

Grafos Ponderados

- Heaps
  - Estructura de datos en forma de “árbol binario”
  - Balanceado de tal manera que para cualquier nodo v, parent(v) > v y children(v) < v

Grafos Ponderados

- Heaps
  - Útil si necesitamos llevar cuenta de forma muy rápida sobre el mínimo o el máximo de una estructura
  - Inserción y borrado en O(LgN)
  - Búsqueda de min-max en O(1) (ver la raíz del árbol)

Grafos Ponderados

- Algoritmo de Dijkstra
- Dado un nodo inicio y un nodo fin, computa en O(VLg(E)) la distancia más corta entre esos puntos
- Algoritmo voraz
- No computa la distancia de todos los pares de nodo
- Computa la distancia de i hacia todos

Grafos Ponderados

- Algoritmo de Dijkstra
- Guarda en un array de distancias todas las distancias desde i hacia el resto
- La implementación es muy parecida a un BFS
- Es casi igual al algoritmo de Prim

Grafos Ponderados

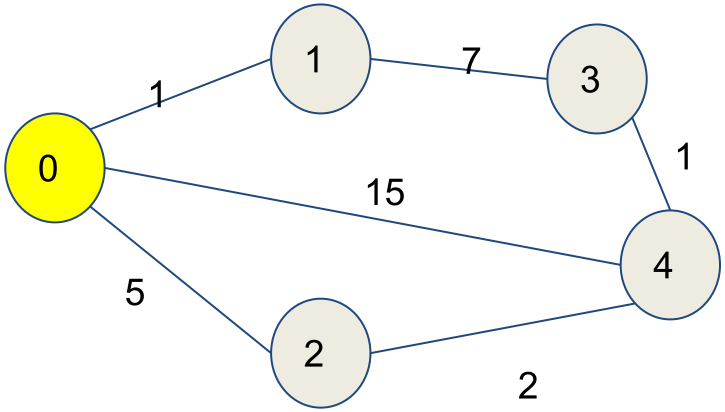
- Algoritmo de Dijkstra

```
function dijkstra(start, end)
  array_dist(start) = 0
  pq.encolar({start, 0})
  mientras !pq.vacio()
    n = pq.desencolar()
    si array_dist[n.node] >= n.dist
      para cada arista D en n.node
        peso = n.dist + D.weight
        si array_dist[D.to] > peso
          pq.encolar({ D.to, peso })
          array_dist[D.to] = peso
  retornar array_dist[end]
```

Grafos Ponderados

- Algoritmo de Dijkstra
- Encolamos el primer nodo con distancia 0
- Tenemos una cola de prioridad ordenando de menor a mayor según la sumatoria del peso de los nodos
- Si la distancia hasta ese nodo es menor o igual, descartamos, si no, encolamos
- Retornamos al final la distancia hasta end ó, si en algún momento de la cola nos topamos con end podemos retornar directamente (es un voraz, no conseguiremos nada mejor)

Grafos Ponderados

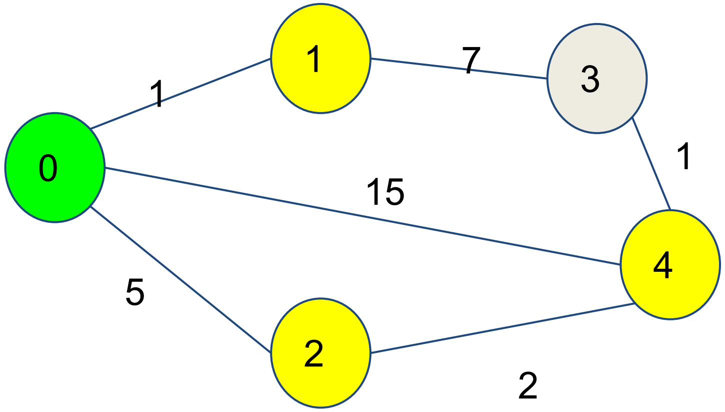


$PQ = \{ \{0, 0\} \}$

Dist = { 0, INF, INF, INF, INF}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

