RECORRIDOS BÁSICOS EN GRAFOS: DFS

Algoritmos y Programación
Javier Miranda

Escuela de Ingeniería Informática
Universidad de Las Palmas de Gran Canaria

20 de Septiembre de 2023

Estrategias básicas de búsqueda

Recorrido a lo ancho

- *Breadth-First Search (BFS)*
- Exploración por niveles
- Se programa con una cola

Aplicaciones:

- Distancia mínima (en número de saltos)
- Componentes conectados

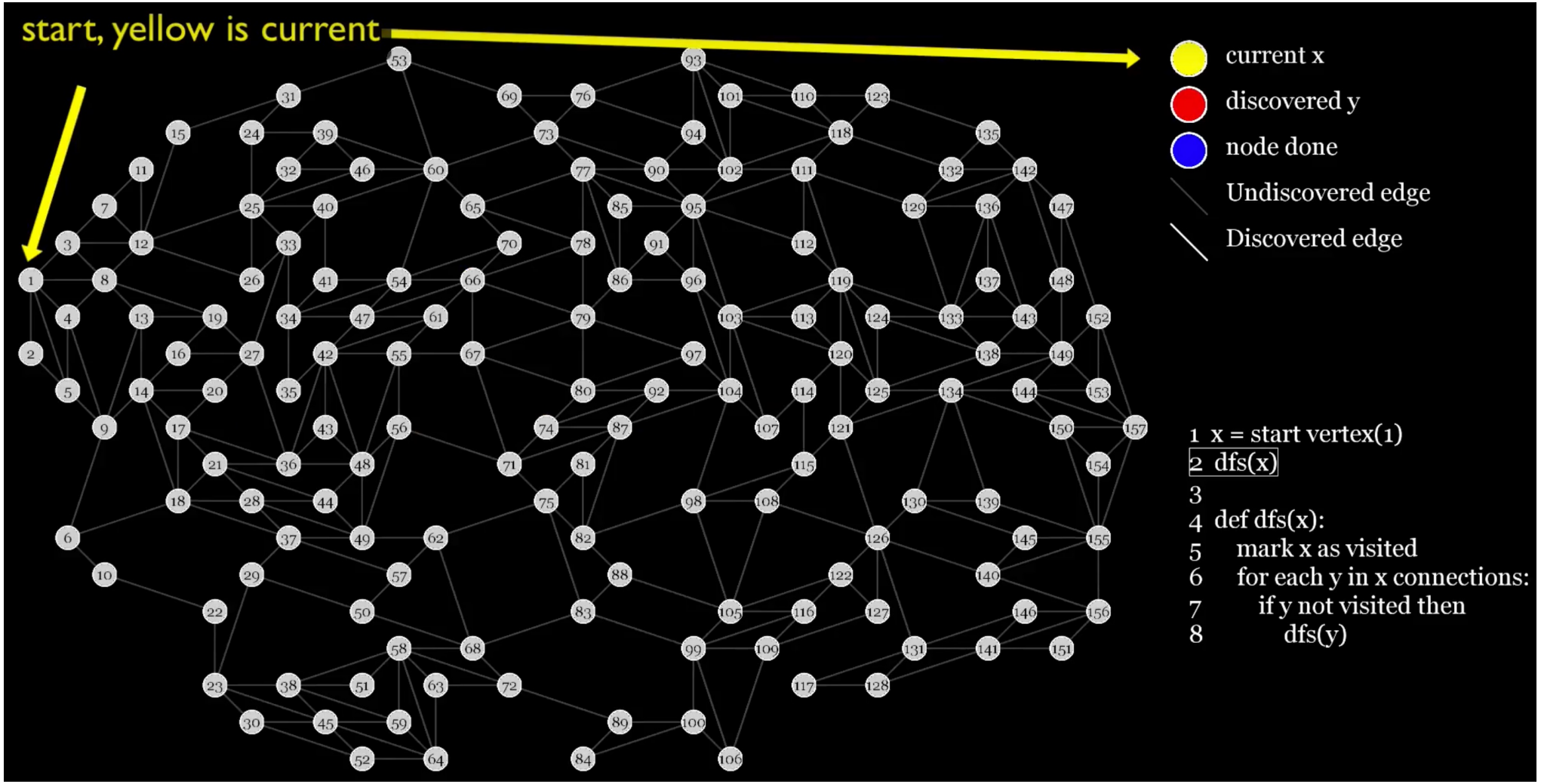
Recorrido en profundidad

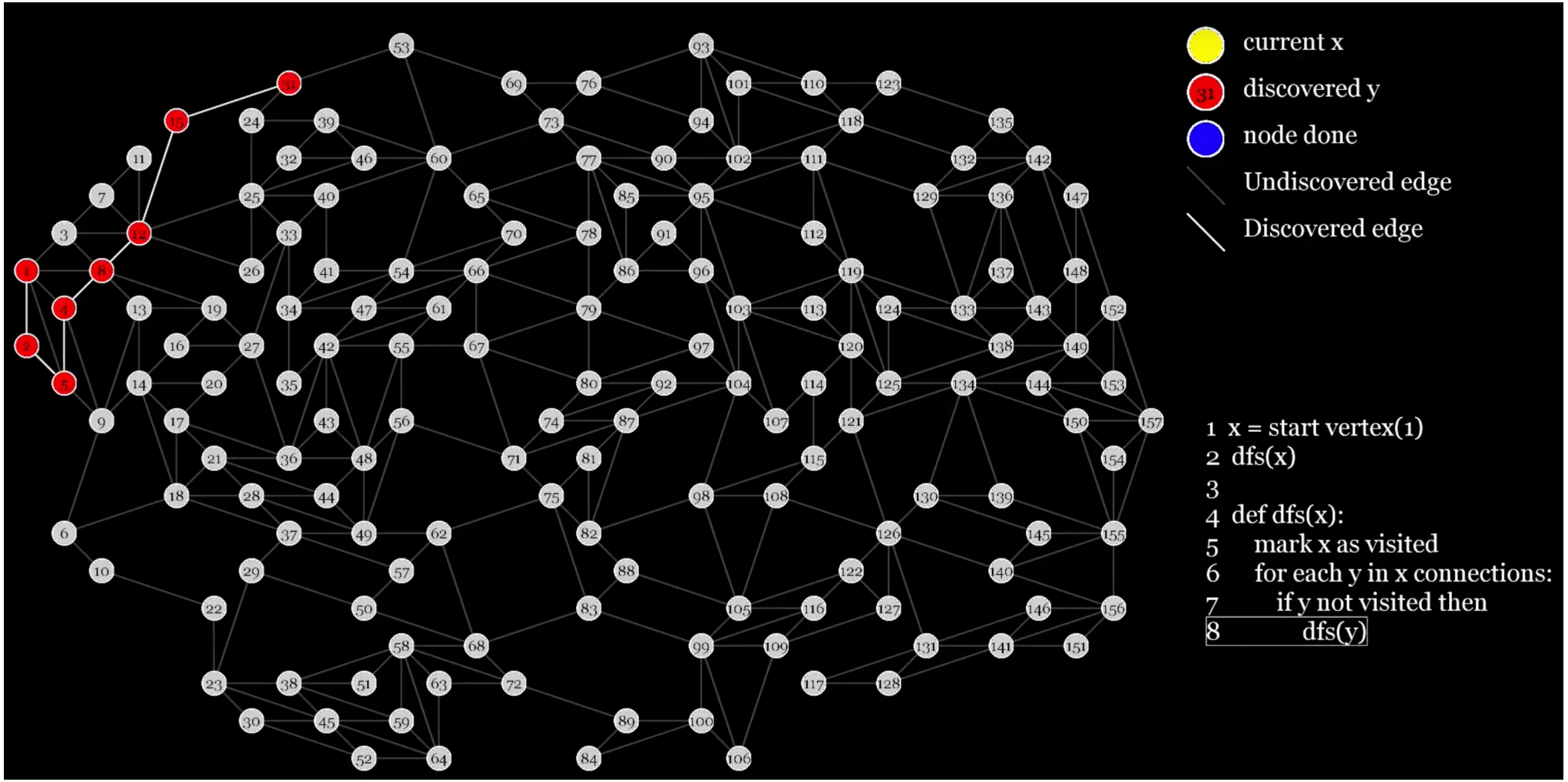
- *Depth-First Search (DFS)*
- Exploración agresiva
- Se programa con una pila (o con recursividad).

