Estructuras de Datos

Dr. Martin Gonzalez-Rodriguez

ISBN 978-1-365-00694-4

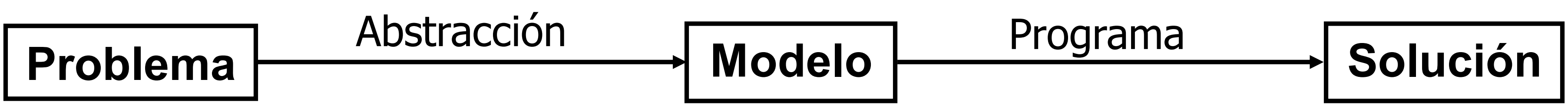
Diseño y Algoritmia

Dr. Martin Gonzalez-Rodriguez

Resolución de Problemas en Ingeniería

Estrategia

- Conocer y acotar el problema (análisis).
- Encontrar un modelo que represente el problema (abstracción).
- Formular el algoritmo sobre el modelo.



Programas

La Frase
`Programas = Estructuras de Datos + Algoritmos`

- Identificar medios para **almacenar datos** y diseñar **algoritmos que resuelvan la tarea** asignada a los procesos.

- Acuñada por Niclaus Wirth en 1976
  - Premio Turing 1984.
    - Diseñador de los lenguajes de programación Euler, Algol, Pascal, Modula, Modula-2 y Oberon.

Tipo de Dato

Definición

- Conjunto de valores que puede asumir una propiedad de una clase.
  - **TDP** (Tipos de Datos Predefinidos) son los Tipos de Datos **por defecto** de un lenguaje de programación.
    - Número Entero.
    - Número Real.
    - Carácter.
    - Booleano.
    - Referencia.

Estructura de Datos

Definición

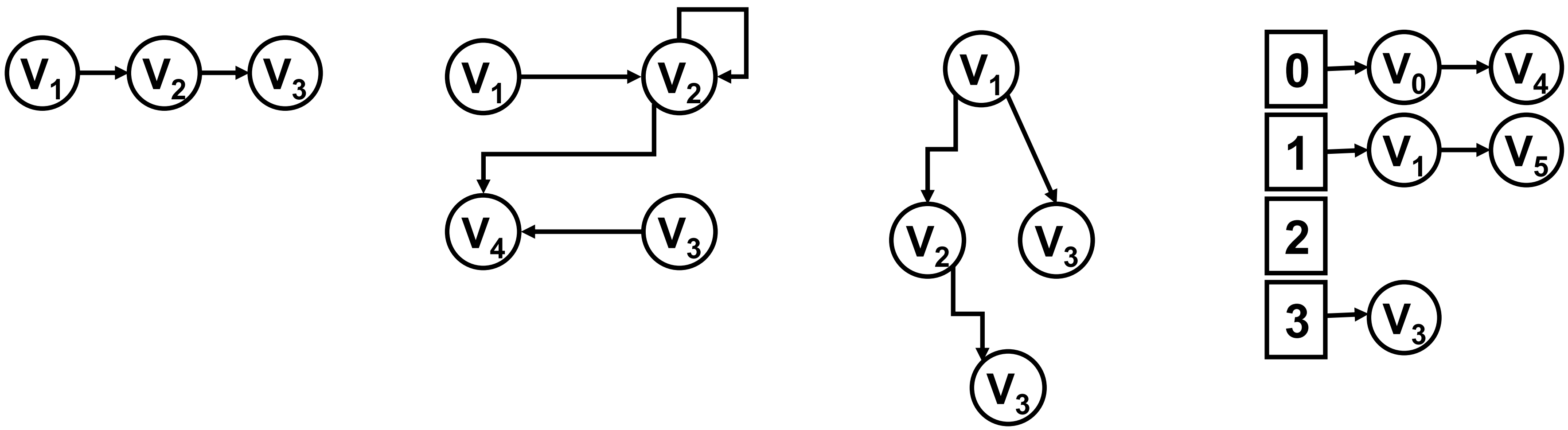
- Conjunto de datos relacionados de una forma determinada[1].
  - Los **TDE** (Tipos de Datos Estructurados) de un lenguaje de programación son colecciones de Tipos de datos almacenados de forma secuencial.
    - Arrays.
    - Cadenas de caracteres.
    - Clases y objetos.
  - Existen otras estructuras de datos básicas *por defecto* implementadas por medio de clases.
    - ArrayList.
    - List.
    - HashMap.
    - Stack.
    - …

[1]***Weiss, Mark Allen; (2000) Estructuras de Datos En Java 2. Addison-Wesley Iberoamericana.***

Estructura de Datos

Clasificación

- Principales familias de estructuras de datos
  - Lineales (listas, pilas y colas).
  - En red (grafos).
  - Jerárquicas (árboles).
  - Diccionario (tablas hash).



- Se pueden realizar combinaciones infinitas de estructuras.

Estructura de Datos

¿Qué estructura elegir?

- La selección de la estructura adecuada para un problema determinado depende de…

1. Adecuación de la estructura a la representación del modelo.
2. Eficiencia de la estructura.
   - Temporal (velocidad de los algoritmos asociados) → $O_T(n)$.
   - Espacial (ocupación en memoria de la estructura) → $O_M(n)$.

Algoritmia (muy) Básica

¿Cuántas veces se ejecuta *test()*?

Algoritmo A — $T_A = 3$

```
{
 test();
 test();

 int i=3;
 return (i*test());
}
```

Algoritmo B — $T_B = 2$

```
{
 test();
 test();
 if (5%2 == 0) {
  test();
  return (test()%2);
 }
 return (0);
}
```

Algoritmia (muy) Básica

¿Cuántas veces se ejecuta *test()*?

Algoritmo C — $T_C(n) = 4n + 6$

```
{
 test();
 test();
 test();

 for (int i=0; i<n; i++) {
  test();
  test();
  test();
  test();
 }

 test();
 test();
 test();
}
```

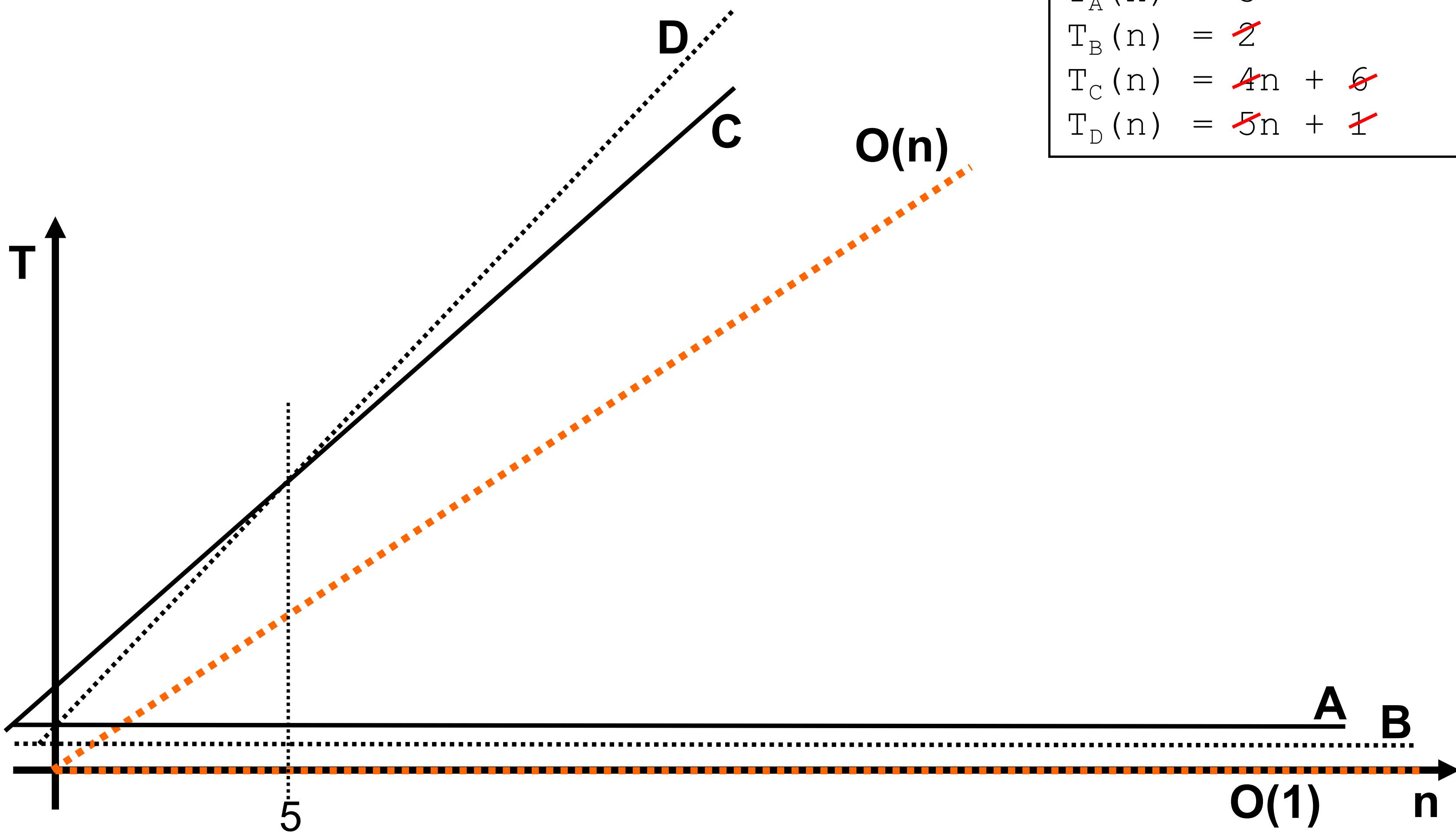
Algoritmo D — $T_D(n) = 5n + 1$

```
{
 for (int i=0; i<n; i++) {
  test();
  test();
  test();
  test();
  test();
 }

 test();
}
```

Algoritmia (muy) Básica

¿Qué algoritmo es más rápido?

Tiempos de Ejecución
$T_A(n) = \cancel{3}$
$T_B(n) = \cancel{2}$
$T_C(n) = \cancel{4}n + \cancel{6}$
$T_D(n) = \cancel{5}n + \cancel{1}$



RECORDAR: en estudios algorítmicos n siempre tiende a infinito

Algoritmia (muy) Básica

¿Cuántas veces se ejecuta *test()*?

Algoritmo E	$T_E(n) = 2n^2 + 1$

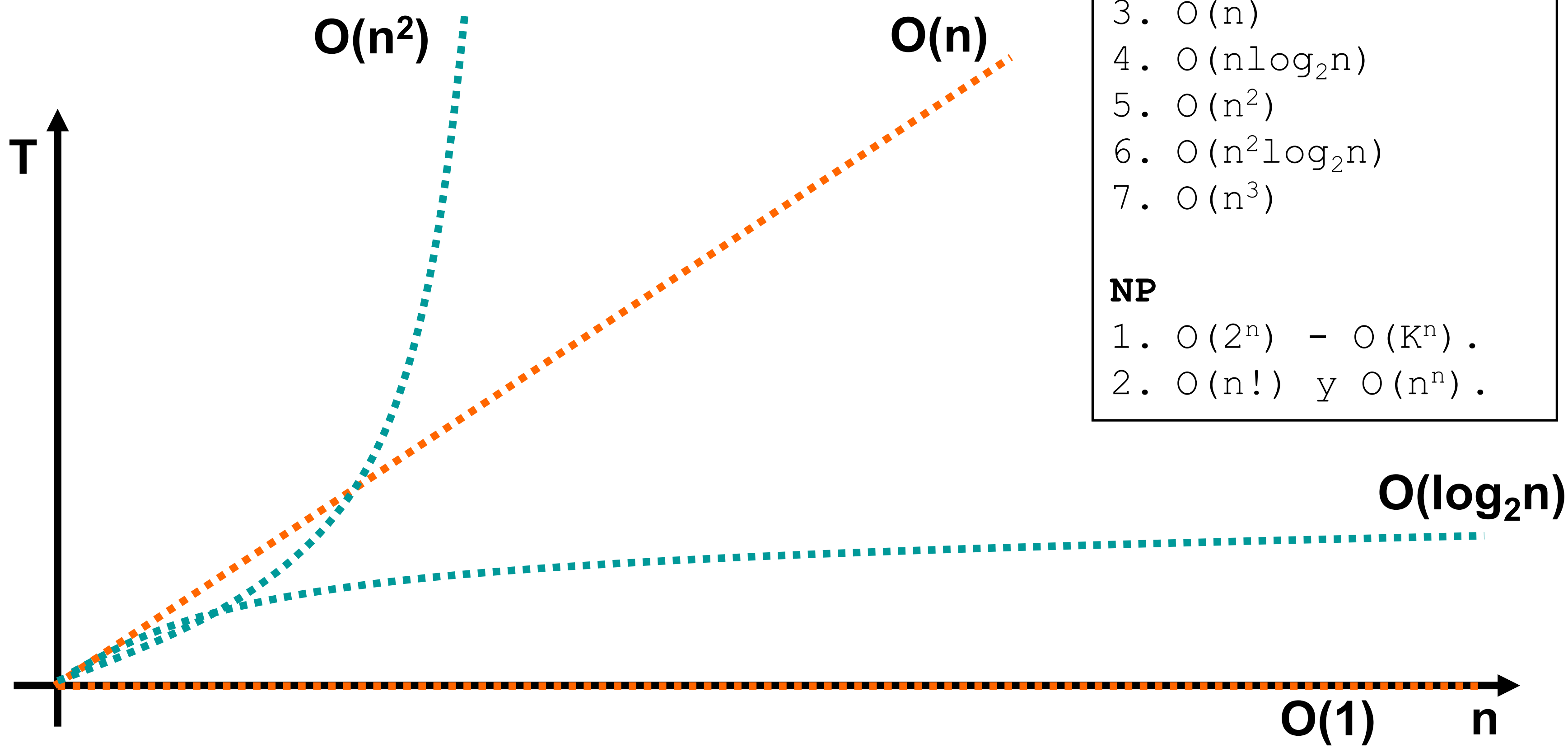
```
{
 for (int i=0; i<n; i++)
  for (int j=0; j<n; j++) {
   test();
   test();
 }
 test();
}
```

Algoritmo F	$T_F(n) = 2([\log_2 n] + 1) + 1$

```
{
 while (n>0) {
  test();
  test();
  n = n/2;
 }

 test();
}
```

Algoritmia (muy) Básica

Complejidad Temporal



Ranking de Eficiencia

P

1. $O(1)$
2. $O(\log_2 n)$
3. $O(n)$
4. $O(n\log_2 n)$
5. $O(n^2)$
6. $O(n^2\log_2 n)$
7. $O(n^3)$

NP

1. $O(2^n)$ – $O(K^n)$.
2. $O(n!)$ y $O(n^n)$.

Algoritmia (muy) Básica

Importancia de la Eficiencia Temporal

N	$T_A(n) = 2^n$	$T_B(n) = n^3$
10	0,1 segundos	10 segundos
15	3,27 segundos	33,7 segundos
20	1,75 minutos	1,3 minutos
25	0,93 horas	2,5 minutos
30	29,8 horas	4,5 minutos
35	39,7 días	7,14 minutos
40	3,4 años	10,66 minutos
45	1,08 siglos	15,18 minutos

Grafos

Dr. Martin Gonzalez-Rodriguez

Estructuras de Datos en Red

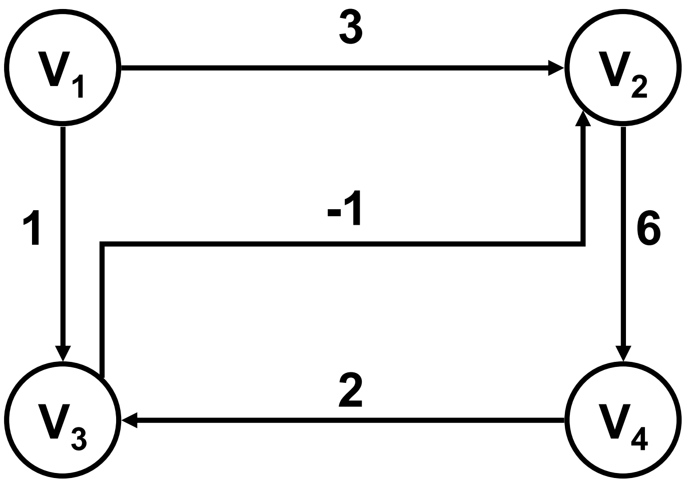
Objetivo

- Modelar relaciones conceptuales complejas entre objetos.
  - Redes de transporte (carreteras, ferrocarril, metro, electricidad, gas, petróleo, etc.).
  - Redes de comunicaciones (Internet, telefonía, correos, etc.)
  - Redes Sociales (Facebook, Google+, préstamo, deuda, etc.).
  - Estructuras (moleculares, neuronales, genéticas, etc.).