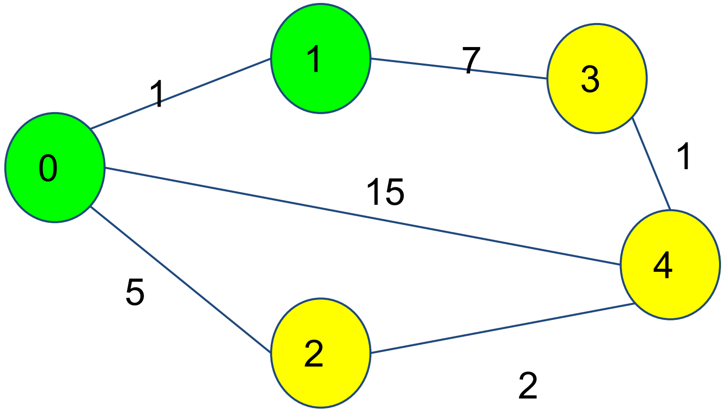


PQ = { {1, 1}, {2, 5}, {4, 15} }

Dist = { 0, 1, 5, INF, 15}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

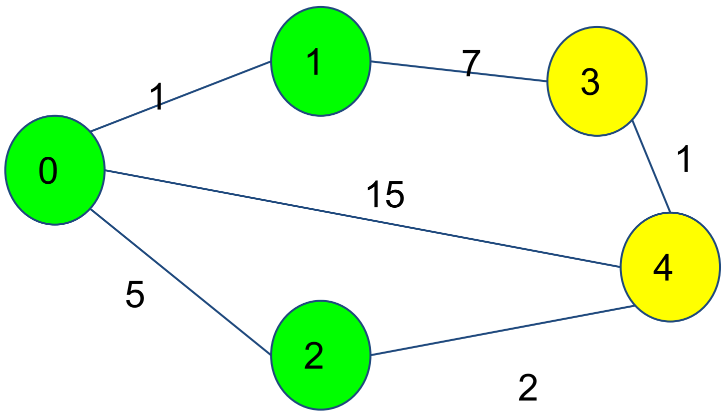


PQ = { {2, 5}, {3, 8}, {4, 15} }

Dist = { 0, 1, 5, 8, 15}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

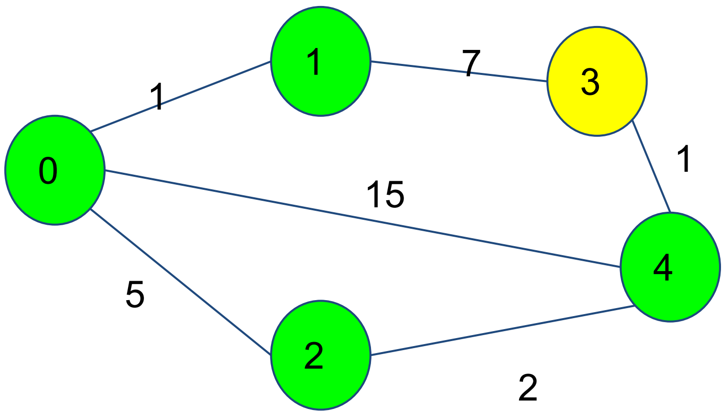


PQ = { {4, 7}, {3, 8}, {4, 15} }

Dist = { 0, 1, 5, 8, 7}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

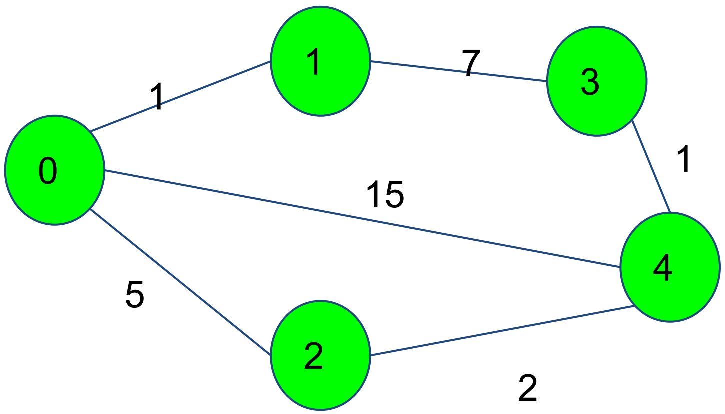


PQ = { {3, 8}, {4, 15} }

Dist = { 0, 1, 5, 8, 7}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados



PQ = { {4, 15} }

Dist = { 0, 1, 5, 8, 7}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

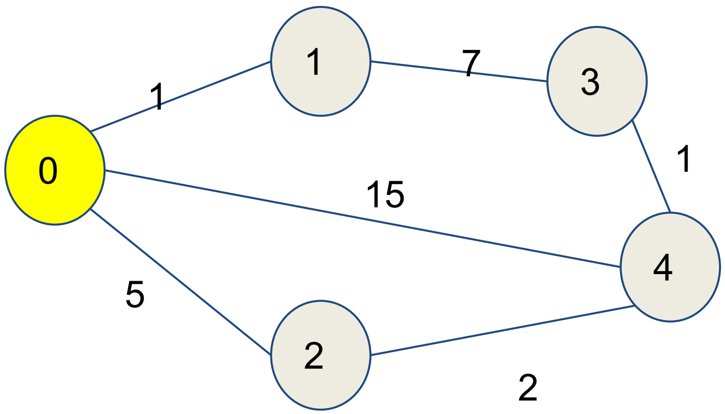
Grafos Ponderados

- Árbol de recubrimiento mínimo
- Dado un grafo cualquiera (conexo), encontrar un árbol dentro del grafo tal que la suma de sus aristas sea **mínima**.