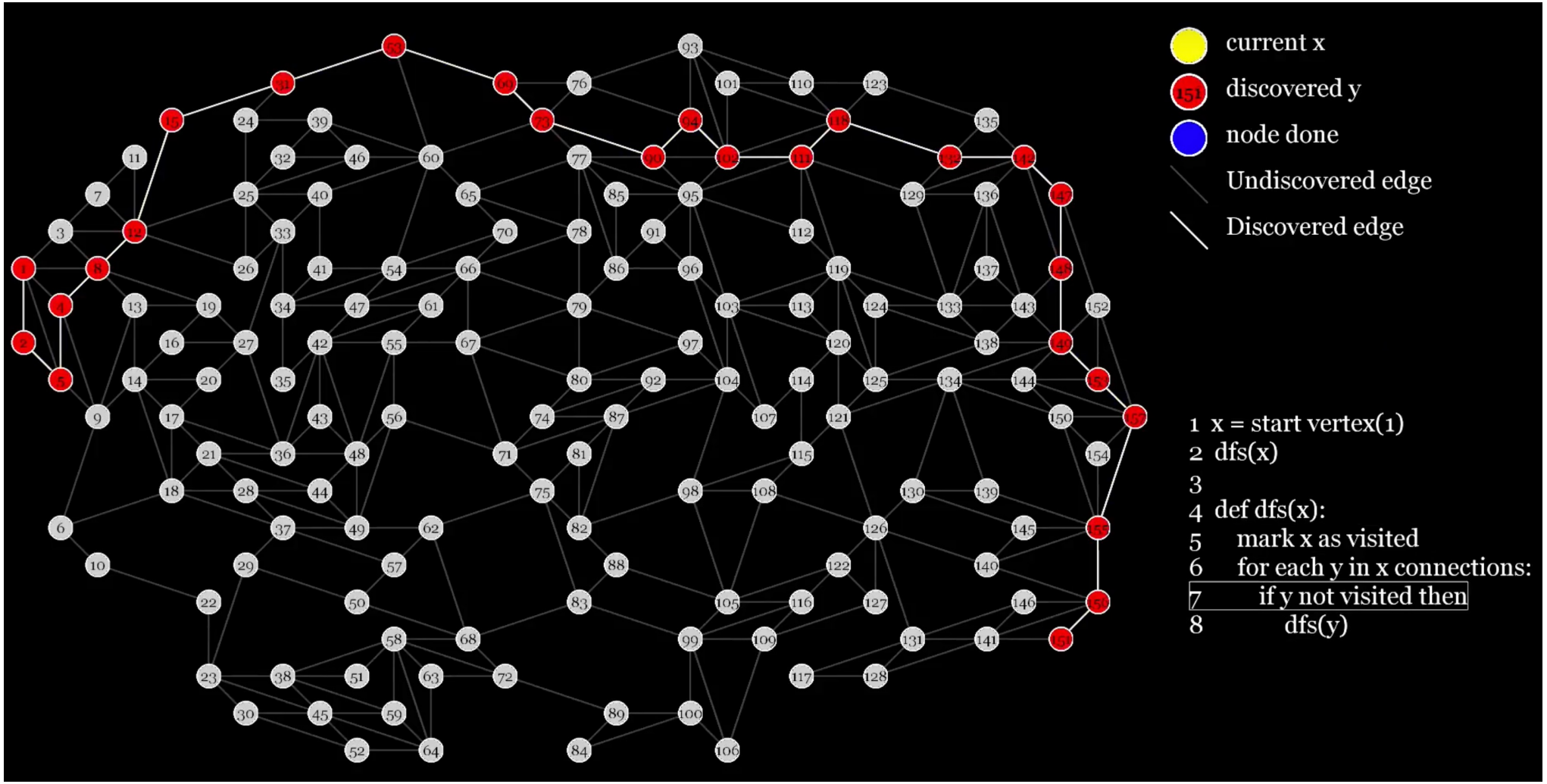
Aplicaciones:

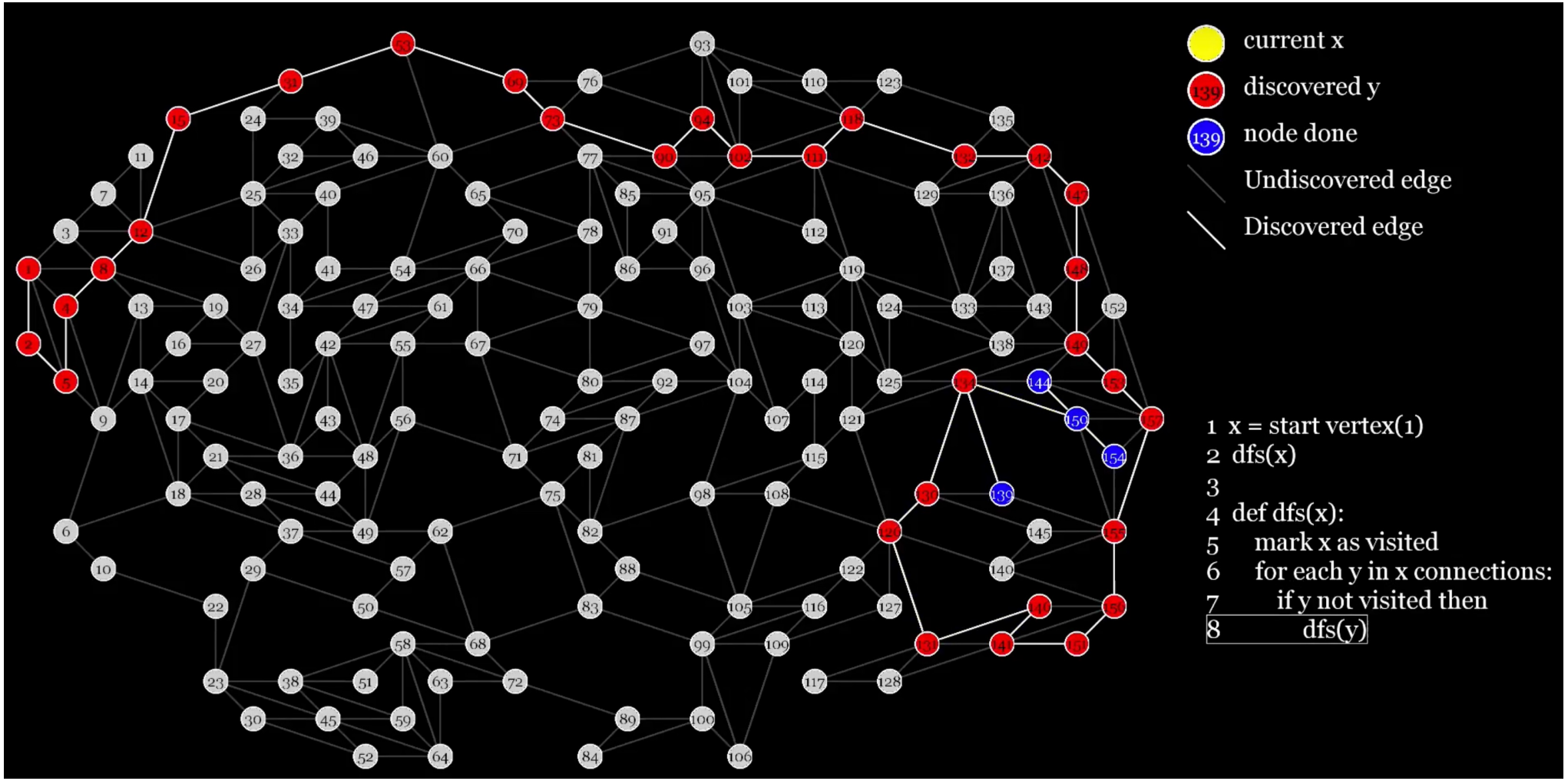
- Orden topológico
- Componentes conectados