Definición

¿Qué es un Grafo?

- Un grafo es un **modelo matemático** que permite representar *relaciones arbitrarias* entre objetos.



Definición

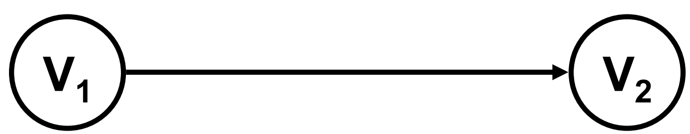
Definición Formal

- Un Grafo es un par (V, E) denotado por G(V, E) donde:
  - **V** es un conjunto finito de **Vértices** (también llamados **Nodos**).

$$\mathbf{V = \{V_1, V_2, ...\}}$$

  - **E** es una familia de **pares de elementos** (v, w) pertenecientes a **V** llamados Aristas (*Edges*).
    - Representan relaciones entre el vértice v y el vértice w.

$$\mathbf{E = \{(V_1, V_2),...\}}$$

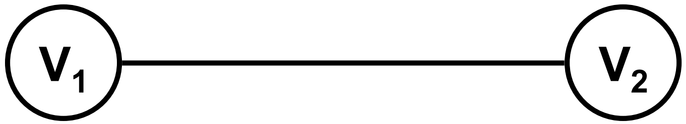


Tipología

Tipos de Grafos

- Si los pares {v,w} son ordenados
  - Éstos se conocen como **Arcos** y se dice que el grafo es ***dirigido*** (AKA *Grafo Orientado* o *Digrafo*).

- Si los pares {v, w} **no son** ordenados
  - Éstos se conocen como **Aristas** y se dice que el grafo es ***no dirigido***.



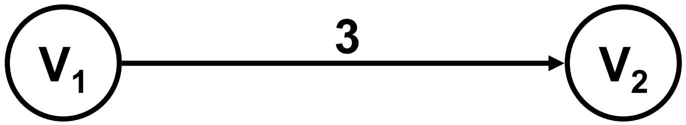
Tipología

Tipos de Grafos

- Un **Grafo Etiquetado** es un trío (V, E, W) denotado por G(V, E, W) donde
  - W es un **conjunto finito** de etiquetas en el que **cada arco u arista** dispone de su propia etiqueta.

$$\mathbf{W = \{W_1, W_2, \ldots\}}$$

  - Las etiquetas pueden ser:
    - **Números**. Las etiquetas de llaman **pesos** y pueden representar costes o beneficios.



    - **Caracteres** o cadenas de caracteres.

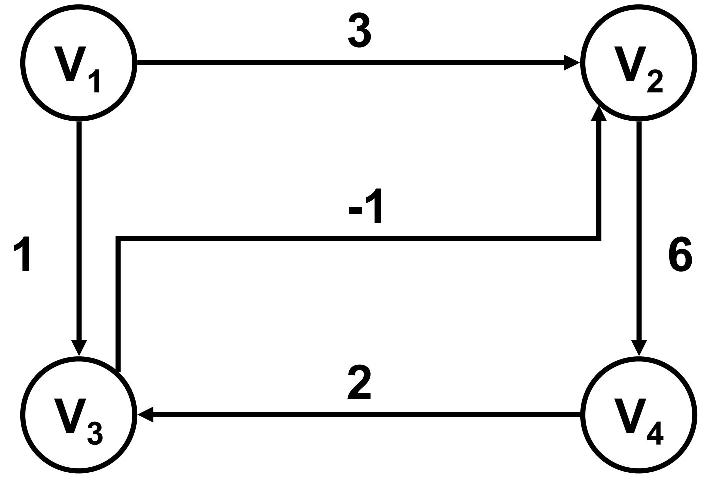
Putting it all Together

Definición Formal Completa

$$V = \{V_1, V_2, V_3, V_4\}$$

$$E = \{(V_1, V_2), (V_1, V_3), (V_2, V_4), (V_3, V_2), (V_4, V_3)\}$$

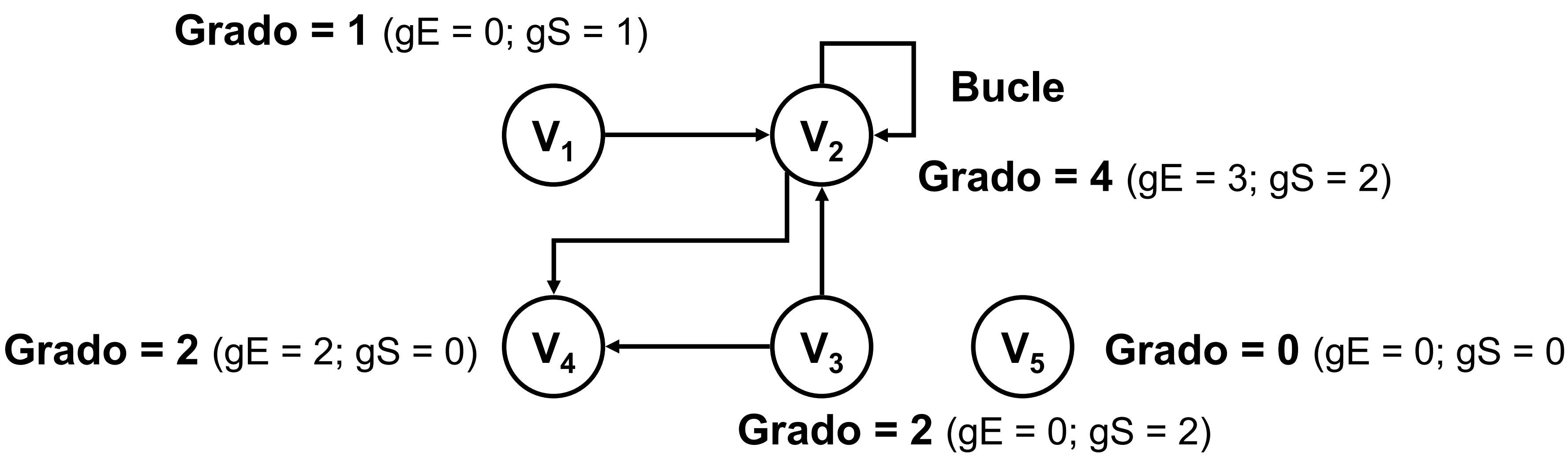
$$W = \{\quad 3, \qquad 1, \qquad 6, \qquad -1, \qquad 2\}$$



Conceptos Básicos

- **Bucle**
  - Arco u arista con igual origen que destino.

- **Grado** de un nodo
  - Número de arcos u aristas conectados al nodo.
    - **Grado de Entrada (gE)** de un nodo:
      - » Número de arcos o aristas que tienen al nodo como destino.
    - **Grado de Salida (gS)** de un nodo:
      - » Número de arcos o aristas que tienen al nodo como origen.

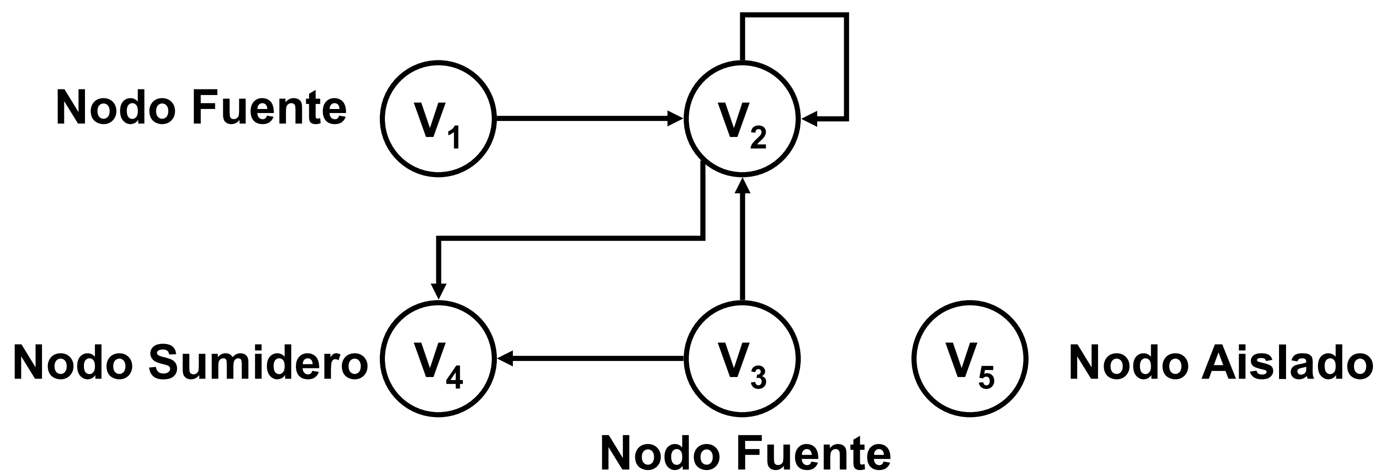


Conceptos Básicos

- Nodo Fuente
  - Si cumple que **GradoSalida > 0** y **GradoEntrada = 0**.

- Nodo Sumidero
  - Si cumple que **GradoSalida = 0** y **GradoEntrada > 0**.

- Nodo Aislado
  - Si cumple que **GradoSalida = 0** y **GradoEntrada = 0**.



Capacidad de un Grafo

n = número de nodos de un grafo

- n = Cardinalidad del conjunto V.

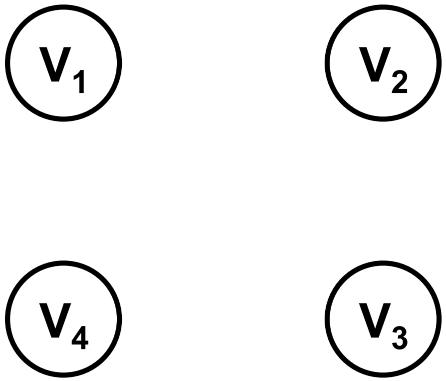
$$\mathbf{V = \{V_1, V_2, \ldots , V_{n-1}, V_n\}}$$

- El valor de **n** se utiliza como parámetro para medir la eficiencia de las operaciones sobre grafos.

Capacidad de un Grafo

Cálculo del número de arcos en base a n

- $A_{min}(n)$: Número **mínimo** de arcos

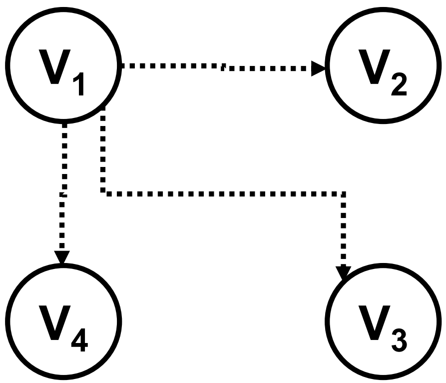


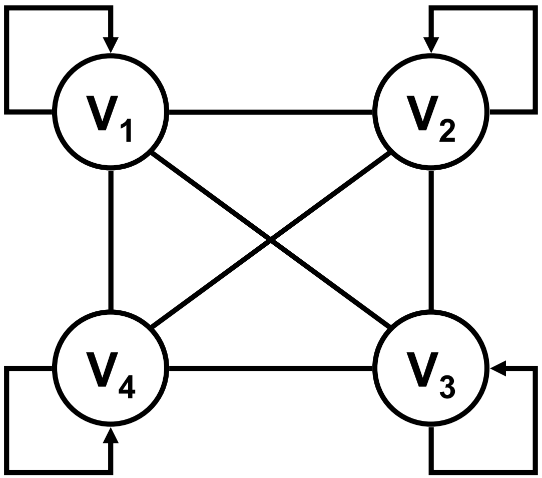
$A_{min}(n) = 0$

Capacidad de un Grafo

Cálculo del número de arcos en base a n

- $A_{max}(n)$: Número **máximo** de arcos (**Grafo Completo**)





$$A_{max}(n) = n(n - 1) = n^2 - n \text{ (sin bucles)}$$

$$A_{max}(n) = n^2 - n + n = n^2 \text{ (con bucles)}$$

Representación en Memoria

Densidad de un grafo

- **Grafos densos**: $A(n) \rightarrow n^2$.
  - Número de arcos similar a la de un grafo completo.
  - Eficiencia máxima sobre memoria estática (matrices, arrays).

- **Grafos ligeros**: $A(n) \rightarrow n$.
  - Promedio de un arco por nodo.
  - Eficiencia máxima sobre memoria dinámica (listas) al requerir de pocos enlaces.

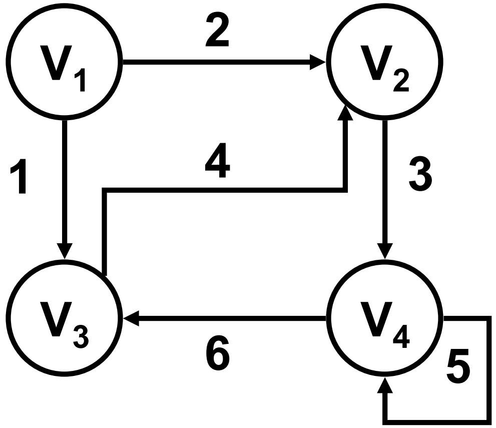
Graph Class – Matrix

Adjacency Matrix

```
private T [] nodes;
private boolean[][] edges;
private double[][] weight;
int size; // real number of nodes stored in the structure
```

- **nodes**: Almacena los objetos de las clases que representan a cada nodo.
- El elemento **edges[i,j]** será *true* **si y solo si** existe un arco que tiene su origen en i y su destino en j.
- El elemento **weight[i, j]** almacena el peso (coste) del arco con origen en i y destino en j.
  - El peso *puede* ser cero (0,0).
  - Si ese arco no existe, el valor *será* cero (0,0).

Graph Class – Matrix



nodes

0	1	2	3	4
V_1	V_2	V_3	V_4	

size = 4

edges

	0	1	2	3	4
0	F	T	T	F	
1	F	F	F	T	
2	F	T	F	F	
3	F	F	T	T	
4					

size = 4

weight

	0	1	2	3	4
0	0	2	1	0	
1	0	0	0	3	
2	0	4	0	0	
3	0	0	6	5	
4					

size = 4

Análisis de Eficiencia

Rendimiento de las Matrices de Adyacencia

- Ventajas
  - Acceso instantáneo a la información de cualquier elemento de las matrices.
    - Acceso O(1).
- Desventajas
  - Dificultad para determinar el tamaño inicial de la matriz.
    - Debería ser lo más cercano posible a n.
  - Desaprovechamiento de memoria en grafos ligeros (matrices casi vacías).
    - Memoria consumida: $O_M(n^2)$.
- Escenario de uso
  - Grafos densos.