  - Algoritmo de Prim (usando heaps)
  - Algoritmo de Kruskal (usando Union-Find)

Grafos Ponderados

- Algoritmo de Prim para árboles de recubrimiento
- Llevar cuenta de distancias y de visitados
- Si se llega al nodo v con menor distancia, encolar
- Asumimos (por el heap) que al ir del nodo **u** a otro nodo **v**, **u** es parte del árbol final de recubrimiento
- Una vez todos los nodos de 1 hasta N estén visitados, paramos y tendremos el árbol mínimo de recubrimiento completado

Grafos Ponderados

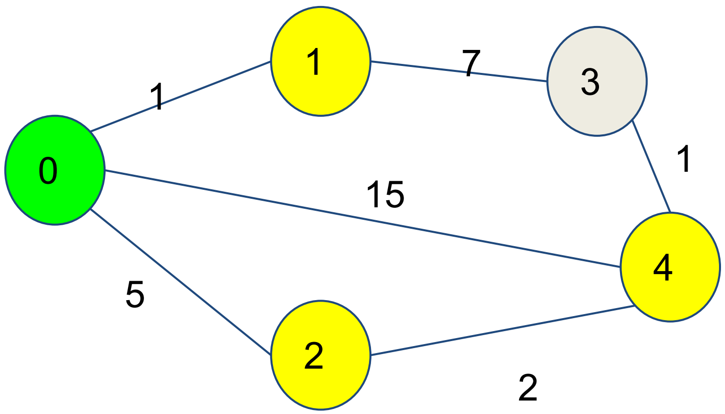


PQ = { {0, 0} }

Dist = { 0, INF, INF, INF, INF}
Visit = { 0, 0, 0, 0, 0}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

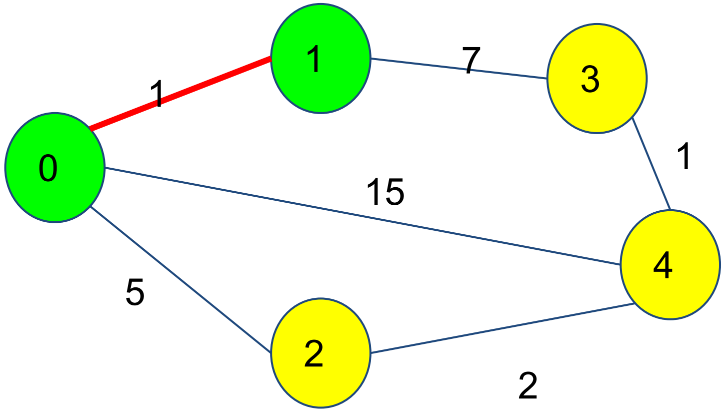


PQ = { {1, 1}, {2, 5}, {4, 15} }

Dist = { 0, 1, 5, INF, 15}
Visit = { 1, 0, 0, 0, 0}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