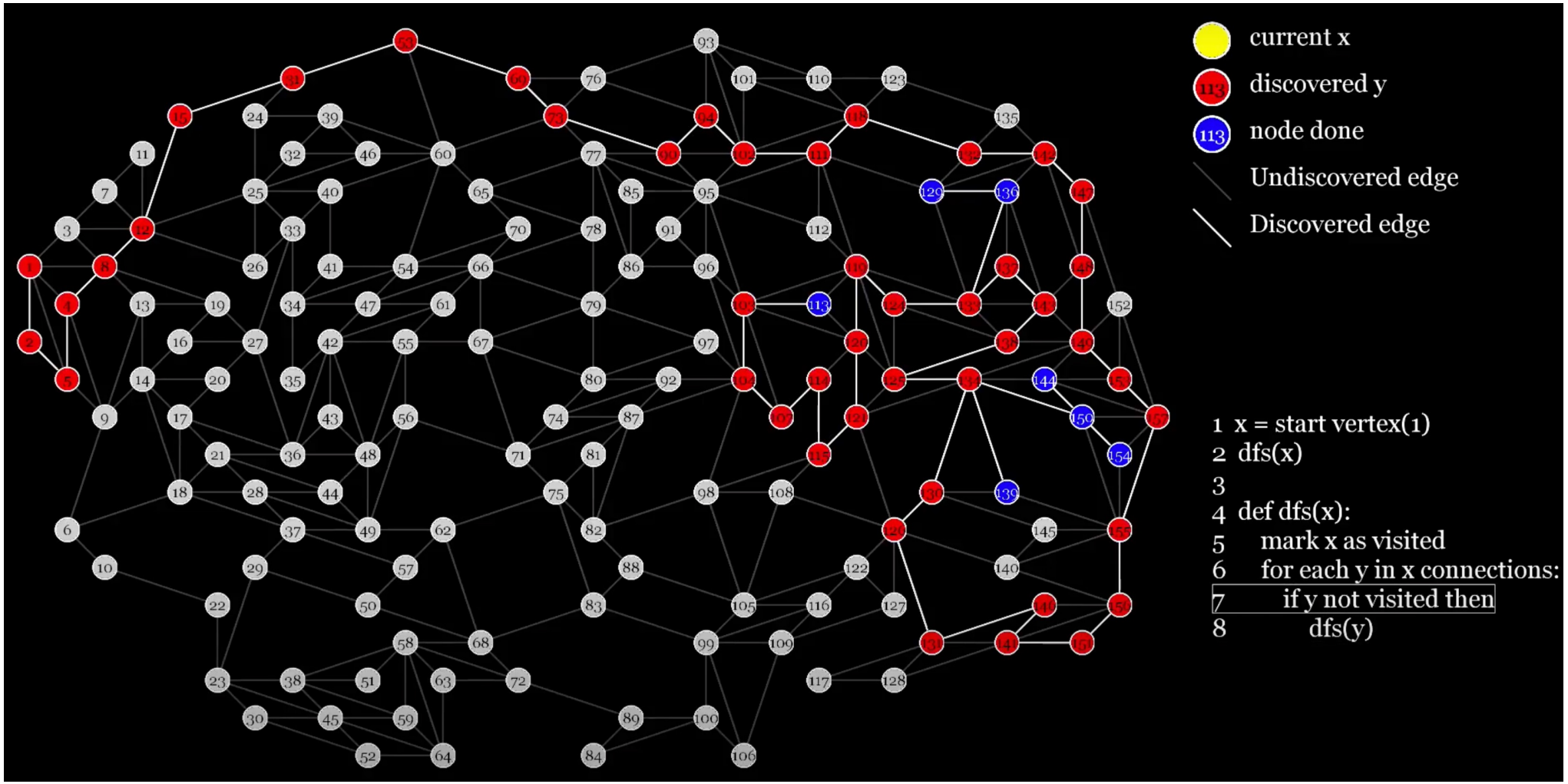
https://www.youtube.com/watch?v=NUgMa5coCoE


start, yellow is current
current x
discovered y
node done
Undiscovered edge
Discovered edge
1 x = start vertex(1)
2 dfs(x)
3
4 def dfs(x):
5 mark x as visited
6 for each y in x connections:
7 if y not visited then
8 dfs(y)


current x
discovered y
node done
Undiscovered edge
Discovered edge
1 x = start vertex(1)
2 dfs(x)
3
4 def dfs(x):
5 mark x as visited
6 for each y in x connections:
7 if y not visited then
8 dfs(y)


current x
151
discovered y
node done
Undiscovered edge
Discovered edge
1 x = start vertex(1)
2 dfs(x)
3
4 def dfs(x):
5 mark x as visited
6 for each y in x connections:
7 if y not visited then
8 dfs(y)


current x
139
discovered y
139
node done
Undiscovered edge
Discovered edge
1 x = start vertex(1)
2 dfs(x)
3
4 def dfs(x):
5 mark x as visited
6 for each y in x connections:
7 if y not visited then
8 dfs(y)


current x
discovered y
node done
Undiscovered edge
Discovered edge
1 x = start vertex(1)
2 dfs(x)
3
4 def dfs(x):
5 mark x as visited
6 for each y in x connections:
7 if y not visited then
8 dfs(y)

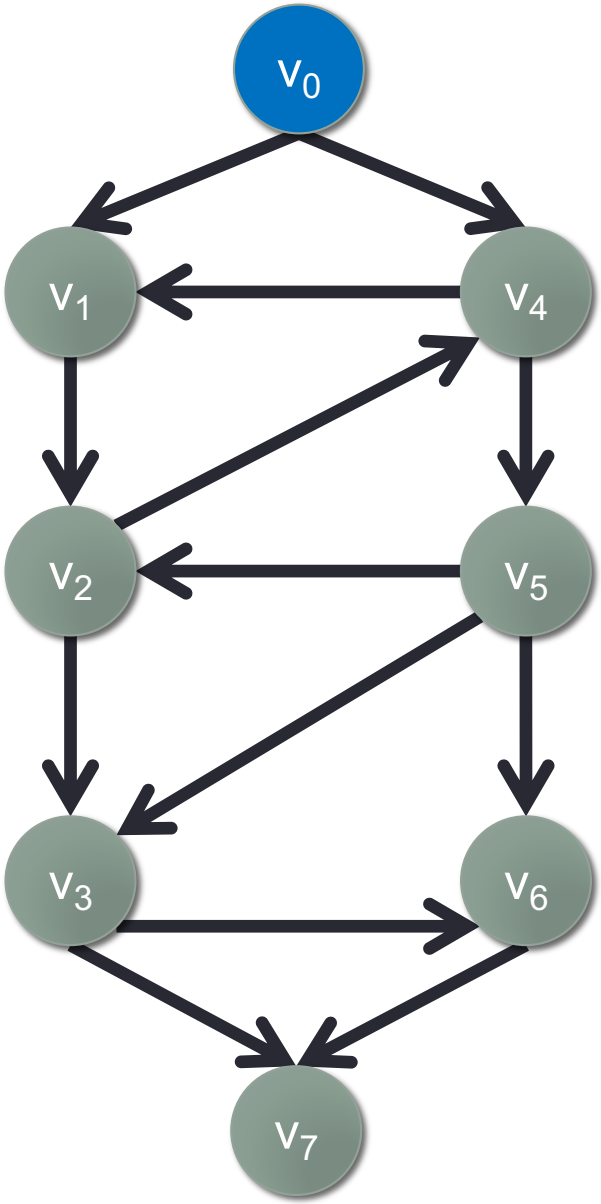
Recorrido en Profundidad (Pila)