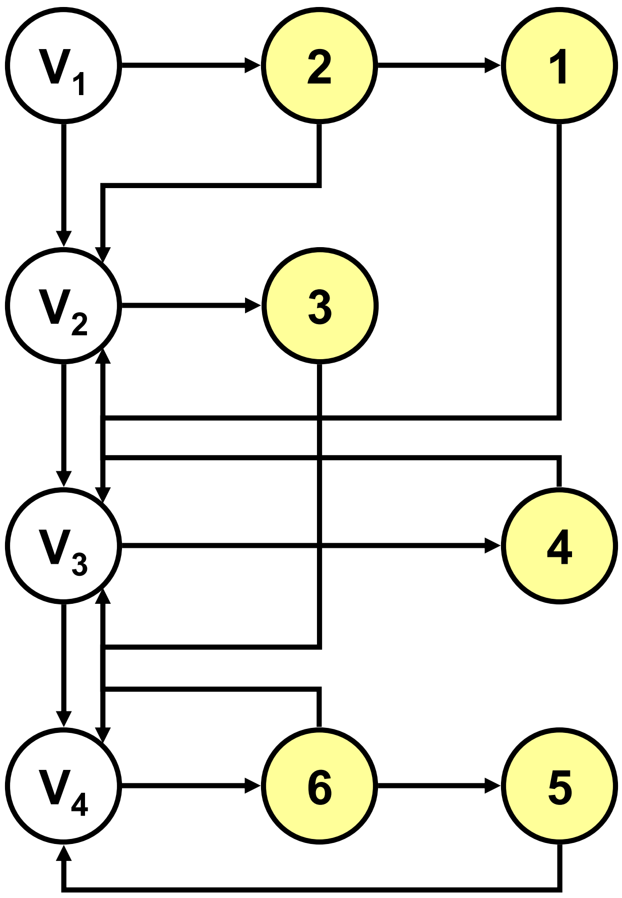
Graph Class – List

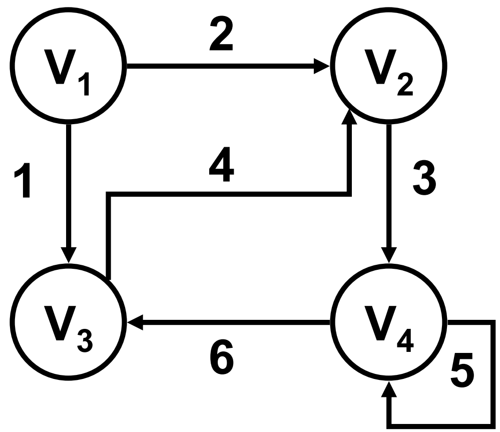
Adjacency List

```
class Edge{
        private double weight;
        private Node target;
}

class Node <T>{
        private T node;
        private LinkedList<Edge> edges;
}

private LinkedList<Node> nodes;
```

- Lista de listas
  - La lista principal (*nodes*) contiene la colección V de nodos.
  - Cada nodo de esta lista contiene a su vez una lista con información sobre sus nodos adyacentes (colección *edges*).

Graph Class – List





Análisis de Eficiencia

Rendimiento de las Listas de Adyacencia

❖ Ventajas

- Memoria consumida en función del número de nodos y del número de aristas reales.
  - Memoria consumida: $O_M(K_1 n + K_2 a)$, donde K_1 = #bytes por nodo y K_2 = #bytes por arco.

❖ Desventajas

- Es necesario realizar complejas búsquedas secuenciales en las listas.
  - Acceso O(n).
- Si el grafo es denso se desaprovecha gran cantidad de memoria en las referencias necesarias para mantener las listas.
  - El grado máximo de desaprovechamiento de memoria se alcanza con el **grafo completo**.

❖ Escenario de Uso

- Grafos ligeros.

Graph Class – Métodos Básicos

Adjacency Matrix

Método	**Complejidad**
graph (constructor)	O(1)
getNode	O(n)
addNode	O(n)
removeNode	O(n)
existEdge ?	O(n)
addEdge	O(n)
removeEdge	O(n)
print	$O(n^2)$

Graph Class – Métodos Básicos

graph (fragment)	O(1)

```
size = 0;
```

nodes

0	1	2	3	4

size = 0

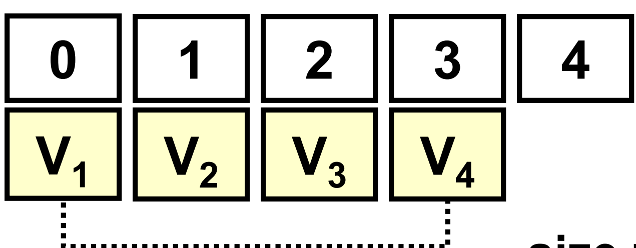
DEBATE: ¿Cuál es la complejidad temporal del algoritmo?

Graph Class – Métodos Básicos

getNode (Pseudocode) **O(n)**

```
public int getNode (T node)
{
 for (int i=0; i<size; i++)
  if (nodes[i].equals(node))
   return (i); // returns the node's position

 return (-1); // search fails, node does not exist
}
```

nodes



size = 4

DEBATE: ¿Cuál es la complejidad temporal del algoritmo?

Graph Class – Métodos Básicos

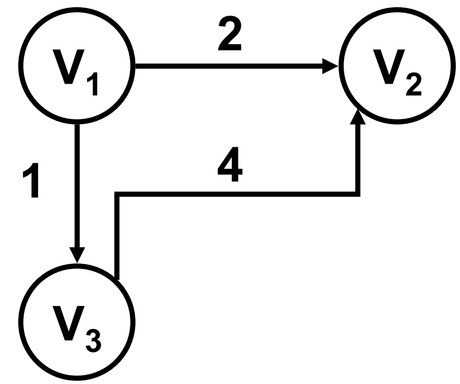
addNode (Pseudocode)	**O(n)**

```
public void addNode (T node)
{
 // precondition: node does not exits and there is
 // available space for the node.

 if (getNode(node)== -1 && size<nodes.length)
 {
  nodes[size] = node;

  //inserts void edges
  for (int i=0; i<=size; i++)
  {
   edges[size][i]=false;
   edges[i][size]=false;
   weight[size][i]=0;
   weight[i][size]=0;
  }
  ++size;
 }
}
```

DEBATE: ¿Cuál es la complejidad temporal del algoritmo?

Graph Class – Métodos Básicos

Antes de insertar V_4



nodes

0	1	2	3	4
V_1	V_2	V_3		

size = 3

edges

	0	1	2	3	4
0	F	T	T		
1	F	F	F		
2	F	T	F		
3					
4					

size = 3

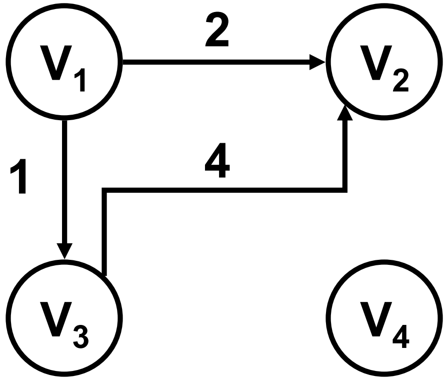
weight

	0	1	2	3	4
0	0	2	1		
1	0	0	0		
2	0	4	0		
3					
4					

size = 3

Graph Class – Métodos Básicos

Después de insertar V_4



nodes

0	1	2	3	4
V_1	V_2	V_3	V_4	

size = 4

edges

	0	1	2	3	4
0	F	T	T	F	
1	F	F	F	F	
2	F	T	F	F	
3	F	F	F	F	
4					

size = 4

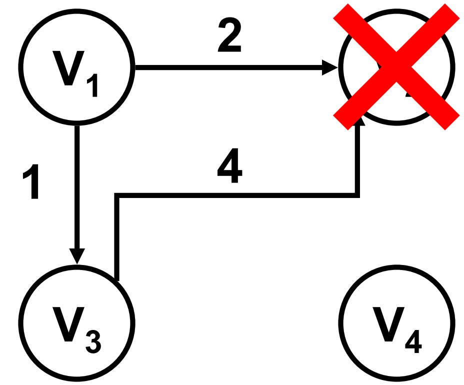
weight

	0	1	2	3	4
0	0	2	1	0	
1	0	0	0	0	
2	0	4	0	0	
3	0	0	0	0	
4					

size = 4

DEBATE: ¿Cómo borrar el nodo V_2?

Después de borrar V_2



nodes

0	1	2	3	4
V_1	V_4	V_3	V_4	

size = 3

edges

	0	1	2	3	4
0	F	F	T	F	
1	F	F	F	F	
2	F	F	F	F	
3	F	F	F	F	
4					

size = 3

weight

	0	1	2	3	4
0	0	0	1	0	
1	0	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4					

size = 3

Graph Class – Métodos Básicos

removeNode (Pseudocode) **O(n)**

```
public void removeNode (T node){
   int i = getNode(node);

   if (i>=0) {
    --size;
    if (i != size+1) { // it is not the last node
     nodes[i] = nodes[size]; //replaces by the last node

     //replace elements in the vectors edges and weights
     for (int j=0; j<=size; j++) {
      edges[j][i]=edges[j][size];
      edges[i][j]=edges[size][j];
      weight[i][j]=weight[size][j];
      weight[j][i]=weight[j][size];
      }
      // loop (diagonal)
      edges[i][i] = edges[size][size];
      weight[i][i] = weight[size][size];
    }
}
```

DEBATE: ¿Cómo implementar el método *existsEdge*?

Graph Class – Métodos Básicos

existsEdge (Pseudocode)	**O(n)**

```
public boolean existsEdge (T origin, T destination)
{
 int i=getNode(origin);
 int j=getNode(destination);

 // precondition: both nodes must exist.
 // if don't… should we throw an exception?

 if (i>=0 && j>=0)
  return(edges[i][j]);
 else
  return (false);
}
```

DEBATE: ¿Cómo implementar el método *addEdge*?

Graph Class – Métodos Básicos

addEdge (Pseudocode) **O(n)**

```
public void addEdge (T origin, T destination, double
edgeWeight)
{

 // precondition: the edge must not already exist.
 if (!existEdge(origin, destination))
 {
  int i=getNode(origin);
  int j=getNode(destination);

  edges[i][j]=true;
  weight[i][j]=edgeWeight;
 }
 else
  ; // what about throwing an exception here?
}
```

DEBATE: ¿Cómo implementar el método *removeEdge*?

Graph Class – Métodos Básicos

removeEdge (Pseudocode)	**O(n)**

```
public void removeEdge (T origin, T destination){

 // precondition: the edge must exist.
 if (existsEdge(origin, destination)) {
  int i=getNode(origin);
  int j=getNode(destination);

  edges[i][j]=false;
  weight[i][j]=0.0;
 }
 else
  ; // what about throwing an exception?
}
```

DEBATO: ¿Cómo implementar el método *print*?

Graph Class – Métodos Básicos

print (Pseudocode)	**O(n²)**

```
public void print(){

 for (int k=0; k<size; k++)
  nodes[k].print();

 for (int i=0; i<size; i++) {
  for (int j=0; j<size; j++) {
   System.out.print(edges[i][j] + "(");
   System.out.print(weight[i][j] + ") ");
  }
  System.out.println();
}
```

DEBATE: ¿Cuál es la complejidad temporal del algoritmo?

Graph Class – Métodos Avanzados

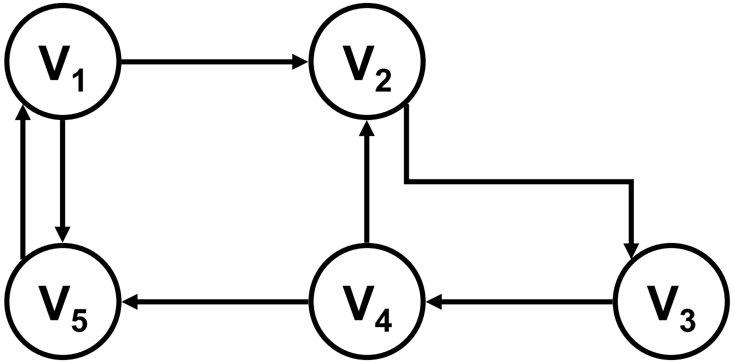
Adjacency Matrix

Método	**Complejidad**
Dijkstra	$O(n^2)$
Floyd	$O(n^3)$
Recorrido en Profundidad	$O(n^2)$
Prim / Warshall	$O(n^2)$

Más Conceptos Básicos

- **Camino** entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Secuencia de nodos (con sus respectivas aristas) que permiten acceder al nodo V_j desde el nodo V_i.
    - Caminos entre V_1 y V_5
      - » $C_A = V_1, V_5$.
      - » $C_B = V_1, V_2, V_3, V_4, V_5$.
      - » $C_C = V_1, V_2, V_3, V_4, V_2, V_3, V_4, V_5$.
      - » …

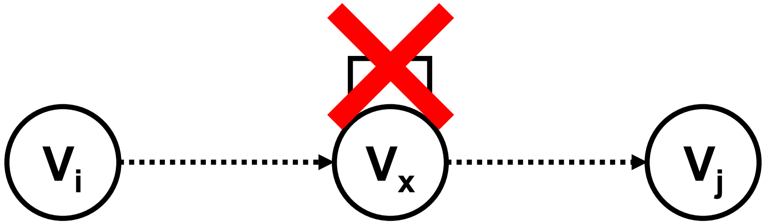
- **Longitud** de un camino entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Número de aristas empleadas para llegar al nodo V_j.
  - Equivale al número de nodos del camino **menos uno**.
    - Longitud de caminos entre V_1 y V_5
      - » $L(C_A) = 1$.
      - » $L(C_B) = 4$.
      - » $L(C_C) = 7$.



Más Conceptos Básicos

- **Camino Simple** entre dos nodos V_i, V_j ($V_i \neq V_j$)
    - Es aquel camino en el que **no se repite** ningún nodo.

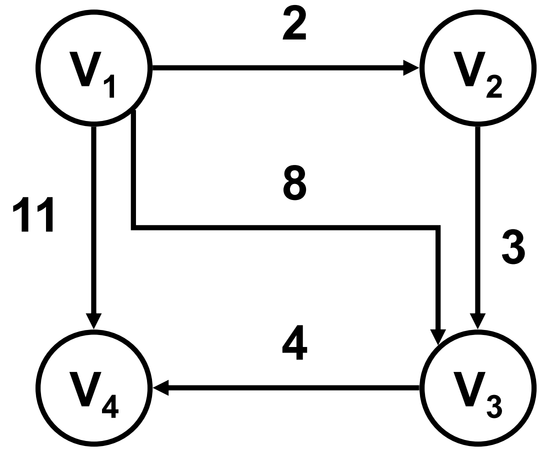
Teorema del Camino Simple
Si existe algún camino entre un par de nodos V_i (origen) y V_j (destino), entonces **existe al menos un camino simple** entre V_i y V_j.



- Es **posible** eliminar los ciclos de un camino para **convertirlo en un camino simple**.

Más Conceptos Básicos

- **Camino de Longitud Mínima** entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Es aquel camino que implique pasar por el menor número de arcos.
    - El Camino de Longitud Mínima **es simple**.
    - Camino de Longitud Mínima entre V_1 y V_4
      - » $C_A = V_1, V_4$ (Longitud 1).



- **Camino de Coste Mínimo** entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Aquel que implica pasar por arcos cuya suma de pesos es mínima.
    - Camino de Coste mínimo entre V_1 y V_4
      - » $C_A = V_1, V_2, V_3, V_4$ (Coste 9).

Algoritmo de Dijkstra

Problema a Resolver

- ¿Cuál es el camino de coste mínimo para acceder a cada uno de los nodos de un grafo desde un nodo ***v*** dado?
  - ¿Cuál es la ruta más barata para llegar a Barcelona **desde Oviedo**?
  - ¿Cuál es la ruta más corta para llegar a Madrid **partiendo de Oviedo**?
    - ¿Y la ruta a Valencia? ¿Y el camino a Sevilla? ¿Y a Bilbao?... **desde Oviedo**.

- Desarrollado por el holandés Edger Dijkstra en 1956
  - Premio Turing 1972.

Algoritmo de Dijkstra

Productos Obtenidos

- **Vector D** (unidimensional) o de Costes Mínimos
  - Guarda el coste mínimo desde ***v*** a cada uno de los nodos del grafo.
- **Vector P** (unidimensional) o de Rutas de Coste Mínimo
  - Almacena la ruta de coste mínimo desde ***v*** a cada uno de los nodos del grafo.

Vector D

V_2	V_3	V_4	V_5	V_6
1	5	3	6	∞

Costes mínimos de ir de V_1 al resto de los nodos

Vector P

2	3	4	5	6
1	4	1	3	-

Se llega a V_3 vía V_4

Para acceder a V_5 hay que ir primero a V_3

Algoritmo de Dijkstra

Ejemplo

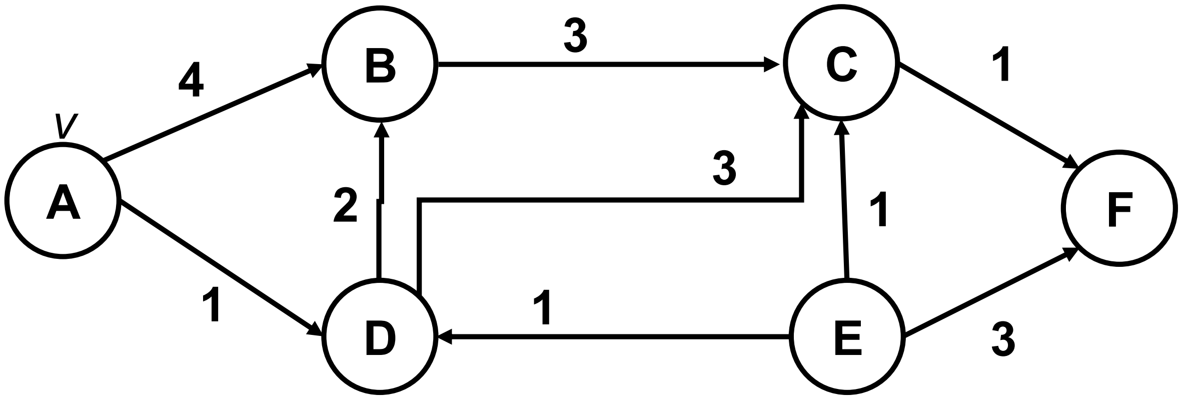
S = {A}

Vector D

V_2	V_3	V_4	V_5	V_6
1	5	3	6	∞

Vector P

2	3	4	5	6
1	4	1	3	-



Algoritmo de Dijkstra

Inicialización

❖ Iniciar **Conjunto S**

- Elementos para los cuales ya se conoce el coste mínimo de ir desde ***v***.
- Se inicializa con el propio ***v*** ya que al principio solo se conoce el coste mínimo de ir desde ***v*** a ***v*** (es decir, cero).
  - S = {v}.