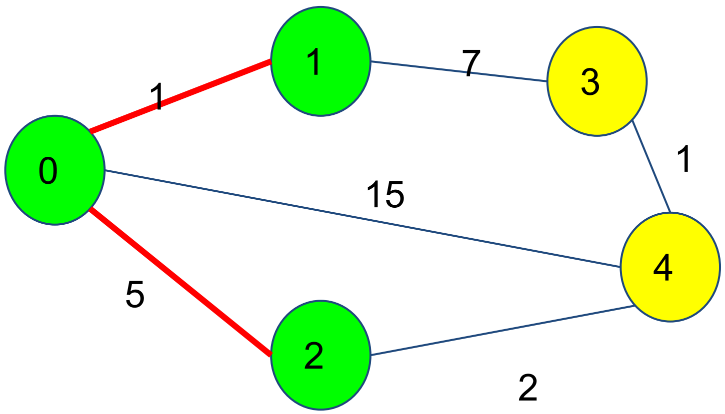


PQ = { {2, 5}, {3, 7}, {4, 15} }

Dist = { 0, 1, 5, 7, 15}
Visit = { 1, 1, 0, 0, 0}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

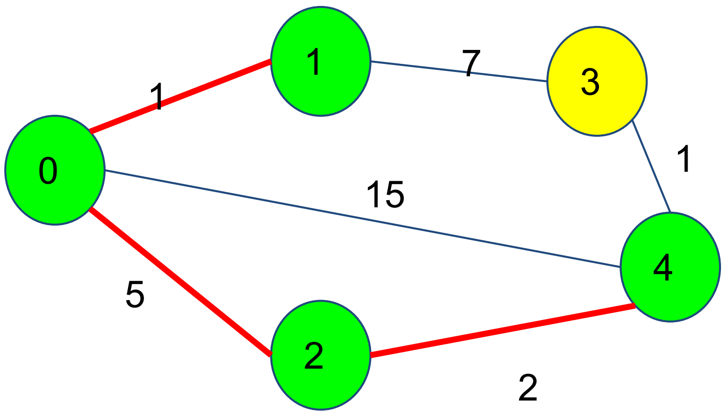


PQ = { {4, 2}, {3, 7}, {4, 15} }

Dist = { 0, 1, 5, 7, 7}
Visit = { 1, 1, 1, 0, 0}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

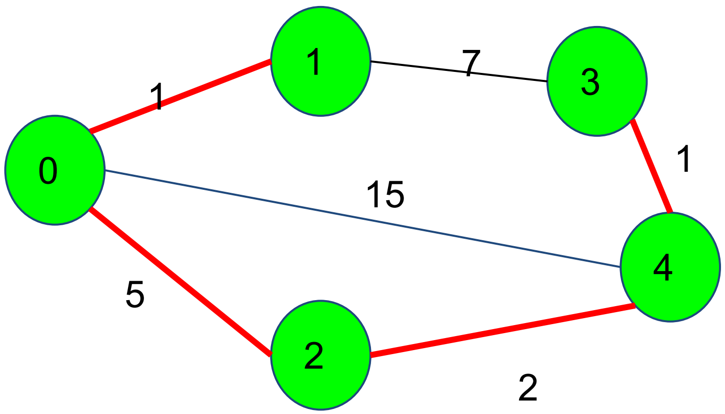


PQ = { {3, 1}, {3, 7}, {4, 15} }

Dist = { 0, 1, 5, 7, 7}
Visit = { 1, 1, 1, 1, 0}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados