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

```
obj = Stack()
obj.push(1)
obj.push(2)


print(obj.peek())    # 2
obj.pop()
print(obj.peek())    # 1
obj.pop()
print(obj.isEmpty()) # True
```

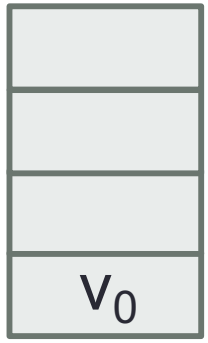
LIFO – Last In First Out

Recorrido en profundidad (*DFS*)



Marcamos el vértice origen como visible
… y lo añadimos a la pila

Pila



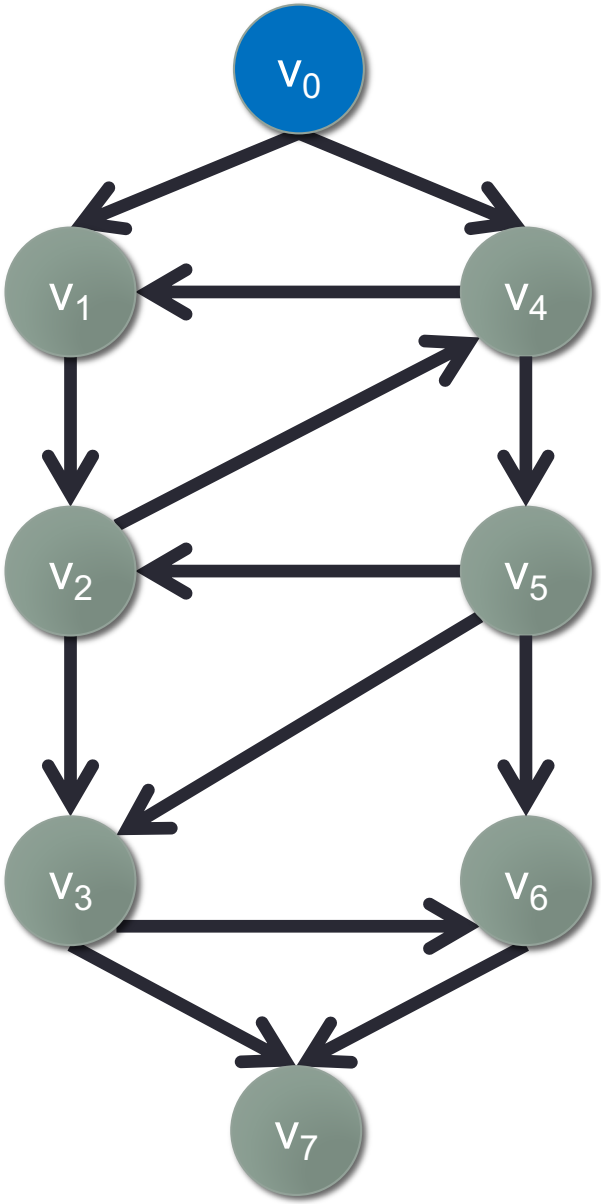
Vértices pendientes de procesar

Vértices visibles

v_0							

Recorrido en profundidad (*DFS*)

Paso 1



v_0

Sacamos un elemento de la pila

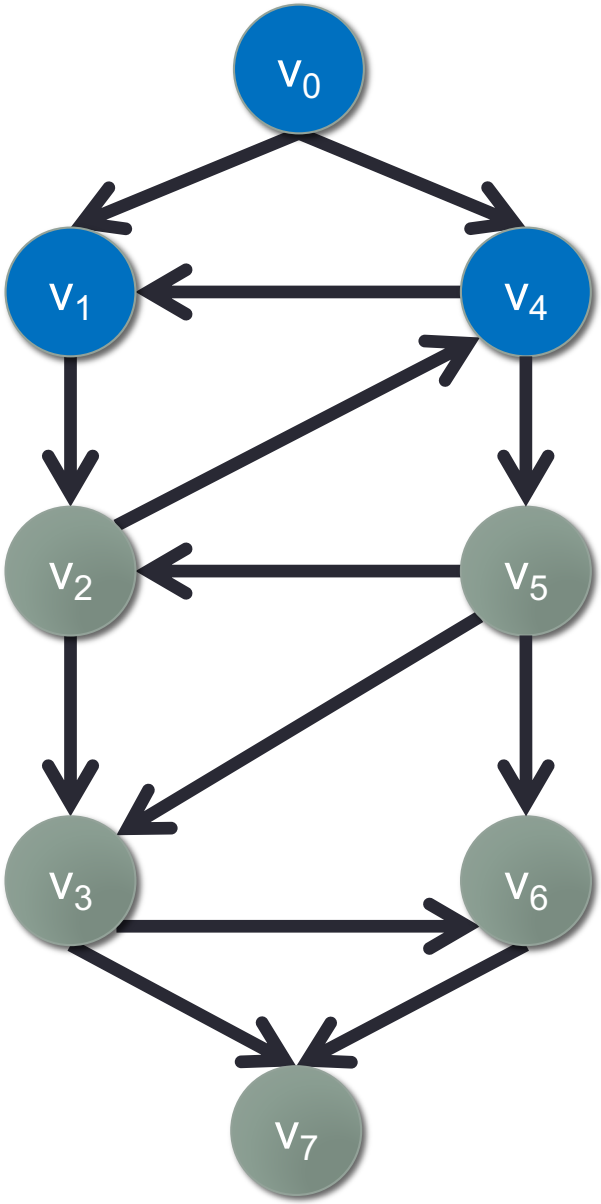
Pila

Vértices pendientes de procesar

Vértices visibles

v_0							

Recorrido en profundidad (*DFS*)