❖ Iniciar **Vector D** de Coste Mínimo

- Copia la fila correspondiente al elemento ***v*** de una matriz **weight** modificada…
  - **…sustituyendo** los valores de coste 0 por ∞.
  - El **coste** de moverse desde un nodo a otro **a través de un camino** (directo) **que no existe es infinito**.
  - En la primera iteración solo se conocen los costes de moverse de ***v*** a todos los demás nodos **a través de un camino directo** (longitud uno).

Algoritmo de Dijkstra

Ejemplo

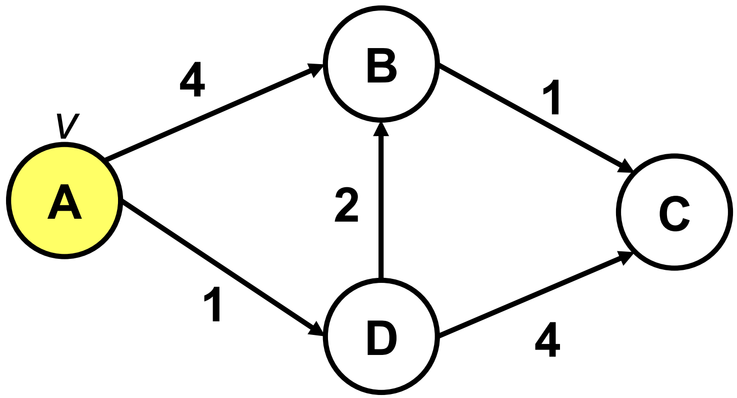
S = {A}

Vector D

B	C	D
4	∞	1

Vector P

B	C	D
-	-	-



Algoritmo de Dijkstra

Ejemplo

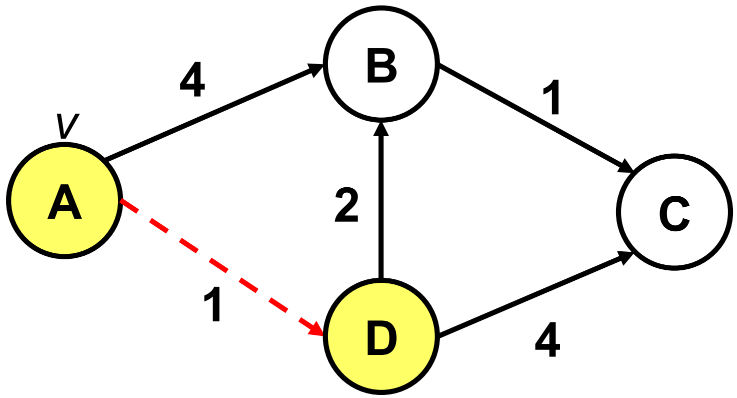
S = {A, D}

Vector D

B	C	D
3	5	1

Vector P

B	C	D
D	D	-



Algoritmo de Dijkstra

Ejemplo

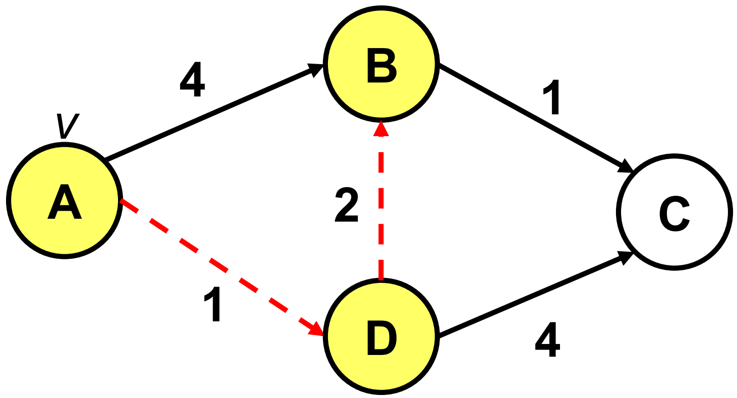
S = {A, D, B}

Vector D

B	C	D
3	4	1

Vector P

B	C	D
D	B	-



Algoritmo de Dijkstra

Ejemplo

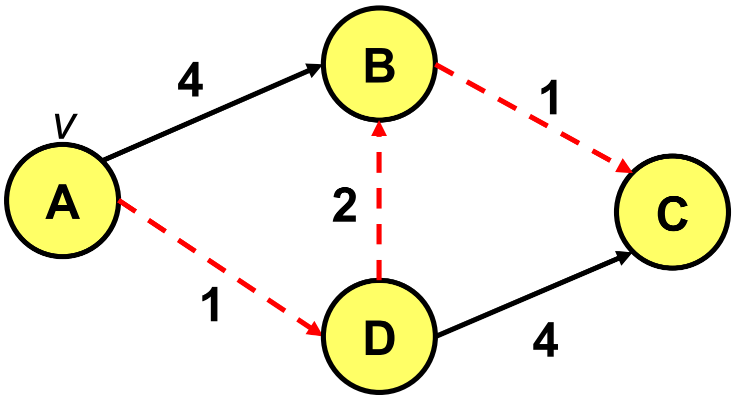
S = {A, D, B, C}

Vector D

B	C	D
3	4	1

Vector P

B	C	D
D	B	-



Algoritmo de Dijkstra

El Algoritmo

En cada Iteración…

1. Evaluar el coste de todos los arcos {k, w} en donde ***k*** pertenece al **conjunto S** y ***w*** al **conjunto V-S.**
2. Seleccionar aquel de coste mínimo, añadiendo ***w*** al **conjunto S.**
   a. w es el nodo con el **menor coste en D!**
3. **Para todo** nodo ***m*** de **V-S** hacer:

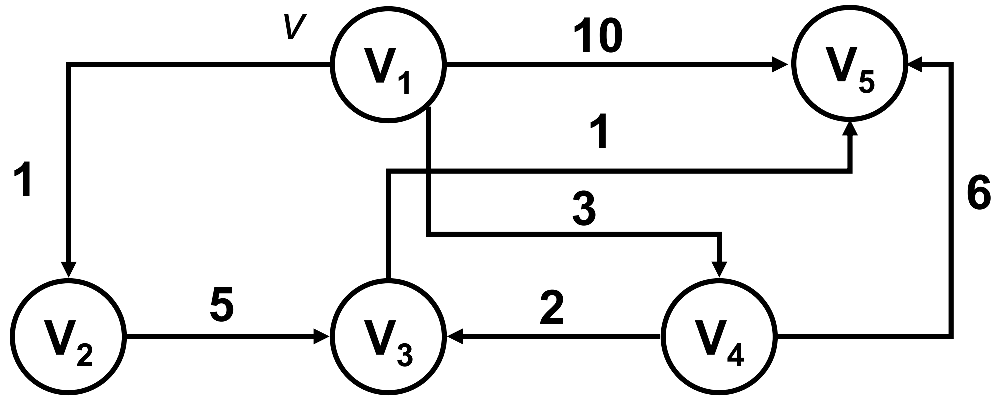
```
if (D[w] + weight[w][m] < D[m]) {
 D[m] = D[w] + weight[w][m];
 P[m] = w;
}
```

Condición de Parada

- **Conjunto S = Conjunto V** (se han explorado todos los nodos del grafo).
  - Realizadas **n – 1 iteraciones**.

Ejercicio

- Costes partiendo de V_1.

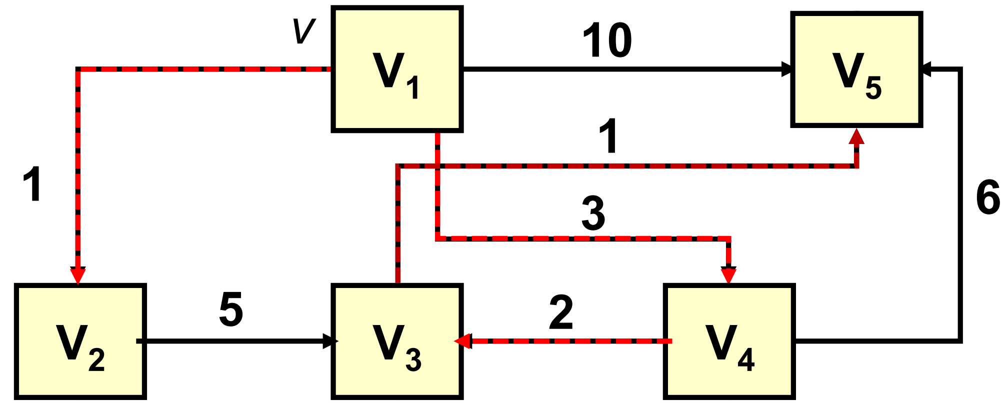


			Vector D					Vector P				
it	S	w	V_2	V_3	V_4	V_5	V_6	2	3	4	5	6
1	1		1	∞	3	10	--	1	-	1	1	-

Algoritmo de Dijkstra

Ejercicio

- Costes partiendo de V_1.

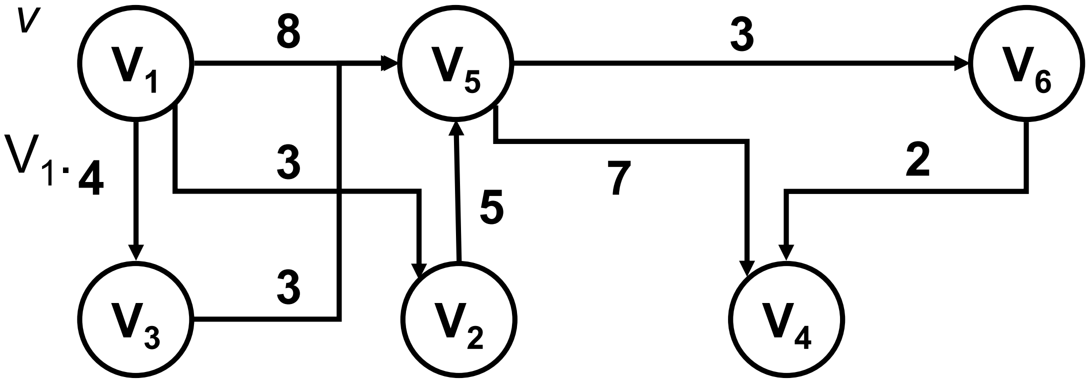


it	S	w	Vector D: V_2	V_3	V_4	V_5	V_6	Vector P: 2	3	4	5	6
1	1		1	∞	3	10	--	1	-	1	1	-
2	1, 2	2	1	6	3	10	--	1	2	1	1	-
3	1, 2, 4	4	1	5	3	9	--	1	4	1	4	-
4	1, 2, 3, 4	3	1	5	3	6	--	1	4	1	3	-
5	1, 2, 3, 4, 5	5	1	5	3	6	--	1	4	1	3	-

Algoritmo de Dijkstra

Ejercicio

- Costes partiendo de V_1.

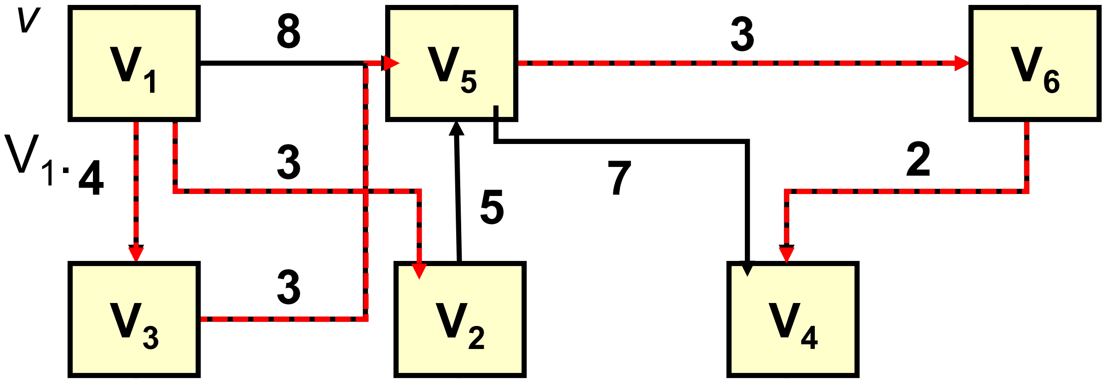


it	S	W	Vector D: V_2	V_3	V_4	V_5	V_6	Vector P: 2	3	4	5	6
1	1		3	4	∞	8	∞	1	1	-	1	-

Algoritmo de Dijkstra

Ejercicio

- Costes partiendo de V_1.



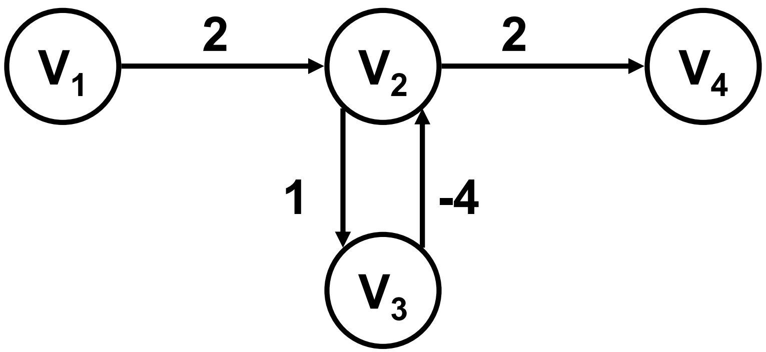
it	S	W	Vector D V_2	V_3	V_4	V_5	V_6	Vector P 2	3	4	5	6
1	1		3	4	∞	8	∞	1	1	-	1	-
2	1, 2	2	3	4	∞	8	∞	1	1	-	1	-
3	1, 2, 3	3	3	4	∞	7	∞	1	1	-	3	-
4	1, 2, 3, 5	5	3	4	14	7	10	1	1	5	3	5
5	1, 2, 3, 5, 6	6	3	4	12	7	10	1	1	6	3	5
6	1, 2, 3, 4, 5, 6	4	3	4	12	7	10	1	1	6	3	5

Algoritmo de Dijkstra

Conclusiones

- Dijkstra supone el coste de ir de un nodo a si mismo como 0
  - Por ello no calcula D[v].

- El algoritmo no funciona con costes negativos (bonificaciones)
  - ¡El camino de coste mínimo no tiene porqué ser simple!



Coste mínimo entre V_1 y V_4 implicaría viajes infinitos entre V_2 y V_3

- Puede calcular el **Camino de Longitud Mínima**
  - Basta con **sustituir costes por 1 en *weight*.**

DEBATE: ¿Cuál es la complejidad temporal del algoritmo?

Algoritmo de Dijkstra

Complejidad Temporal

En cada Iteración… n – 1 iteraciones

1. Evaluar el coste de todos los arcos {k, w} en donde ***k*** pertenece al **conjunto S** y ***w*** al **conjunto V-S.**
2. Seleccionar aquel de coste mínimo, añadiendo ***w*** al **conjunto S.**
   a. w es el nodo con el **menor coste en D!**

3. Para todo nodo ***m*** de **V-S** hacer:

```
if (D[w] + weight[w][m] < D[m]) {
 D[m] = D[w] + weight[w][m];
 P[m] = w;
}
```

O(n)

O(n)

O(n^2)

HOMEWORK

PLAYGROUND

- Consulte la entrada para el **Algoritmo de Dijkstra** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Ponga especial atención a como el uso de **Colas de Prioridad** puede afectar a la complejidad temporal del algoritmo.
    - La estructura de datos *Colas de Prioridad* será tratada cuando se analicen las Estructuras de Datos Jerárquicas.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

Algoritmo de Floyd-Warshall

Problema a Resolver

- Obtener caminos de coste mínimo entre ***cualquier*** par de nodos del grafo
  - ¿Cuál es la ruta más barata para llegar a Barcelona desde Oviedo, Sevilla o Burgos?
    - ¿Aplicar Dijkstra n veces? (una por cada nodo de partida).