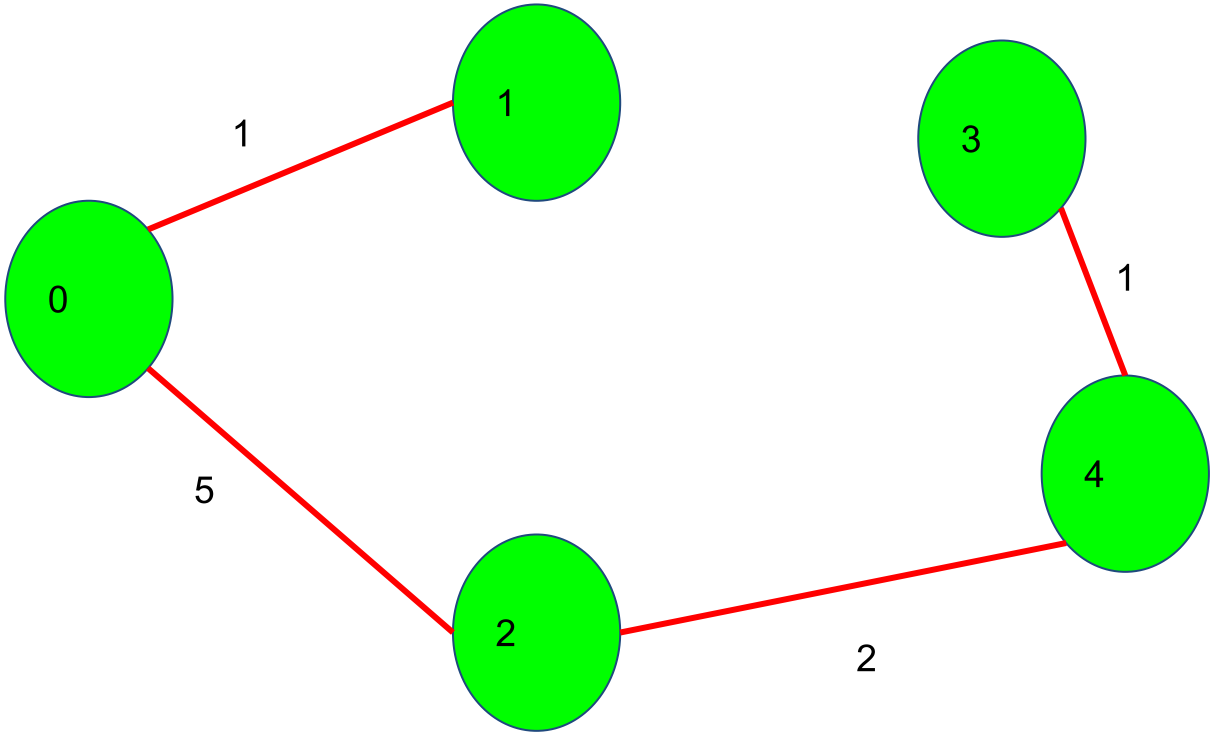


PQ = { {3, 7}, {4, 15} }

Dist = { 0, 1, 5, 7, 7}
Visit = { 1, 1, 1, 1, 1}

0	1	5	INF	15
1	0	INF	7	INF
5	INF	0	INF	2
INF	7	INF	0	1
15	INF	2	1	0

Grafos Ponderados



Grafos Ponderados

- Estructura Union-Find
- Conjuntos disjuntos (partimos de que todos los conjuntos posibles están separados)
- Todos los conjuntos son árboles
- Unimos dos árboles para hacer uno más
- Si dos nodos tienen la misma raíz, se considera que están juntos
- Operaciones básicas:
  - Buscar
  - Unir

Grafos Ponderados

```
UF_init(n):
  parents[n]
  tree_size[n]
  for i=1 to n
    parents[i] = i // no tiene padre
    tree_size[i] = 1 // es un solo nodo
```

Grafos Ponderados

```
UF_find(u):
  while(u != parents[u]):
    u = parents[u]
  return u
```

Grafos Ponderados

```
UF_connect(u, v):
  root_u = UF_find(u)
  root_v = UF_find(v)
  if root_u == root_v:
    return ALREADY_CONNECTED
  UF_join(root_u, root_v)
```

Grafos Ponderados

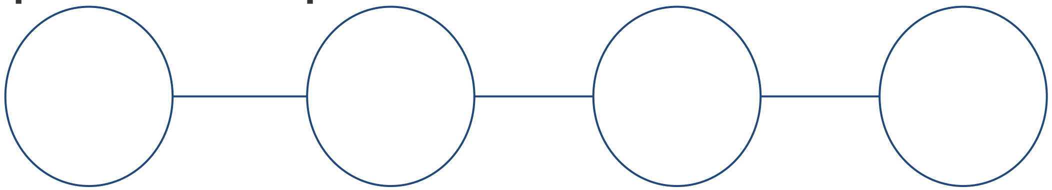
```
UF_join(u, v):
  parents[u] = v
  tree_size[u] += tree_size[v]
```

¿Es esto correcto?