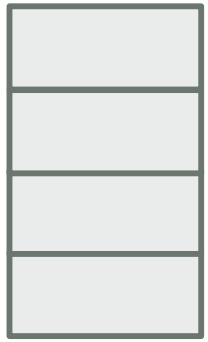
V_0

Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

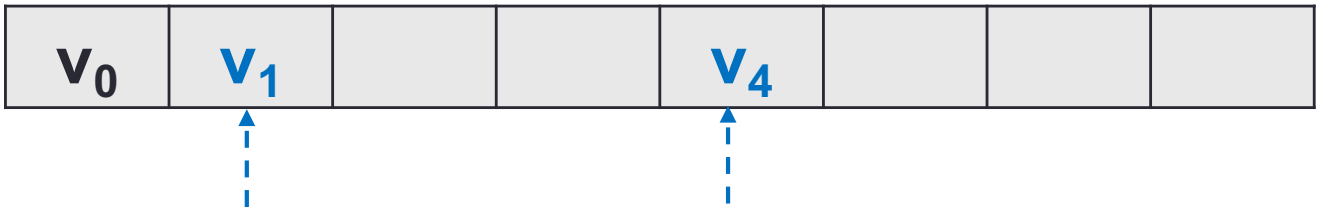
V_1 y V_4

Pila



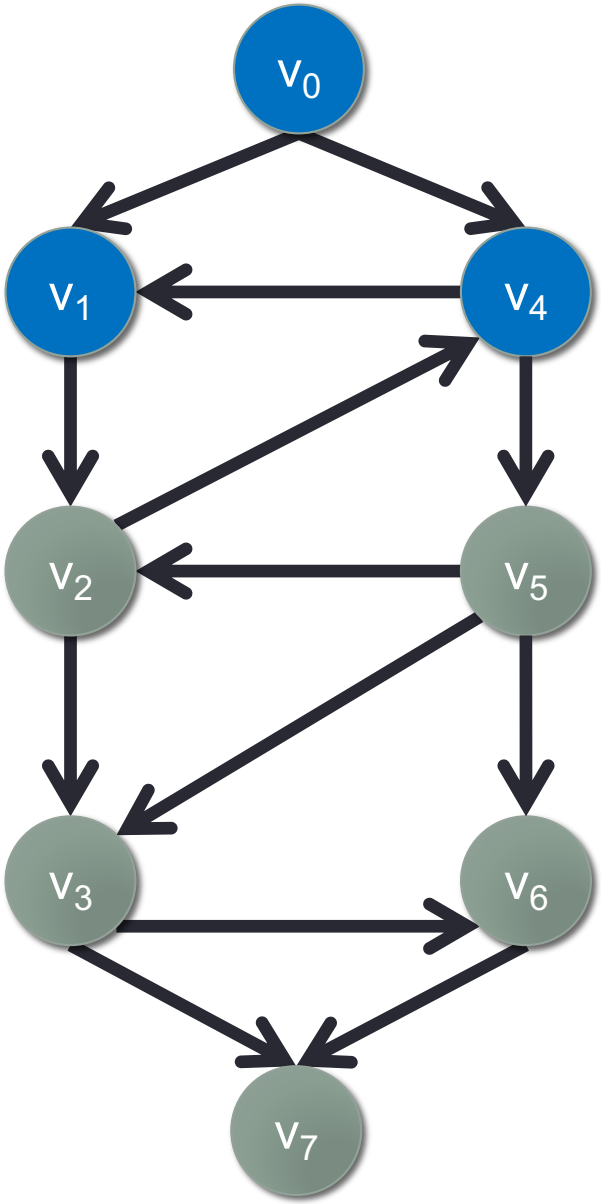
Vértices pendientes de procesar

Vértices visibles



Recorrido en profundidad (*DFS*)

Paso 3



v_0

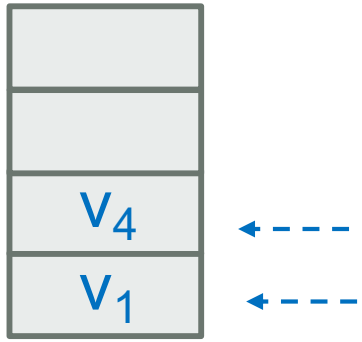
Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

v_1 y v_4

… y los metemos en la pila para procesarlos

Pila



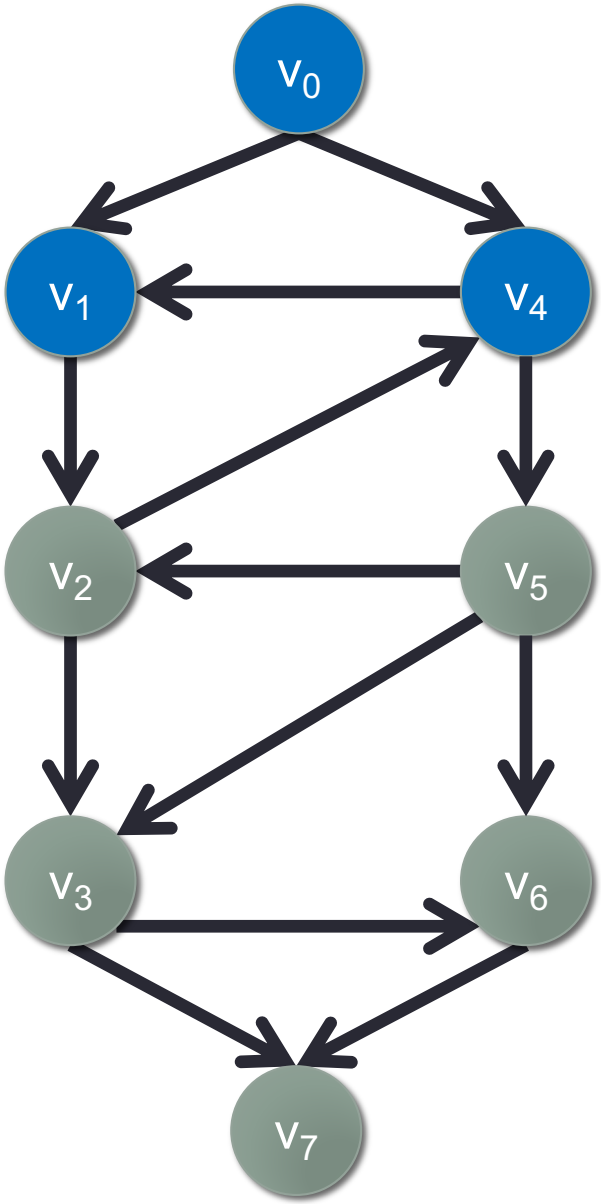
Vértices pendientes de procesar

Vértices visibles

v_0	**v_1**			**v_4**			

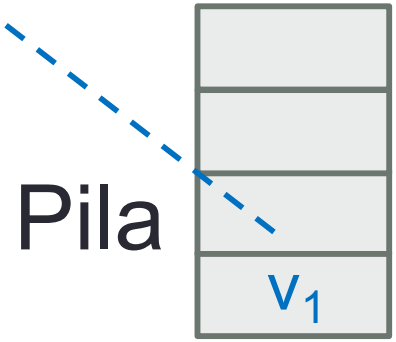
Recorrido en profundidad (*DFS*)

Paso 1



v_0 v_4

Sacamos un elemento de la pila

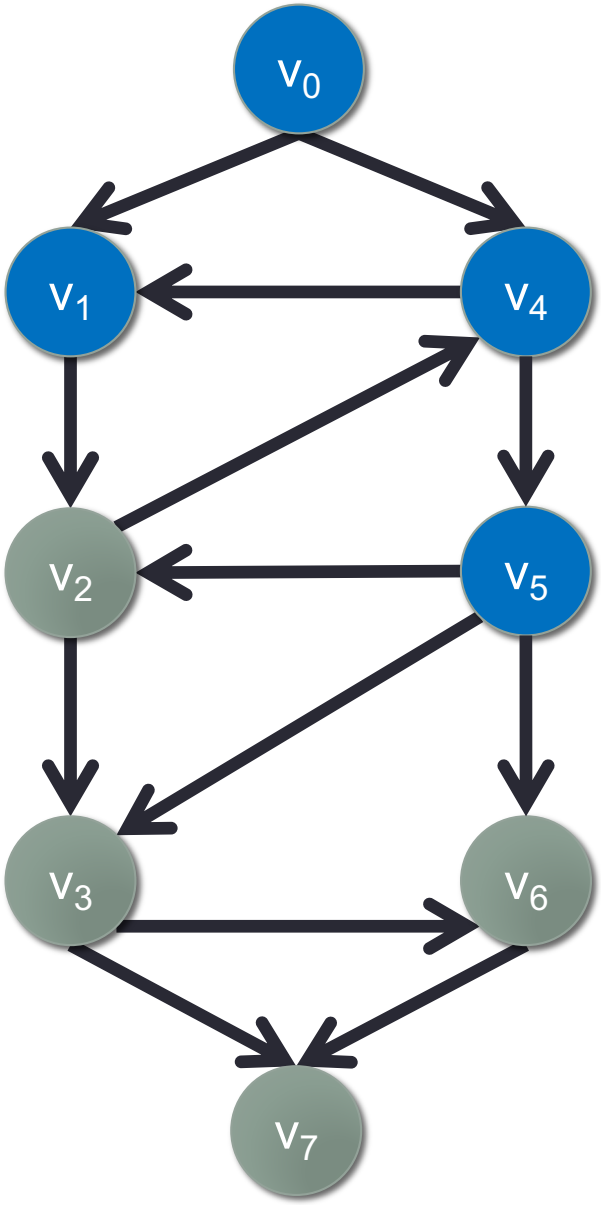


Vértices pendientes de procesar

Vértices visibles

v_0	**v_1**			**v_4**			

Recorrido en profundidad (*DFS*)