- Desarrollado por los estadounidenses Robert Floyd y Stephen Warshall en 1962.

Algoritmo de Floyd-Warshall

Productos Obtenidos (1/2)

- **Matriz A** (AKA matriz de Costes Mínimos)
  - Guarda el coste mínimo de ir **desde cualquier nodo a cada uno de los restantes** nodos del grafo.

Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	12	7	10
V_2	∞	0	∞	10	5	8
V_3	∞	∞	0	8	3	6
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	5	0	3
V_6	∞	∞	∞	2	∞	0

Algoritmo de Floyd-Warshall

Productos Obtenidos (2/2)

- **Matriz P** o de Rutas de Coste Mínimo
  - Almacena la secuencia de nodos que forman **todos los caminos** de coste mínimo.

printPath (fragmento)

```
private void printPath(int i, int j)
{
 int k = P[i][j];
 if (k>0) {
  printPath (i, k);
  System.out.print ('-' + k);
  printPath (k, j);
 }
}

System.out.print (departure);
printPath (departure, arrival);
System.out.println ('-' + arrival);
```

Matriz P	1	2	3	4	5	6
V_1	-	-	-	6	3	5
V_2	-	-	-	6	-	5
V_3	-	-	-	6	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	6	-	-
V_6	-	-	-	-	-	-

TEST: Imprimir la ruta entre V_1 y V_6

Algoritmo de Floyd-Warshall

Inicialización

- Iniciar **Matriz A** de Coste Mínimo
  - Copia de todos los valores de una matriz **weight** modificada de forma idéntica al algoritmo de Dijkstra
    - **Sustituyendo** los valores de coste 0 por ∞.
    - **Pero**… **utilizando valores 0 en la diagonal principal** (el coste de ir de un nodo a si mismo se considera nulo).

Algoritmo de Floyd-Warshall

El Algoritmo

Floyd (fragmento)	$O(n^3)$

```
for (int k=0; k<size; k++)                O(n)
 for (int i=0; i<size; i++)               O(n)
  for (int j=0; j<size; j++)              O(n)
    if (A[i][k] + A[k][j] < A[i][j])
    {
        A[i][j] = A[i][k] + A[k][j];
        P[i][j] = k;
    }
```

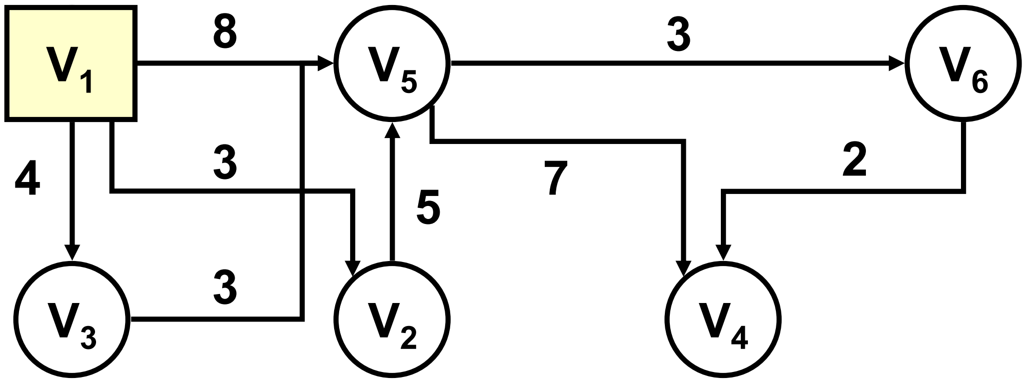
- En cada iteración se considera un nodo k **por el que hay** que pasar obligatoriamente
  - Se ejecutan **n iteraciones**
    - Equivalente de ir añadiendo uno a uno todos los nodos al conjunto S utilizado por Dijkstra.
  - En cada iteración se evalúa el coste de ir de **cualquier nodo i** a **cualquier nodo j** pasando por k.
    - **Si el coste es menor** que el registrado hasta entonces en A, se actualiza el valor de A[i,j] y de P[i,j] indicando que el camino de coste mínimo pasa por k.

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_0 (V_1)



Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	8	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

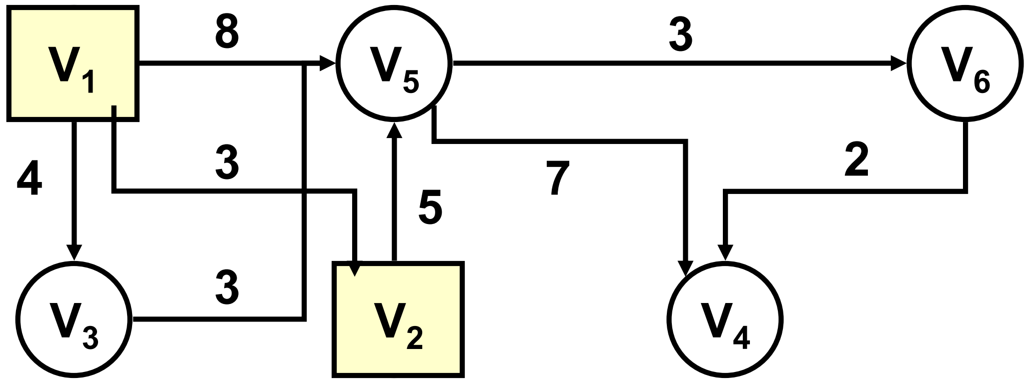
Matriz P	1	2	3	4	5	6
V_1	-	-	-	-	-	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_2 a V_3 vía V_1 (coste ∞ + 4 = ∞) es más barato que ir directamente (coste ∞)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_1 (V_2)



Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	8	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

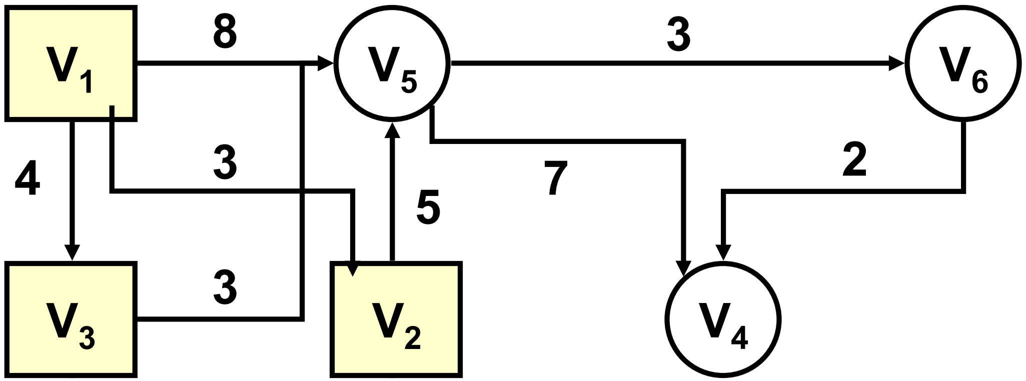
Matriz P	1	2	3	4	5	6
V_1	-	-	-	-	-	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_1 a V_5 vía V_2 (coste 3 + 5 = 8) es más barato que ir con coste A_0 (8)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_2 (V_3)



Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	7	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

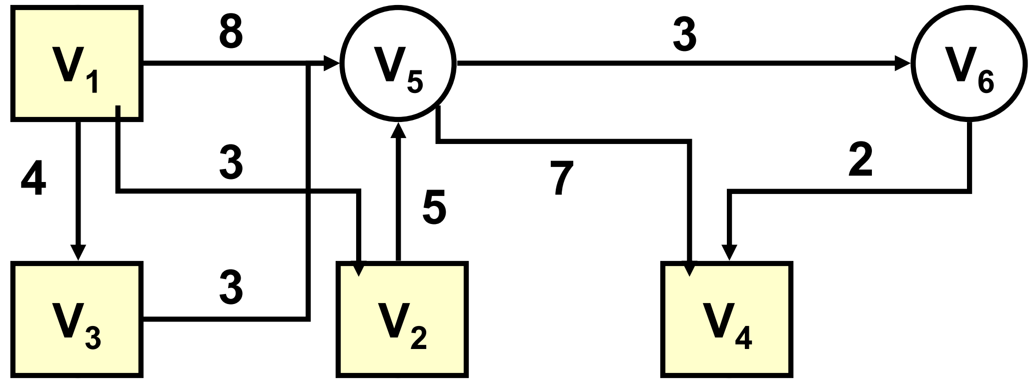
Matriz P	1	2	3	4	5	6
V_1	-	-	-	-	3	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_1 a V_5 vía V_3 (coste 4 + 3 = 7) es más barato que ir con coste A_1 (8)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_3 (V_4)



Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	7	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

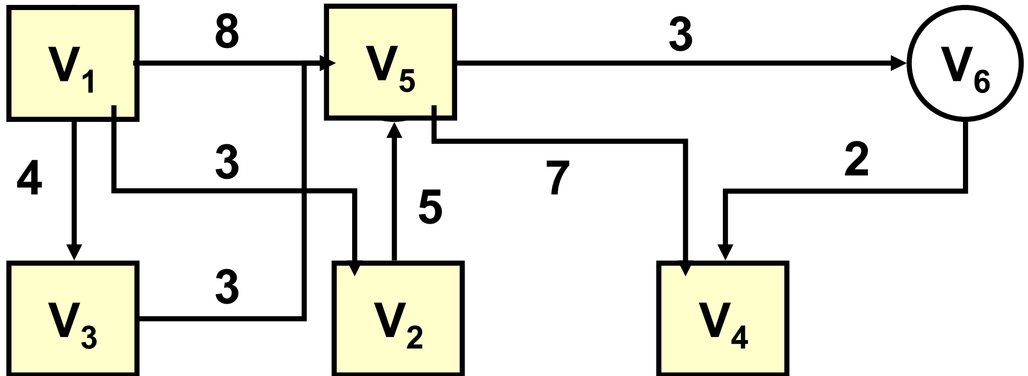
Matriz P	1	2	3	4	5	6
V_1	-	-	-	-	3	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_5 a V_6 vía V_4 (coste 7 + ∞ = ∞) es más barato que ir con coste A_2 (3)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_4 (V_5)



Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	14	7	10
V_2	∞	0	∞	12	5	8
V_3	∞	∞	0	10	3	6
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

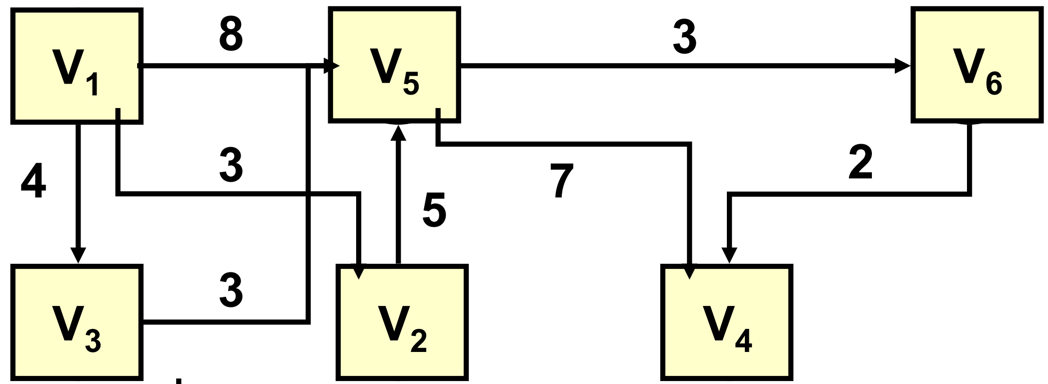
Matriz P	1	2	3	4	5	6
V_1	-	-	-	5	3	5
V_2	-	-	-	5	-	5
V_3	-	-	-	5	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_1 a V_4 vía V_5 (coste 7 + 7 = 14) es más barato que ir con coste A_3 (∞)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_5 (V_6)



Matriz A

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	12	7	10
V_2	∞	0	∞	10	5	8
V_3	∞	∞	0	8	3	6
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	5	0	3
V_6	∞	∞	∞	2	∞	0

Matriz P

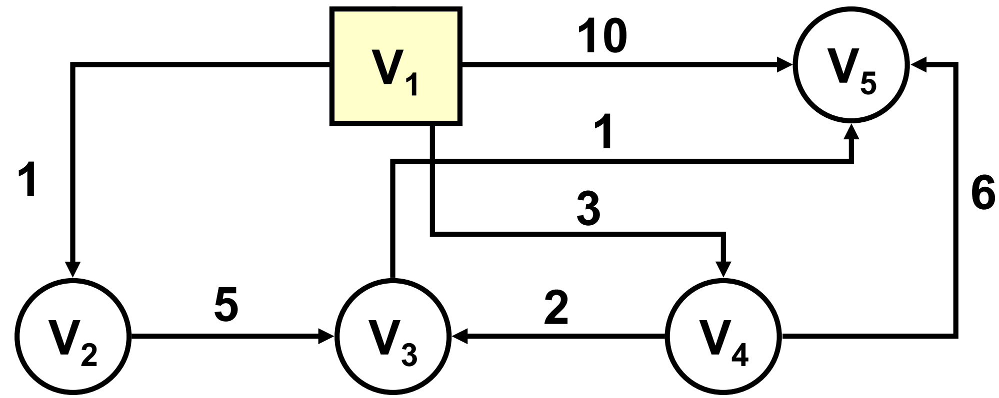
	1	2	3	4	5	6
V_1	-	-	-	6	3	5
V_2	-	-	-	6	-	5
V_3	-	-	-	6	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	6	-	-
V_6	-	-	-	-	-	-

¿Ir de V_1 a V_4 vía V_6 (coste 10 + 2 = 10) es más barato que ir con coste A_4 (14)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_0 (V_1)



Matriz A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	∞	3	10
V_2	∞	0	5	∞	∞
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	6
V_5	∞	∞	∞	∞	0

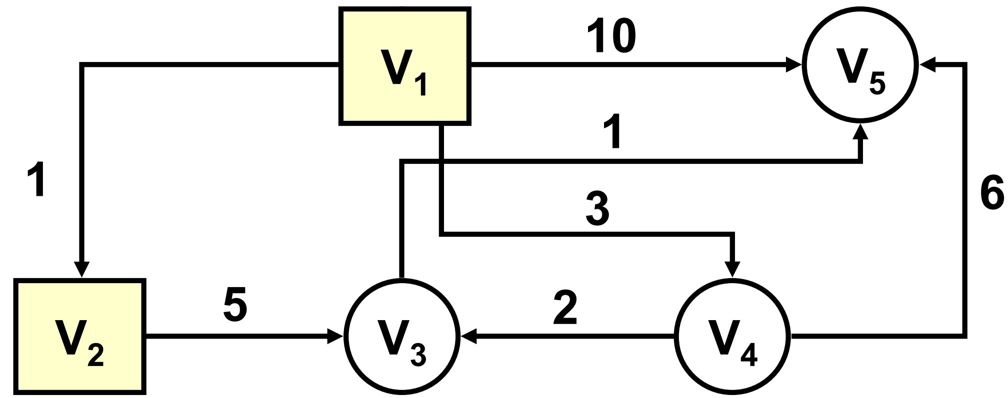
Matriz P	1	2	3	4	5
V_1	-	-	-	-	-
V_2	-	-	-	-	-
V_3	-	-	-	-	-
V_4	-	-	-	-	-
V_5	-	-	-	-	-

¿Ir de V_4 a V_5 vía V_1 (coste ∞ + 10 = ∞) es más barato que ir directamente (6)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_1 (V_2)



Matriz A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	6	3	10
V_2	∞	0	5	∞	∞
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	6
V_5	∞	∞	∞	∞	0