¿Qué pasa en el peor de los casos?

Grafos Ponderados

- En el peor de los casos se forma un árbol degradado, buscar un nodo tendría complejidad **O(N)**!
- Tenemos que buscar otras formas de colocar quien es padre de quien!



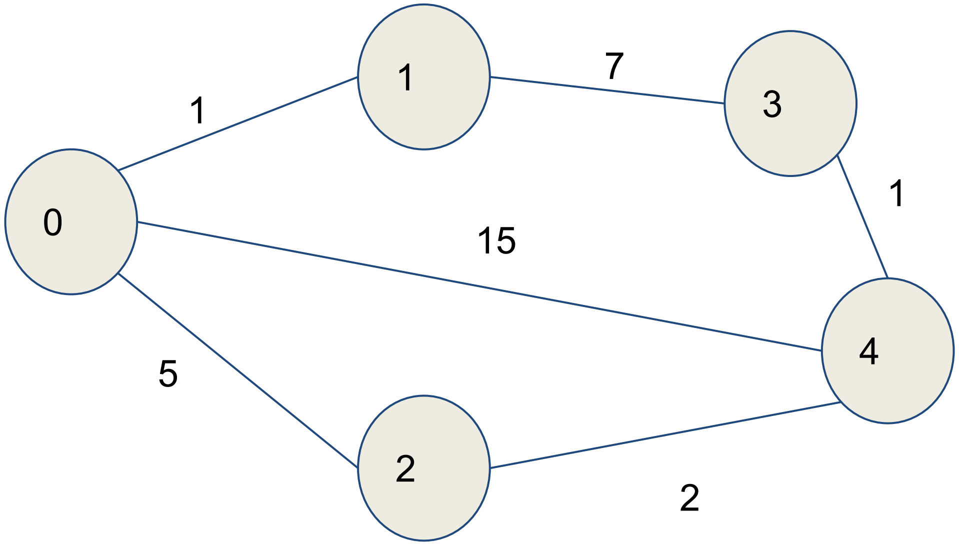
Grafos Ponderados

```
UF_join(u, v):
  if tree_size[u] < tree_size[v]:
    parents[u] = v
    tree_size[v] += tree_size[u]
  else:
    parents[v] = u
    tree_size[u] += tree_size[v]
```

Grafos Ponderados

- Algoritmo de Kruskal
- Partimos de la base perfecta para usar el Union-Find. Todos los nodos están aislados
- Ordenamos una lista de aristas de menor a mayor peso tal que $W_i < W_j$ para todo $j > i$
- Vamos conectando todos los nodos y llevamos cuenta de que nodos se han conectado, si dos nodos se conectan, se produce una arista en el árbol de recubrimiento

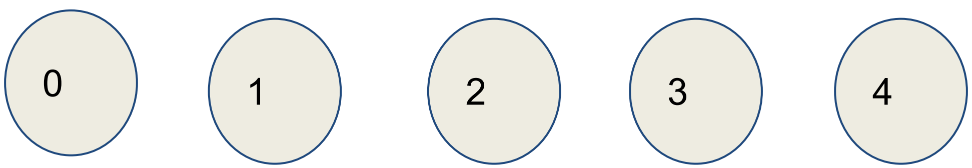
Grafos Ponderados



[{0, 1, 1},
{3, 4, 1},
{2, 4, 2},
{0, 2, 5},
{1, 3, 7},
{0, 4, 15}]