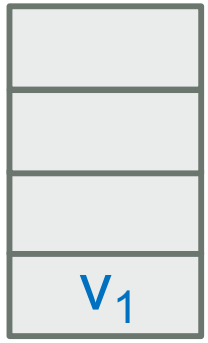
V_0 V_4

Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

V_5

Pila

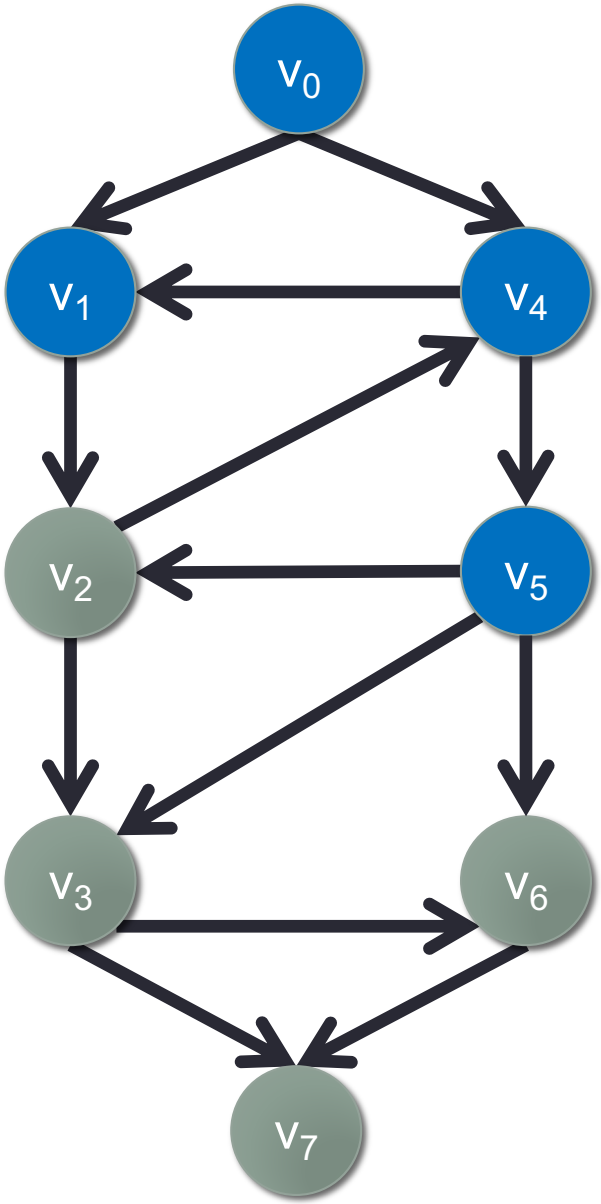


Vértices pendientes de procesar

Vértices visibles

Recorrido en profundidad (*DFS*)

Paso 3



V_0 V_4

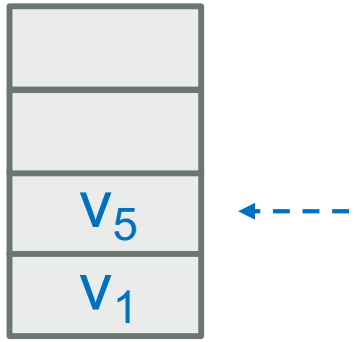
Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

V_5

… y los metemos en la pila para procesarlos

Pila



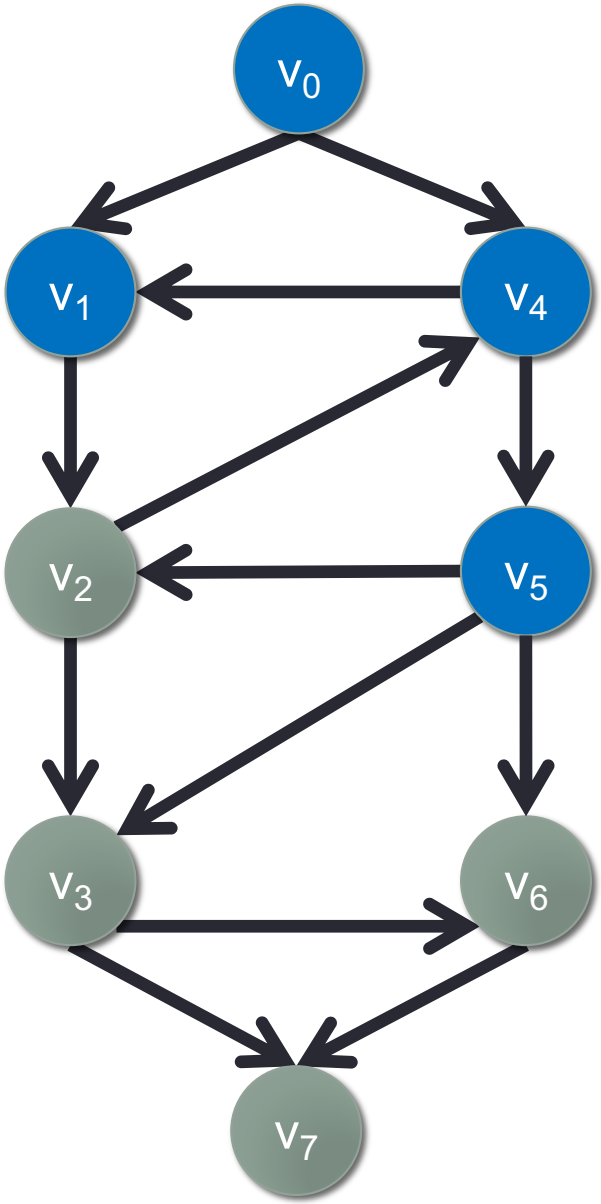
Vértices pendientes de procesar

Vértices visibles

v_0	**v_1**			**v_4**	**v_5**		

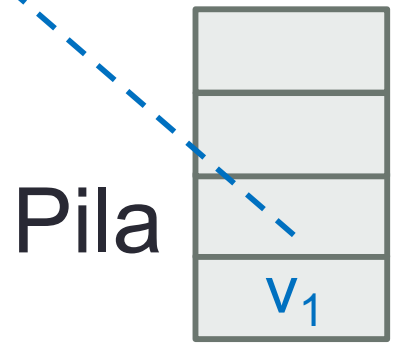
Recorrido en profundidad (*DFS*)