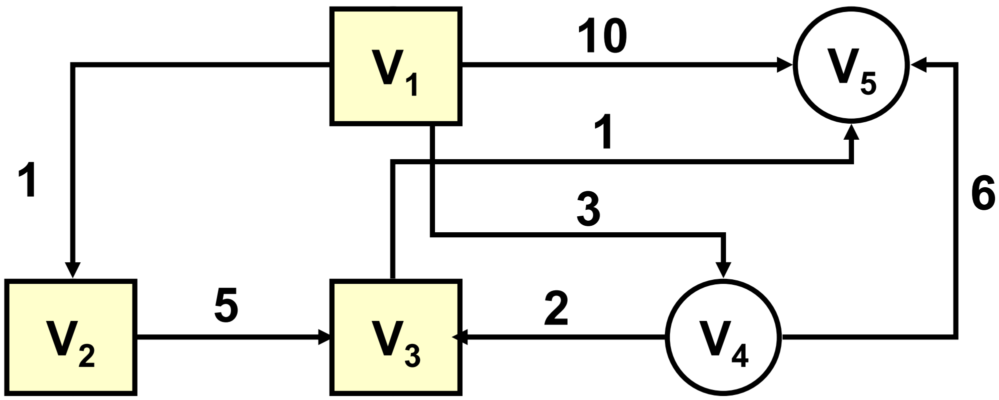
Matriz P	1	2	3	4	5
V_1	-	-	2	-	-
V_2	-	-	-	-	-
V_3	-	-	-	-	-
V_4	-	-	-	-	-
V_5	-	-	-	-	-

¿Ir de V_1 a V_3 vía V_2 (coste 1 + 5 = 6) es más barato que ir con coste A_0 (∞)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_2 (V_3)



Matriz A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	6	3	7
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
V_5	∞	∞	∞	∞	0

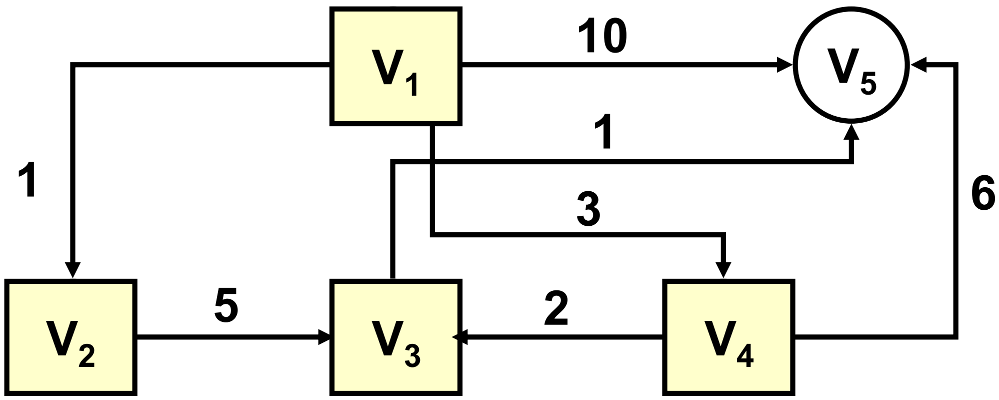
Matriz P	1	2	3	4	5
V_1	-	-	2	-	3
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

¿Ir de V_1 a V_5 vía V_3 (coste 6 + 1 = 7) es más barato que ir con coste A_1 (10)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_3 (V_4)



Matriz A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	5	3	6
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
V_5	∞	∞	∞	∞	0

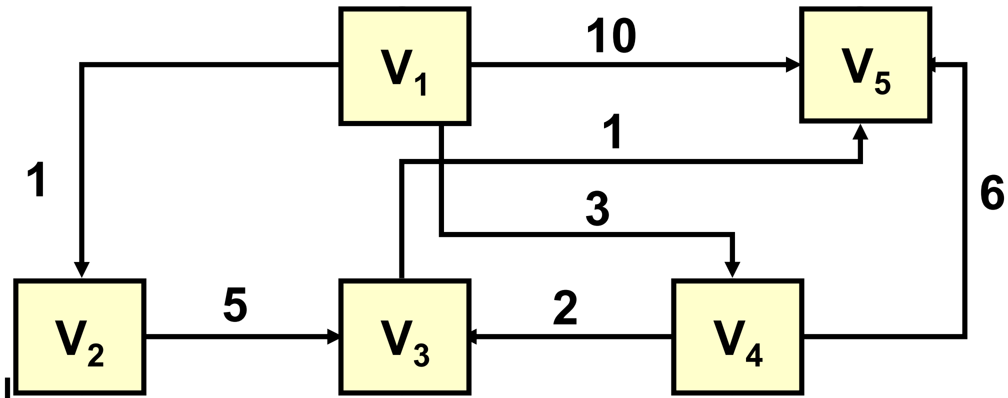
Matriz P	1	2	3	4	5
V_1	-	-	4	-	4
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

¿Ir de V_1 a V_3 vía V_4 (coste 3 + 3 = 5) es más barato que ir con coste A_2 (6)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_4 (V_5)



Matriz A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	5	3	6
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
V_5	∞	∞	∞	∞	0

Matriz P	1	2	3	4	5
V_1	-	-	4	-	4
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

¿Ir de V_1 a V_2 vía V_5 (coste $6 + \infty = \infty$) es más barato que ir con coste A_3 (1)?

Algoritmo de Floyd-Warshall

Floyd para rutas especiales

- Es posible mejorar el algoritmo para calcular caminos de coste mínimo que pasen por un **conjunto L de nodos**.

Floyd (fragmento)

```
    for (int k=0; k<size; k++)
     if (k in L)
      for (int i=0; i<size; i++)
       for (int j=0; j<size; j++)
        if (A[i][k] + A[k][j] < A[i][j])
        {
            A[i][j] = A[i][k] + A[k][j];
            P[i][j] := k;
        }
```

Algoritmo de Floyd-Warshall

Centro de un Grafo Dirigido

- Es centro de un grafo es aquel nodo ***v* más cercano al nodo más distante**.
  - ¿Dónde ubicar un centro de distribución en una región?
  - ¿Dónde colocar el hospital o la estación central en una ciudad?

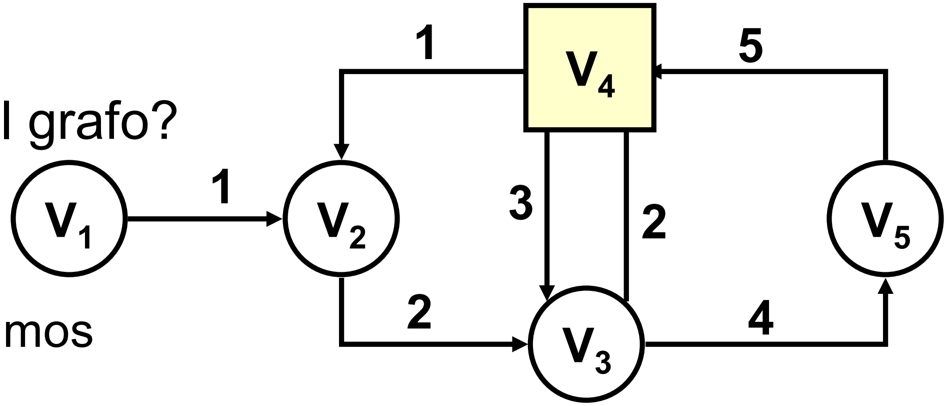
- Excentricidad
  - La excentricidad de un nodo ***v*** es el **máximo de los costes de todos los caminos** de coste mínimo con destino ***v***.
  - El centro de un grafo se encuentra en aquel nodo de **mínima** excentricidad.

Algoritmo de búsqueda del centro de un grafo
1. Aplicar Floyd para **obtener matriz de costes mínimos.** 2. Buscar el **coste mayor en cada columna** (excentricidad de cada nodo destino). 3. Elegir aquel nodo con **la menor excentricidad** como centro del grafo.

Algoritmo de Floyd-Warshall

Ejercicio

- ¿Qué nodo es el centro del grafo?



Obtener mínimo de los máximos

Matriz A Original

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	∞	∞	∞
V_2	∞	0	2	∞	∞
V_3	∞	∞	0	2	4
V_4	∞	1	3	0	∞
V_5	∞	∞	∞	5	0

Matriz A Final

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	3	5	7
V_2	∞	0	2	4	6
V_3	∞	3	0	2	4
V_4	∞	1	3	0	7
V_5	∞	6	8	5	0

Buscar máximo en cada columna

HOMEWORK

PLAYGROUND

- Consulte la entrada para el **Algoritmo de Floyd-Warshall** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Averigüe que quiere decir que el algoritmo utilice *Memoria Cuadrática*.
    - Preste atención al tratamiento que reciben los **Ciclos Negativos** y como pueden ser detectados por el algoritmo.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

Recorrido en Profundidad (DFPrint)

Problema a Resolver

- Recorrer todos los nodos de un grafo a partir de un nodo inicial, siguiendo el camino señalado por sus sus arcos.
  - Emplea la estrategia *visitar primero a los hijos (depth-first) y luego a los hermanos*.
  - Es necesario llevar un control de los nodos visitados.

resetVisited **O(n)**

```
public void resetVisited ()
{
 for (int i=0; i<size; i++)
  nodes[i].setVisited(false);
}
```

Recorrido en Profundidad (DFPrint)

Problema a Resolver

- Recorrer todos los nodos de un grafo a partir de un nodo inicial, siguiendo el camino señalado por sus sus arcos.
  - Emplea la estrategia *visitar primero a los hijos (depth-first) y luego a los hermanos*.
  - Es necesario llevar un control de los nodos visitados.

Deep-first print (pseudocode)

```
public void DFPrint(int v){
  nodes[v].setVisited(true);
  nodes[v].print();

  for each node w accessible from v do
   if (!nodes[w].getVisited())
    DFPrint(w);
}
```

Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_1)*

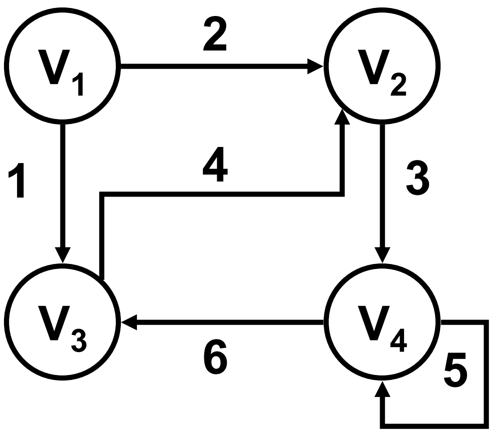
- Antes de visitar V_1

0	1	2	3
V_1	V_2	V_3	V_4
F	F	F	F

nodos

	1	2	3	4
V_1	F	T	T	F
V_2	F	F	F	T
V_3	F	T	F	F
V_4	F	F	T	T

arcos



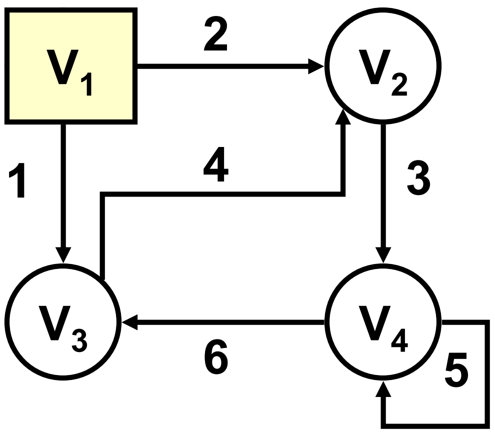
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_1)

- Visitando V_1

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	F	F	F	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



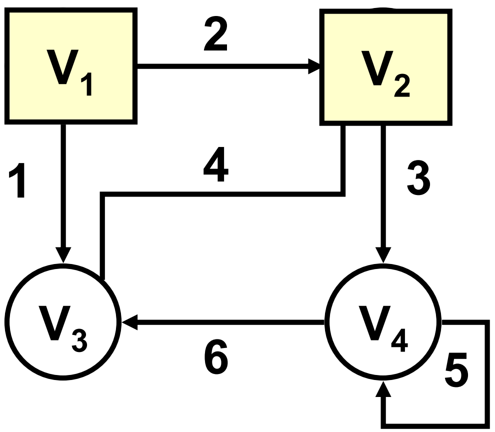
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_1)

- Visitando V_2

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	F	F	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



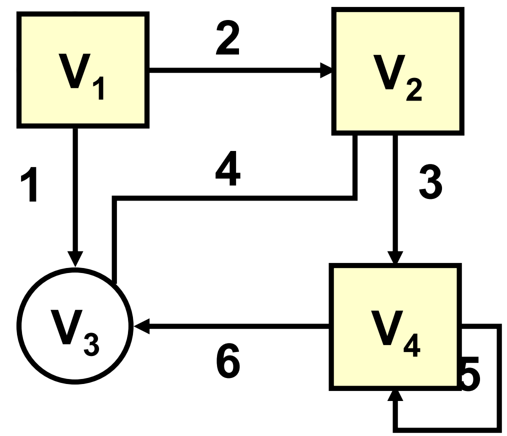
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_1)*

- Visitando V_4

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	F	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



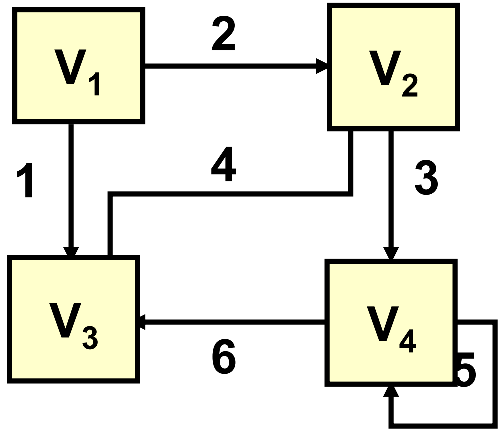
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_1)

- Visitando V_3

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	T	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



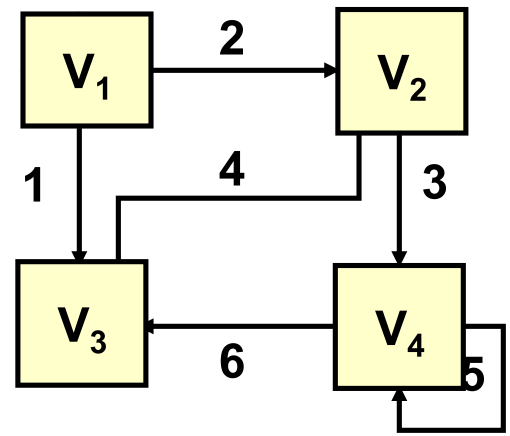
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_1)*

- Continuamos visita de V_4

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	T	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



Recorrido en Profundidad (DFPrint)

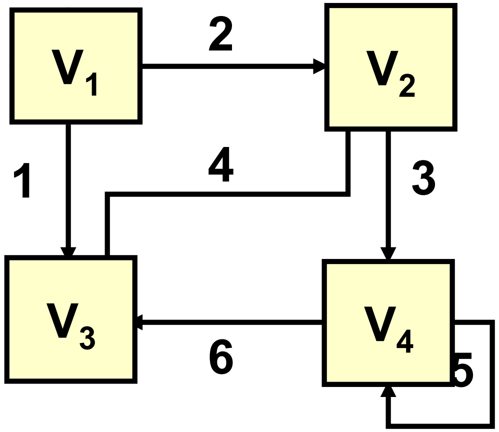
Ejercicio *DFPrint (V_1)*

- Continuamos visita de V_1

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	T	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	n
V_3	F	T	F	F	
V_4	F	F	T	T	

n



$O(n^2)$

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

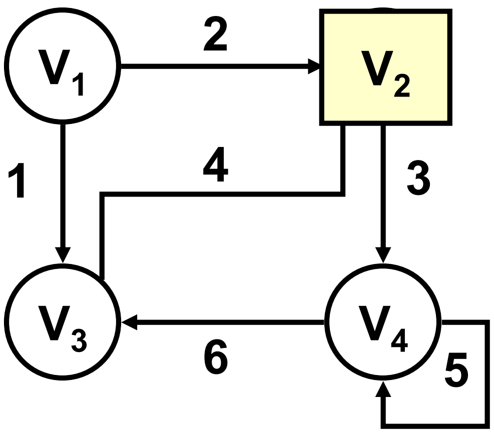
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_2)

- Visitando V_2

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
F	T	F	F	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_2)*

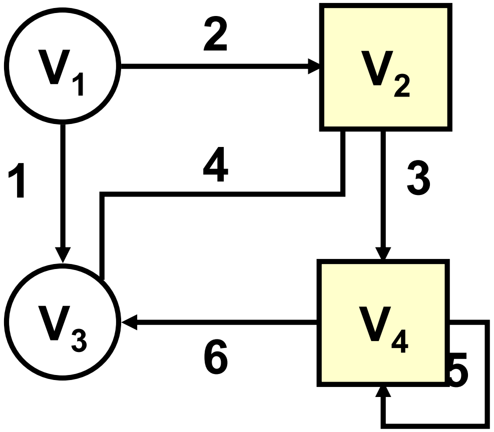
- Visitando V_4

0	1	2	3
V_1	V_2	V_3	V_4
F	T	F	T

nodos

	1	2	3	4
V_1	F	T	T	F
V_2	F	F	F	T
V_3	F	T	F	F
V_4	F	F	T	T

arcos



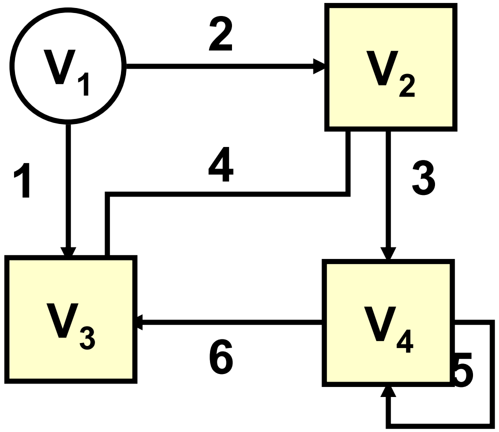
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_2)*

- Visitando V_3

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
F	T	T	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



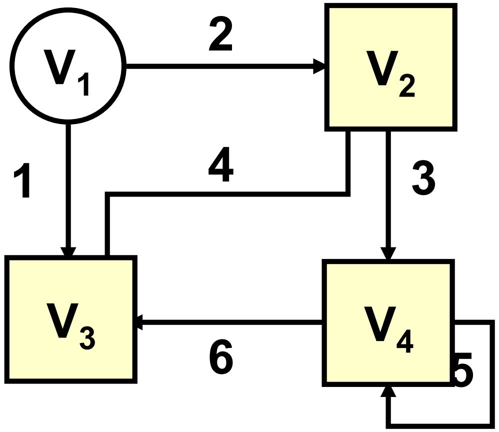
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_2)*

- Continuamos visita de V_4

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
F	T	T	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



Recorrido en Profundidad (DFPrint)

Para garantizar el recorrido completo del grafo

Invocación especial a DFPrint

```
resetVisited();

For (int i=0; i<size; i++)
 if (!nodes[i].getVisited())
   DFPrint (i);
```

Recorrido en Profundidad (DFPrint)

Búsqueda **primero** en profundidad

- Modificación de *DFPrint* para detener el recorrido una vez cumplida una determinada condición sobre un nodo concreto.