Grafos Ponderados

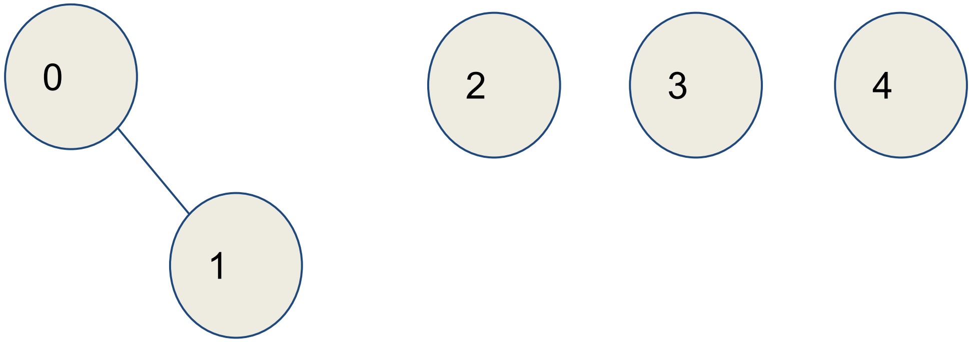
parents = { 0, 1, 2, 3, 4}
tree_size = { 1, 1, 1, 1, 1}
MST = 0



[{0, 1, 1},
{3, 4, 1},
{2, 4, 2},
{0, 2, 5},
{1, 3, 7},
{0, 4, 15}]

Grafos Ponderados

parents = { 0, 0, 2, 3, 4}
tree_size = { 2, 1, 1, 1, 1}
MST = 0 + 1

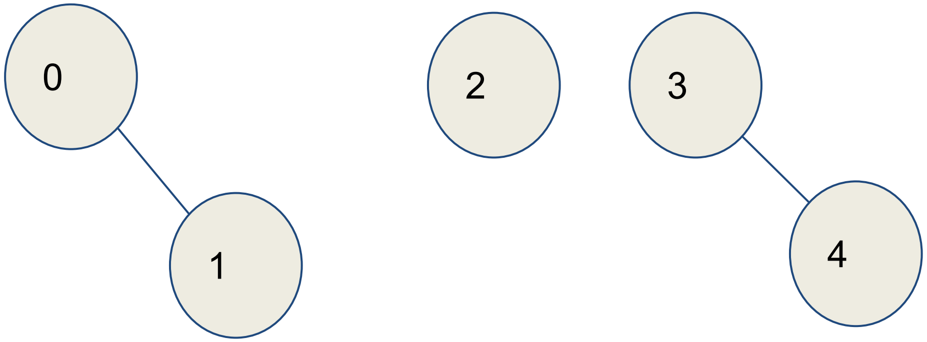


[{0, 1, 1},
{3, 4, 1},
{2, 4, 2},
{0, 2, 5},
{1, 3, 7},
{0, 4, 15}]

Grafos Ponderados

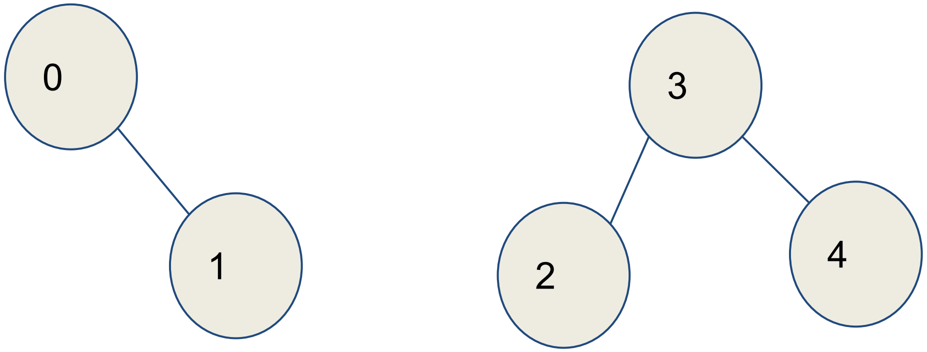
parents = { 0, 0, 2, 3, 3}
tree_size = { 2, 1, 1, 2, 1}
MST = 1 + 1

[{0, 1, 1},
{3, 4, 1},
{2, 4, 2},
{0, 2, 5},
{1, 3, 7},
{0, 4, 15}]