Paso 1



V_0 V_4 V_5

Sacamos un elemento de la pila



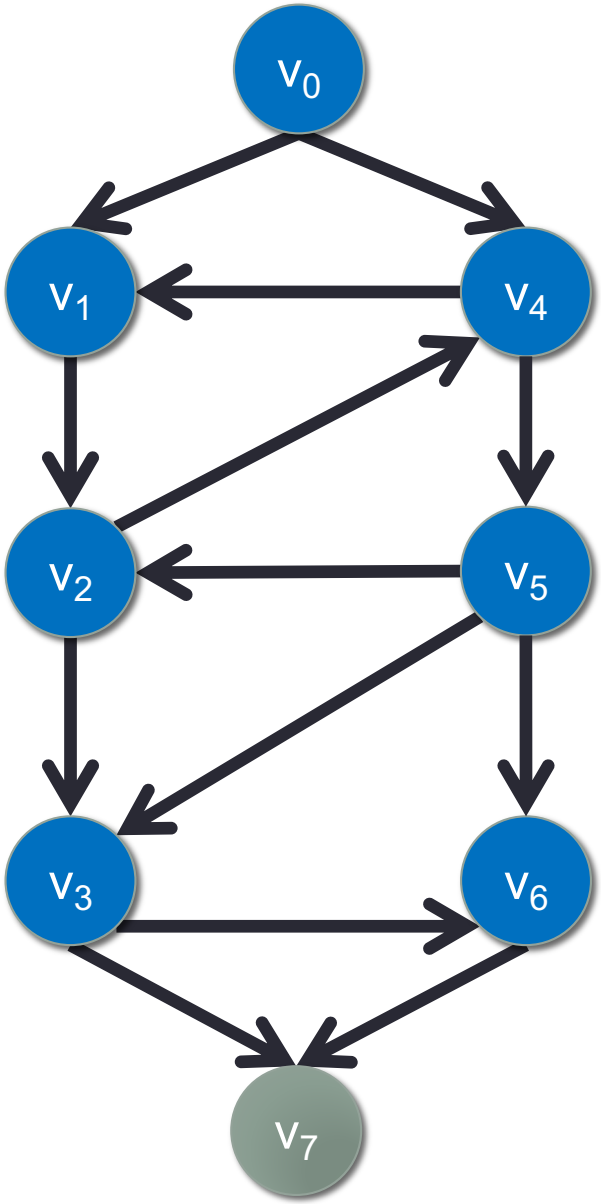
Vértices pendientes de procesar

Vértices visibles

v_0	**v_1**			**v_4**	**v_5**		

Recorrido en profundidad (*DFS*)

Paso 2



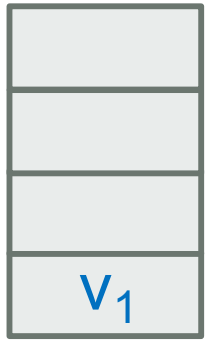
V_0 V_4 V_5

Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

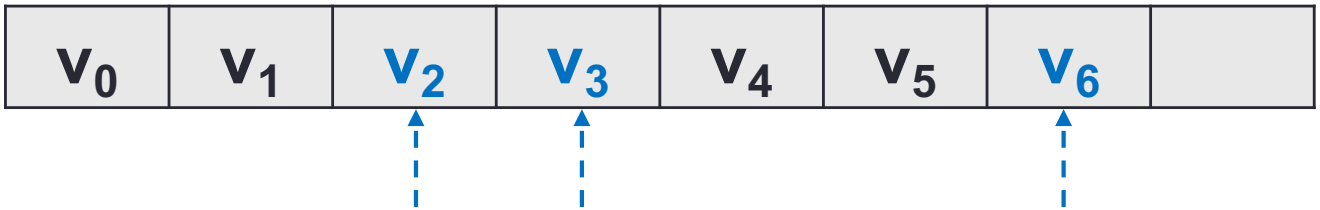
V_2 V_3 V_6

Pila



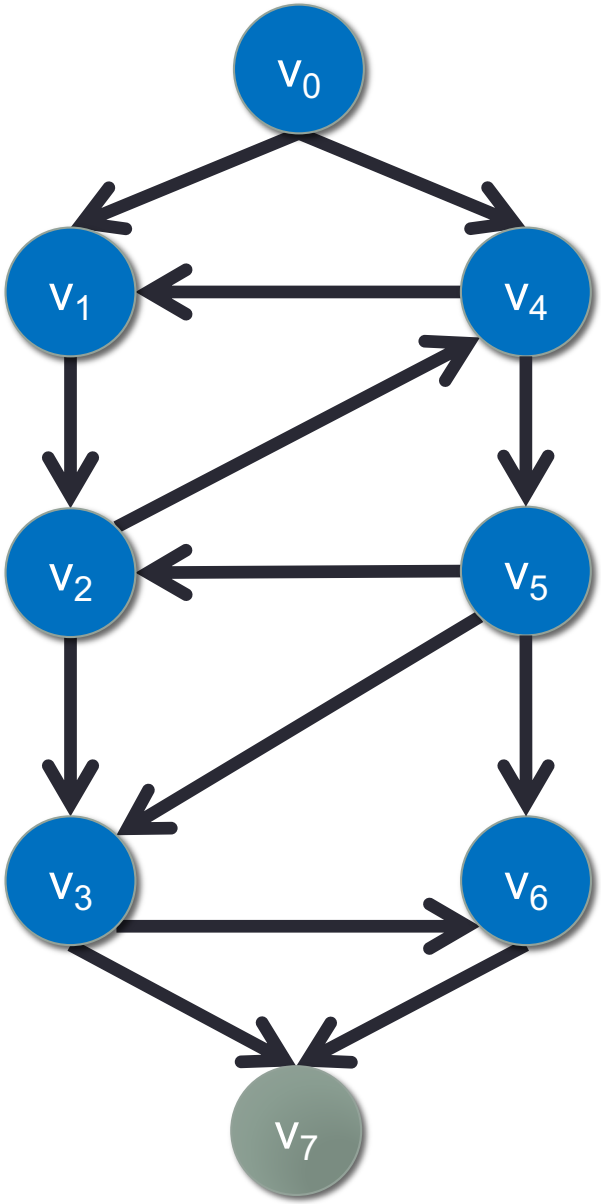
Vértices pendientes de procesar

Vértices visibles



Recorrido en profundidad (*DFS*)

Paso 3



V_0 V_4 V_5

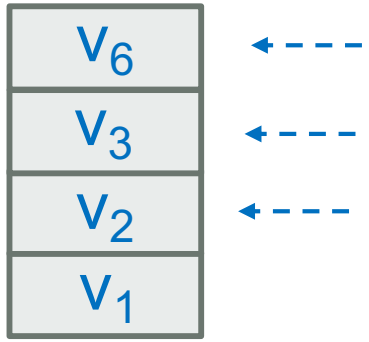
Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