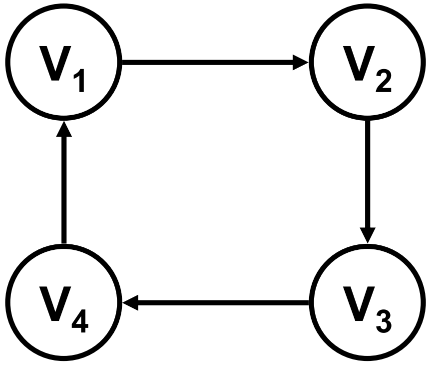
DFSearch (pseudocode)

```
public boolean DFPrint(int v){
  nodes[v].setVisited(true);
  nodes[v].print();

  if (boolean_condition(v))
   return (true);

   for each node w accessible from v do
    if (!nodes[w].getVisited())
     DFPrint(w);

  return (false);
}
```

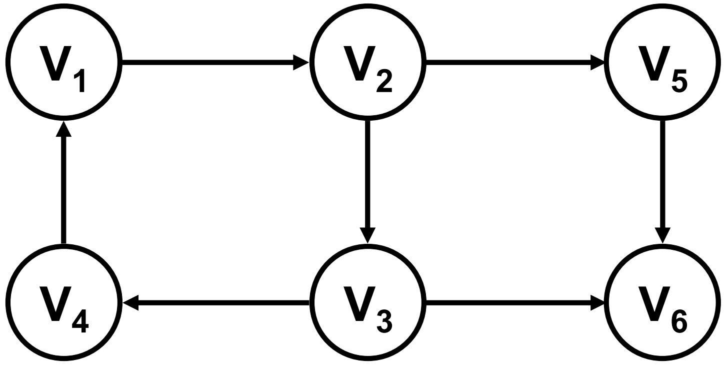
Más Conceptos Básicos

- **Nodo Fuertemente Conexo**
  - Si desde el nodo se puede acceder a todos los demás nodos del grafo **Y** viceversa.

- **Grafo Fuertemente Conexo**
  - Si **todos** los nodos del grafo son fuertemente conexos.
    - Si existe un nodo fuertemente conexo, todos los demás también lo serán y por ende, también el grafo.



Grafo fuertemente conexo

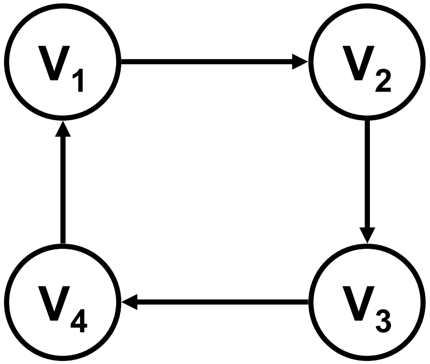


Grafo no conexo (ver V_6)

Más Conceptos Básicos

❖ **Ciclo sobre un nodo**

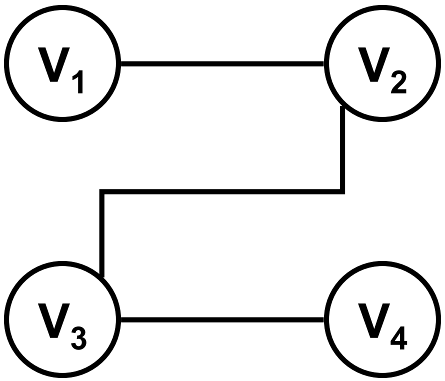
- Camino desde un nodo hasta si mismo.
  - Ciclo para V_1
    - » $C = V_1, V_2, V_3, V_4, V_5$ (longitud 4).



Más Conceptos Básicos

Árbol

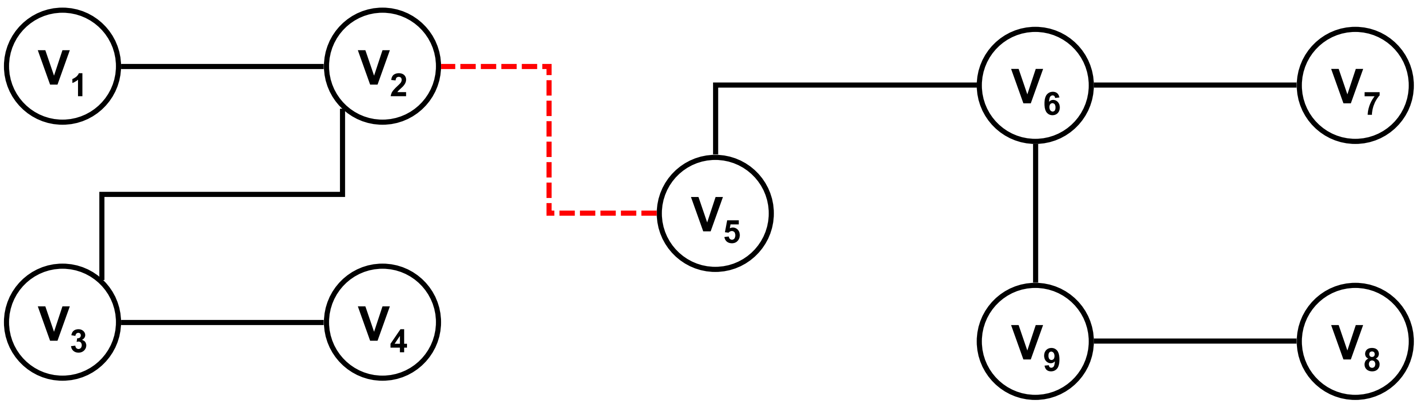
- Grafo Conexo sin Ciclos
  - Todo árbol libre con n > 0 nodos, tiene n - 1 aristas.
  - Si se agrega una arista, ésta formará parte de un ciclo (el grafo deja de ser árbol libre).
  - Para cualquier par de nodos, sólo hay un camino simple.



Más Conceptos Básicos

Árbol Abarcador

- Árbol que conecta todos los **nodos** del grafo.
  - El árbol forma **una única componente conexa.**

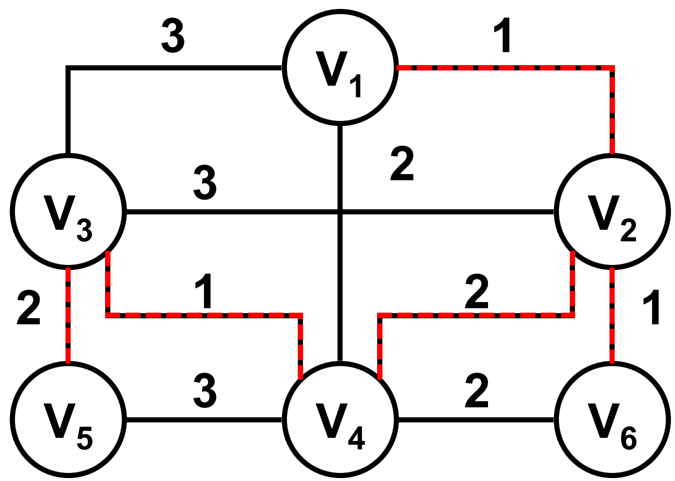


Las dos componentes conexas necesitan contactarse entre ellas para formar un árbol libre abarcador

Más Conceptos Básicos

❖ Árbol Libre Abarcador de Coste Mínimo

- Aquel en el que la suma de los pesos de sus aristas es la mínima posible.
  - Permite conectar todos los componentes de una red al menor coste.



Algoritmo de Prim

Problema a Resolver

- Dado un árbol libre abarcador, devolver el equivalente de coste mínimo
  - ¿Qué carreteras se deben construir para conectar todas las ciudades de Europa de la forma más barata?
  - ¿Cómo se pueden conectar todos los ordenadores de una red con la menor longitud de cable?

- Desarrollado por el estadounidense Robert C. Prim en 1957

Algoritmo de Prim

Inicialización

- **Conjunto T** (vacío)
  - En donde se irán almacenando las aristas que formarán parte del Árbol Libre Abarcador de coste mínimo.

- **Conjunto U** (se inicia con un nodo cualquiera del grafo)
  - Similar al conjunto S del algoritmo de Dijkstra, almacena los nodos que se van evaluando en cada iteración.

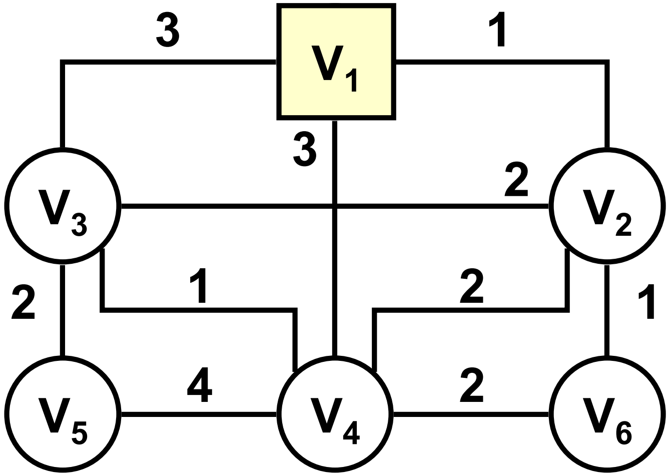
En cada iteracción (mientras que U != V)
1. Evaluar todas las aristas {u, v} en las que u pertenezca a U y v pertenezca a V – U y quedarse con la de menor coste 2. T = T + {u, v} 3. U = U + {v}

- **Condición de Parada**
  - **Conjunto U != Conjunto V** (se han explorado todos los nodos del grafo).
    - Realizadas **n – 1 iteraciones**.

Algoritmo de Prim

Ejercicio 1

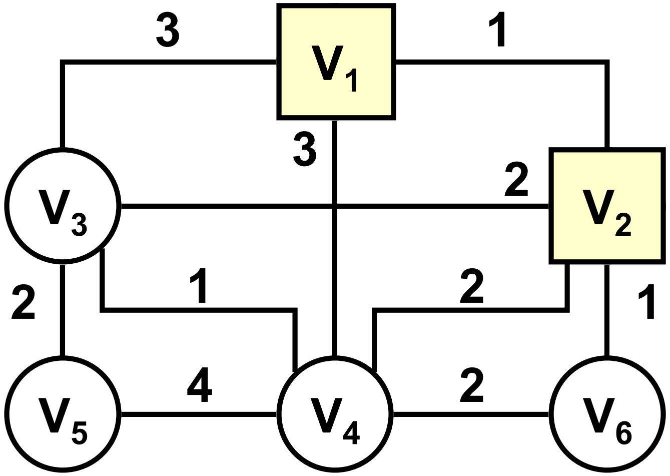
- Empezando con V_1.



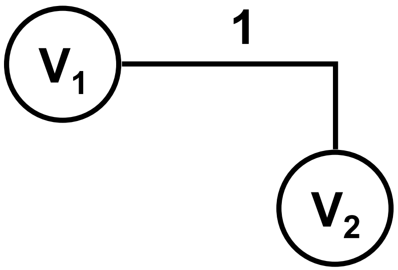
it	U	w
1	1	

Algoritmo de Prim

Ejercicio 1

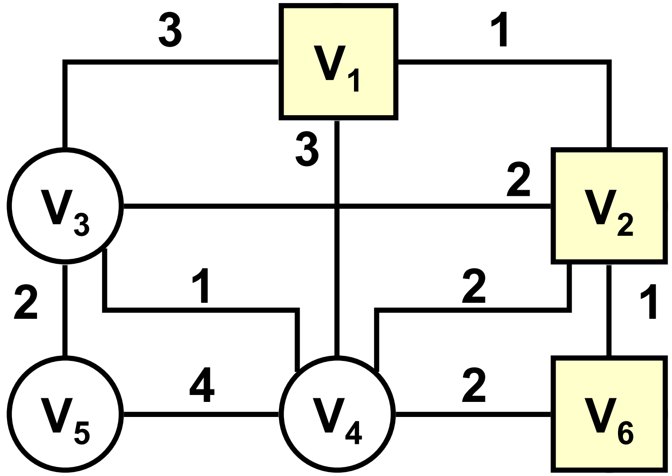


it	U	w
1	1	
2	1, 2	2

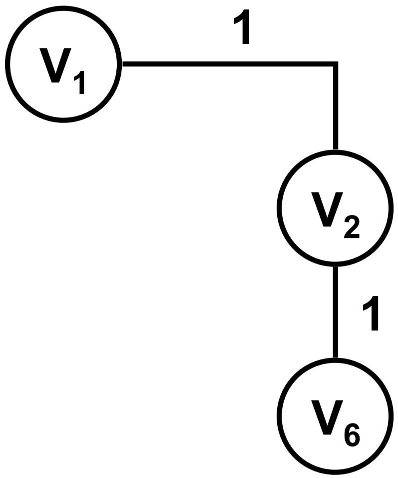


Algoritmo de Prim

Ejercicio 1



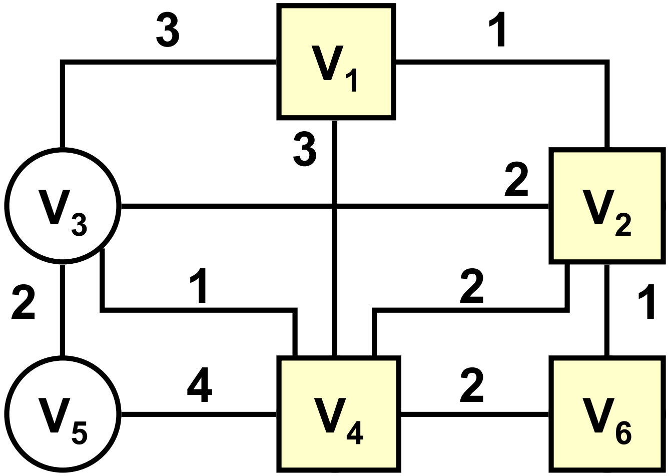
it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6