Grafos Ponderados

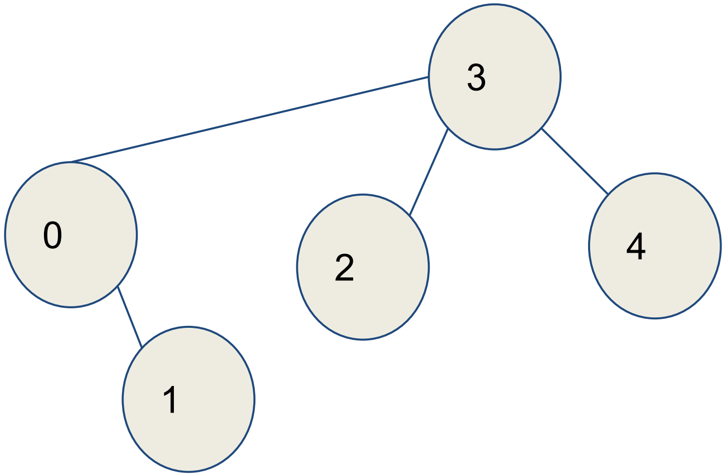
parents = { 0, 0, 3, 3, 3}
tree_size = { 2, 1, 1, 3, 1}
MST = 2 + 2



[{0, 1, 1},
{3, 4, 1},
{2, 4, 2},
{0, 2, 5},
{1, 3, 7},
{0, 4, 15}]

Grafos Ponderados

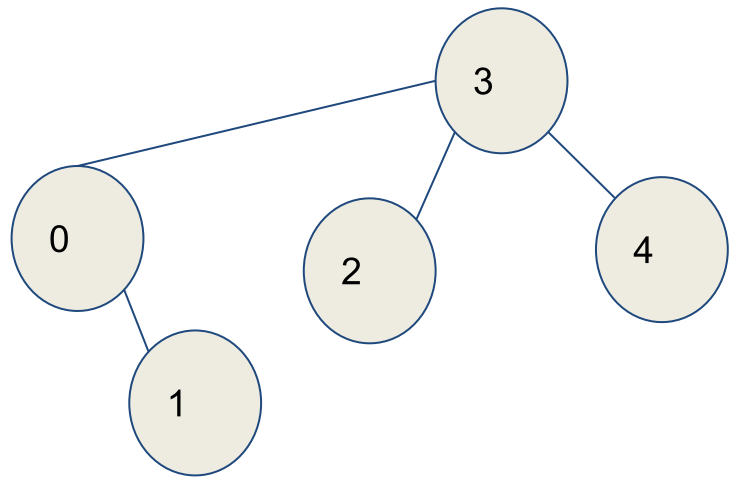
parents = { 3, 0, 3, 3, 3}
tree_size = { 2, 1, 1, 5, 1}
MST = 4 + 5



[{0, 1, 1},
{3, 4, 1},
{2, 4, 2},
{0, 2, 5},
{1, 3, 7},
{0, 4, 15}]

Grafos Ponderados

parents = { 3, 0, 3, 3, 3}
tree_size = { 2, 1, 1, 5, 1}
MST = 9



[{0, 1, 1},
{3, 4, 1},
{2, 4, 2},
{0, 2, 5},
{1, 3, 7},
{0, 4, 15}]

Ejemplo de grafos

Acepta el reto - De aventuras por el amazonas
Problema: 281

Bloque siguiente...

- Bloque siguiente:
  - Programación Dinámica
  - Estructura
  - Memoization
  - Tipos de programación dinámica
  - Problemas ejemplo

¡Hasta la próxima semana!

Ante cualquier duda sobre el curso o sobre los problemas podéis escribirnos (preferiblemente con copia a algunos / todos los docentes)

- David Morán (david.moran@urjc.es)
- Sergio Pérez (sergio.perez.pelo@urjc.es)
- Jesús Sánchez-Oro (jesus.sanchezoro@urjc.es)
- Isaac Lozano (isaac.lozano@urjc.es)
- Raúl Martín (raul.martin@urjc.es)
- Jakub Jan (jakubjanluczyn@gmail.com)
- Antonio Gonzalez (antonio.gpardo@urjc.es)
- Iván Martín (ivan.martin@urjc.es)
- Leonardo Antonio Santella (leocaracas2010@gmail.com)