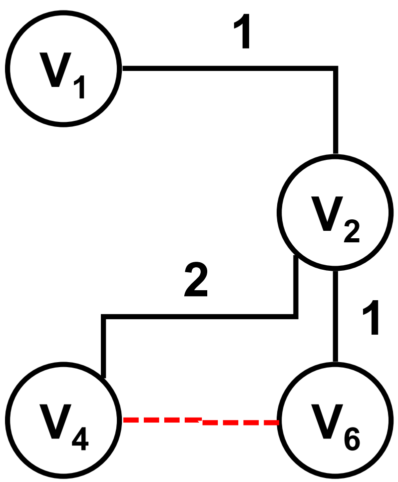
Algoritmo de Prim

Ejercicio 1

- También se podría escoger V_3

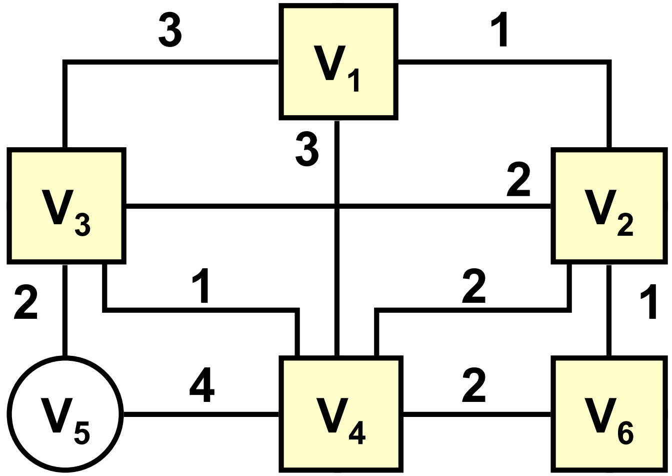


it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4

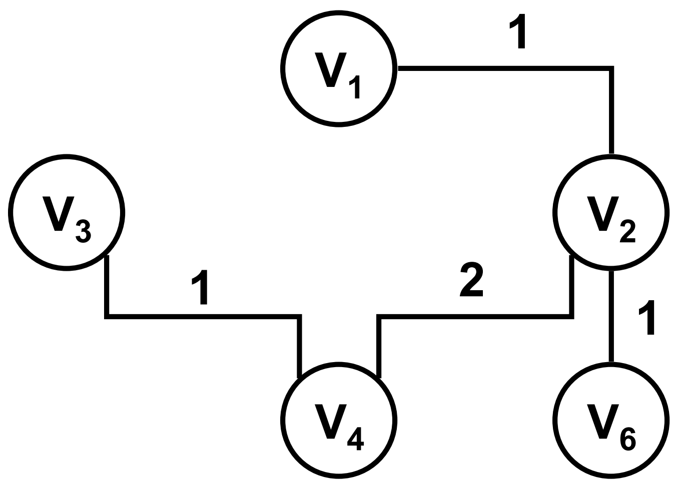


Algoritmo de Prim

Ejercicio 1

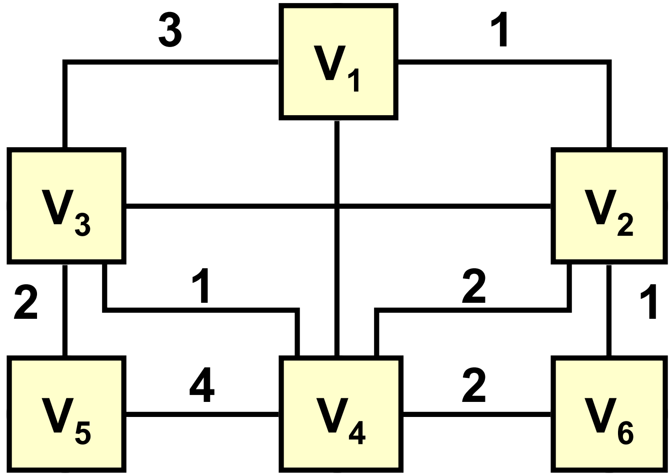


it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4
5	1, 2, 3, 4, 6	3

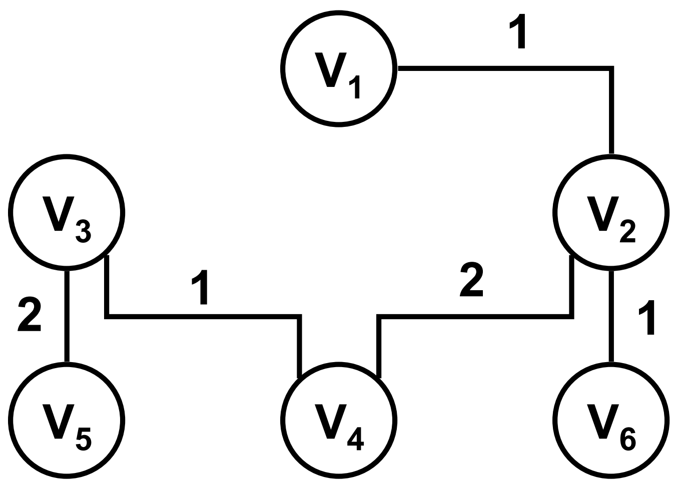


Algoritmo de Prim

Ejercicio 1



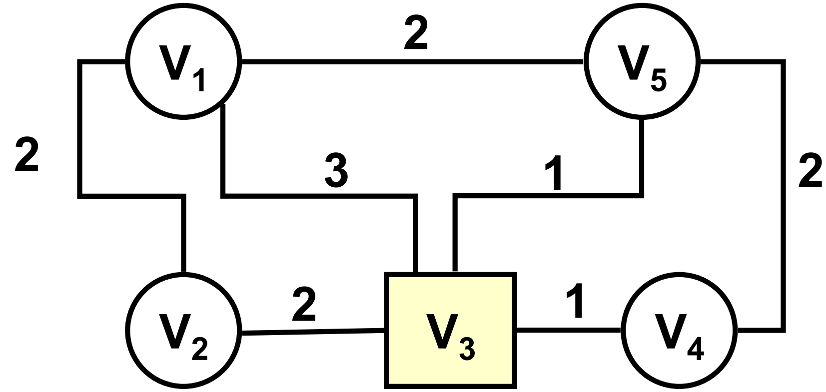
it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4
5	1, 2, 3, 4, 6	3
6	1, 2, 3, 4, 5, 6	5



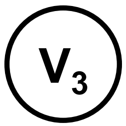
Algoritmo de Prim

Ejercicio 2

- Empezando con V_3.



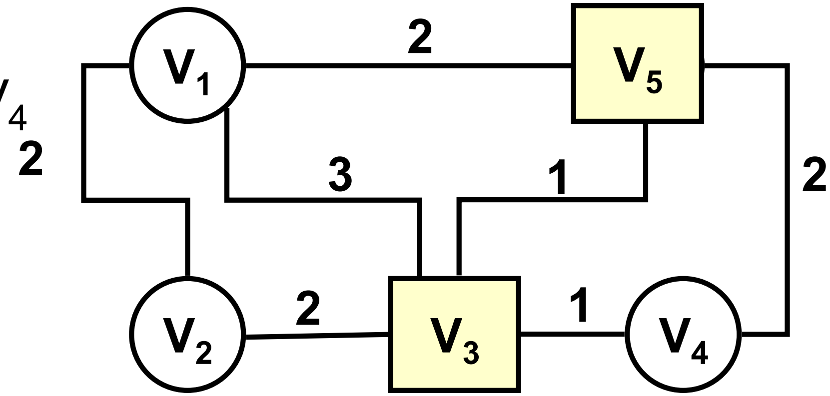
it	U	w
1	3	
2		
3		
4		
5		



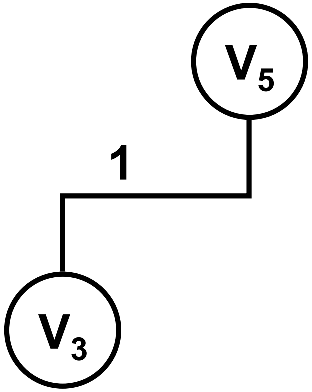
Algoritmo de Prim

Ejercicio 2

- También se podría escoger V_4

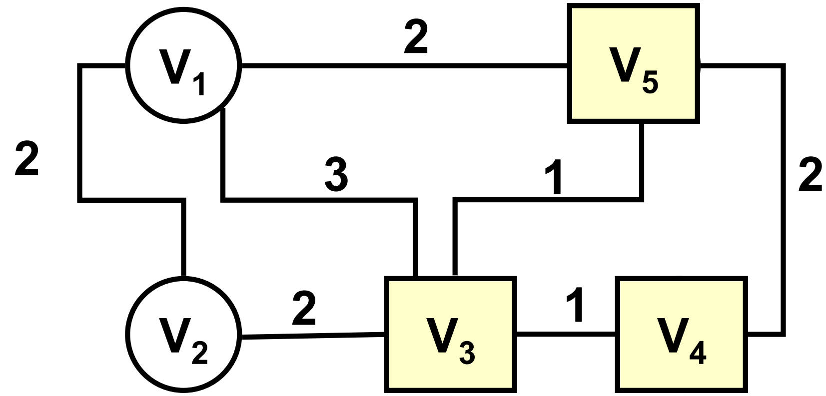


it	U	w
1	3	
2	3, 5	5
3		
4		
5		

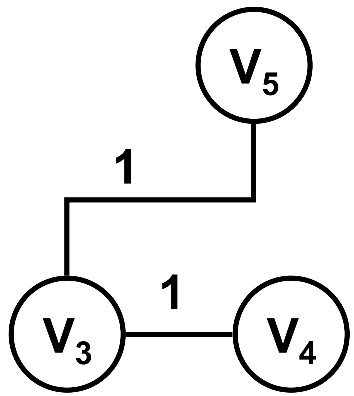


Algoritmo de Prim

Ejercicio 2



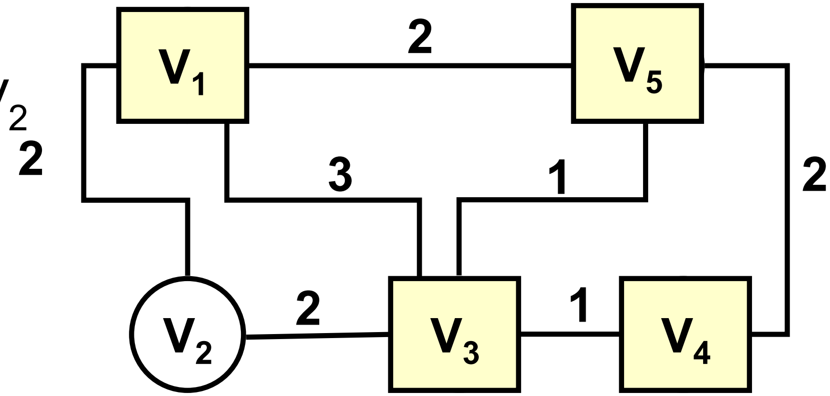
it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4		
5		



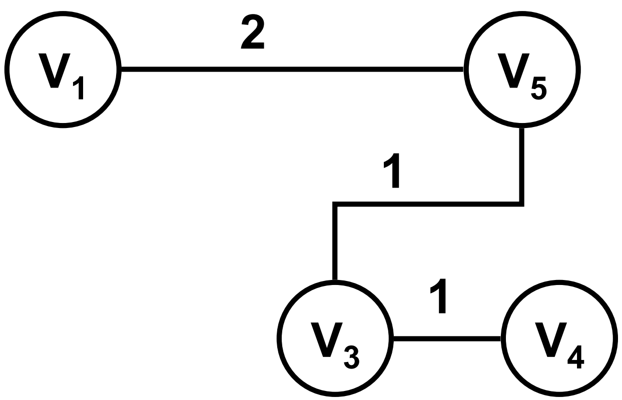
Algoritmo de Prim

Ejercicio 2

- También se podría escoger V_2



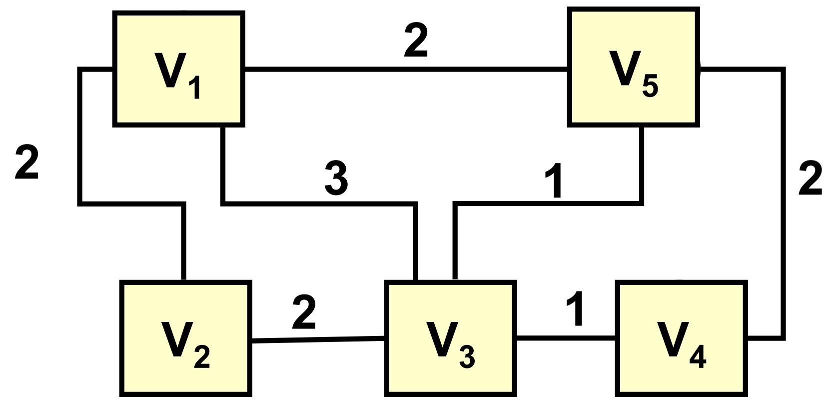
it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4	1, 3, 4, 5	1
5		



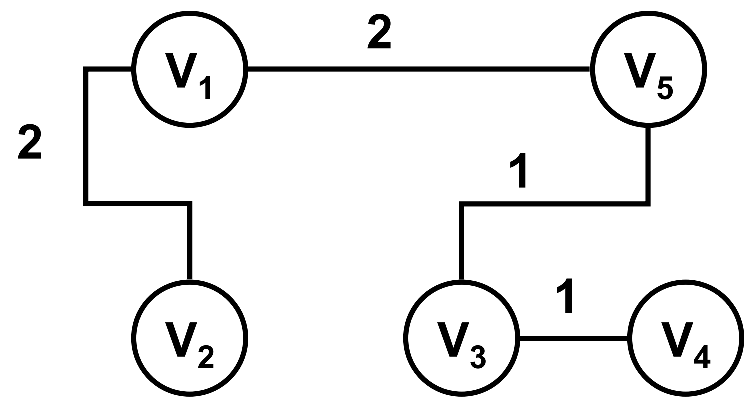
Algoritmo de Prim

Ejercicio 2

- Alternativa: $\{V_2, V_3\}$



it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4	1, 3, 4, 5	1
5	1, 2, 3, 4, 5	2



Algoritmo de Prim

Conclusiones

- El árbol resultante depende de…
  - Nodo de partida.
  - Selección de la arista de coste mínimo en cada iteración.
    - Puede existir más de una con el coste más pequeño.

En cada iteracción (hasta que U == V)	**n**	
1. `Evaluar todas las aristas {u, v} en las que u pertenezca a U y v pertenezca a V - U y quedarse con la de` **`menor`** `coste` 2. `T = T + {u,v}` 3. `U = U + {v}`		**n^2**

$O(n^3)$

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Algoritmo de Prim

- Optimización
  - Utilizar vectores auxiliares **ordenados** para elegir la arista de menor coste, reduciendo la complejidad a O(n).
    - Mayor velocidad a costa de mayor consumo de memoria.

En cada iteracción (hasta que U == V)	**n**

`1. Evaluar todas las aristas {u, v} en las que u pertenezca a U y v pertenezca a V – U y quedarse con la de` **`menor`** `coste`
`2. T = T + {u,v}`
`3. U = U + {v}`
	n

$O(n^2)$

HOMEWORK

PLAYGROUND

- Consulte la entrada para el **Algoritmo de Prim** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Ponga especial atención a la demostración de por qué el algoritmo realmente funciona.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

HOMEWORK

PLAYGROUND

- El problema del Árbol Libre Abarcador de coste mínimo también fue resuelto por el estadounidense Joseph Kruskal.
- Consulte la entrada para el **Algoritmo de Kruskal** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Preste especial atención a las diferencias entre el Algoritmo de Kruskal y el Algoritmo de Prim.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

Apéndices

Referencias

Referencias

AHO, A; HOPCROFT, J; ULLMAN, D; (1988) *Estructuras de Datos y Algoritmos*. Addison-Wesley Iberoamericana. México [Cap 9].

JOYANES AGUILAR, Luis; ZAHONERO MARTÍNEZ, Ignacio; (1998) *Estructura de Datos: Algoritmos, Abstracción y Objetos*. Mc Graw Hill. ISBN: 84-481-2042-6. [Cap 14.]

ORTEGA F., Maruja; (1988) *Grafos y Algoritmos*. Universidad Metropolitana, Oficina Metrópolis.

WEISS, Mark Allen; (2000*) Estructuras de Datos En Java 2*. Addison-Wesley Iberoamericana. ISBN 84-7829-035-4. [Cap 14.].

WEISS, Mark Allen; (1995) *Estructuras de Datos y Algoritmos* Addison-Wesley Iberoamericana. ISBN 0-201-62571-7. [Cap 9.].