V_2 V_3 V_6

… y los metemos en la pila para procesarlos

Pila



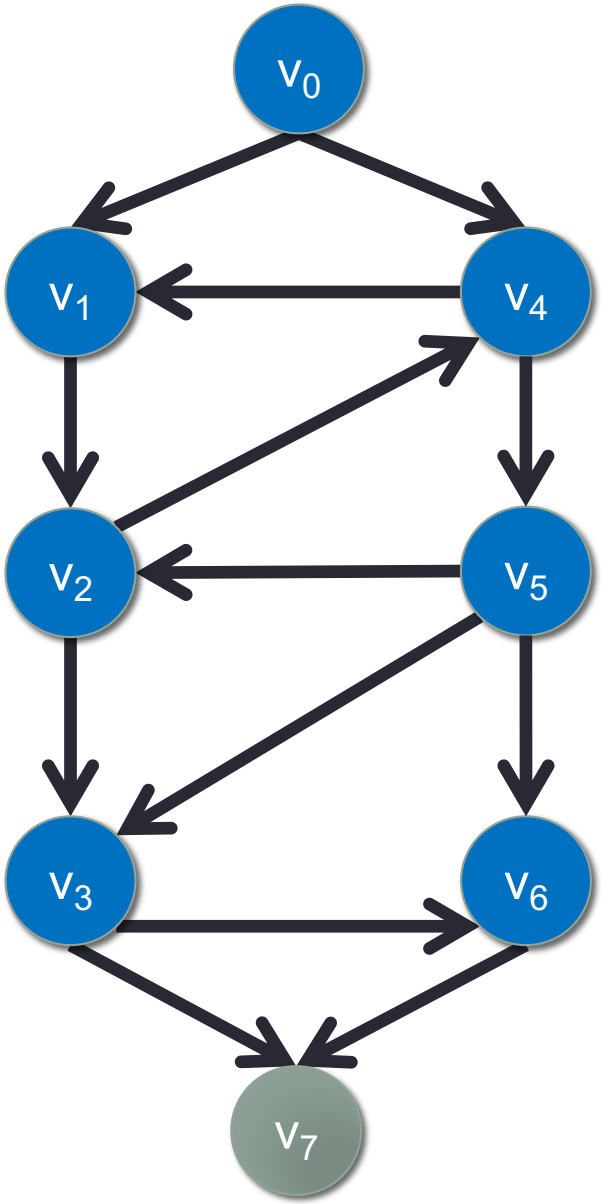
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	

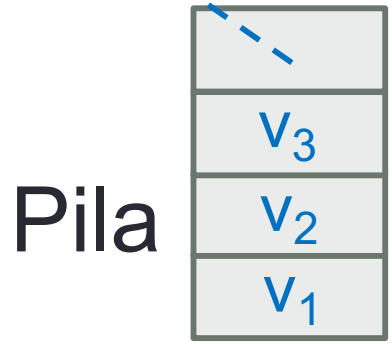
Recorrido en profundidad (*DFS*)

Paso 1



V_0 V_4 V_5 V_6

Sacamos un elemento de la pila



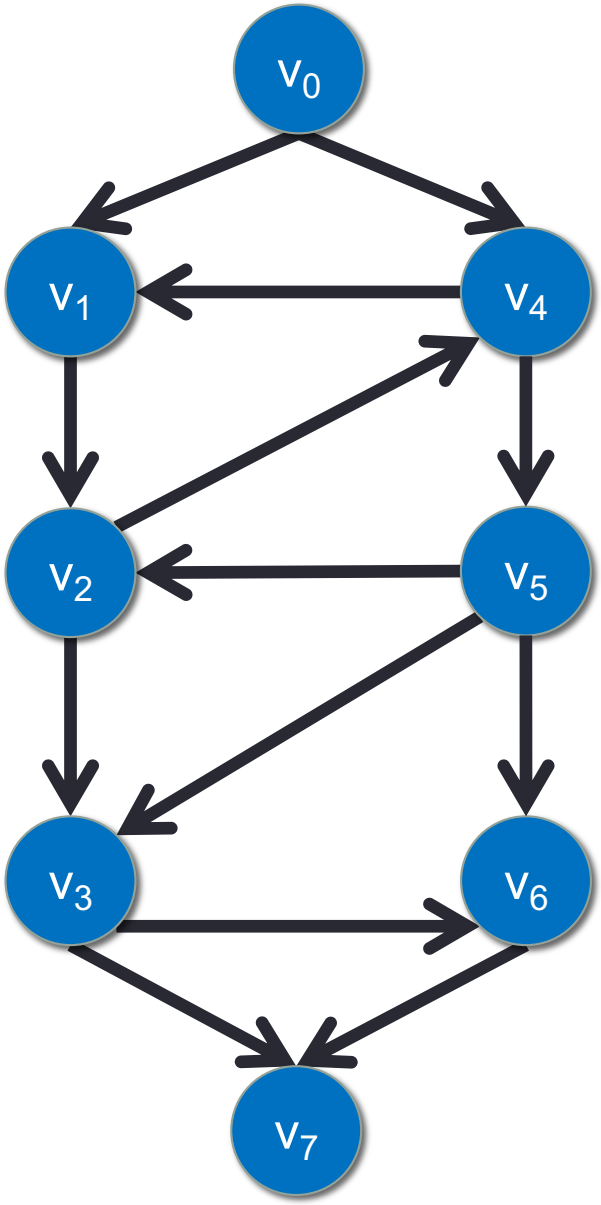
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	

Recorrido en profundidad (*DFS*)

Paso 3



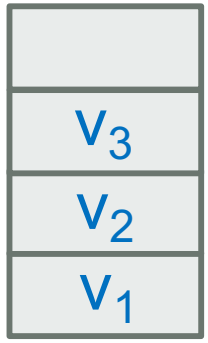
V_0 V_4 V_5 V_6

Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

V_7

Pila



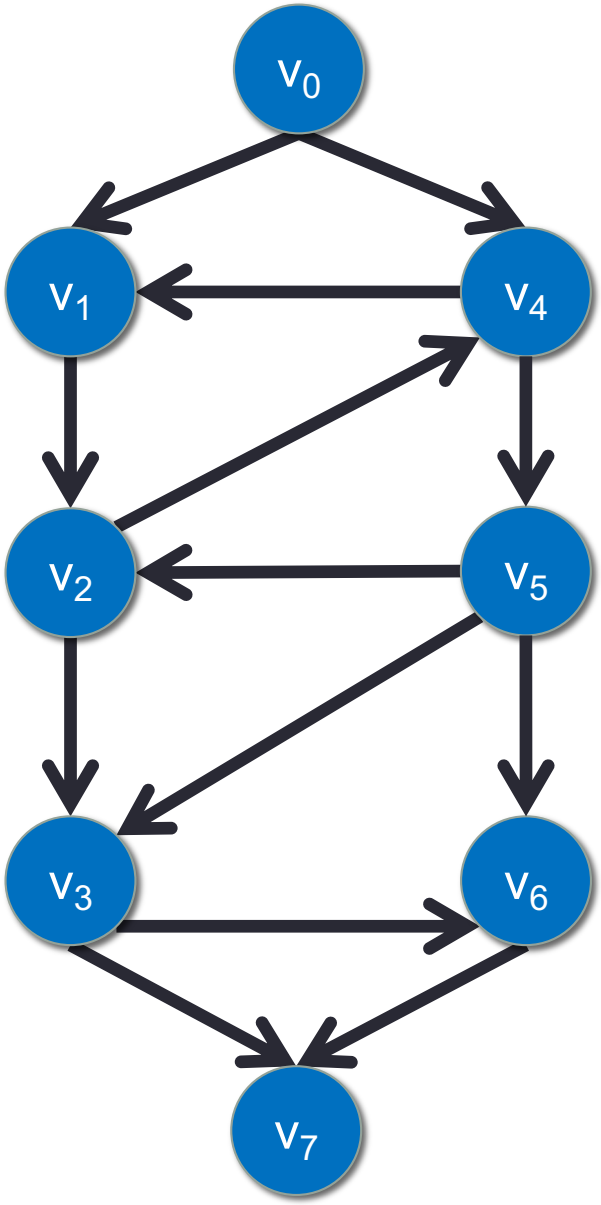
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Recorrido en profundidad (*DFS*)

Paso 3



V_0 V_4 V_5 V_6

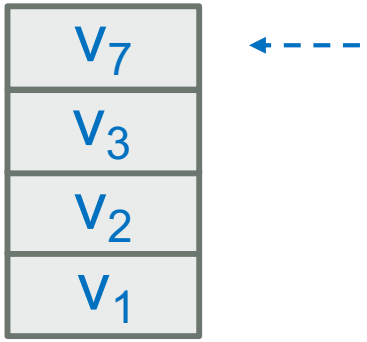
Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

V_7

… y los metemos en la pila para procesarlos

Pila



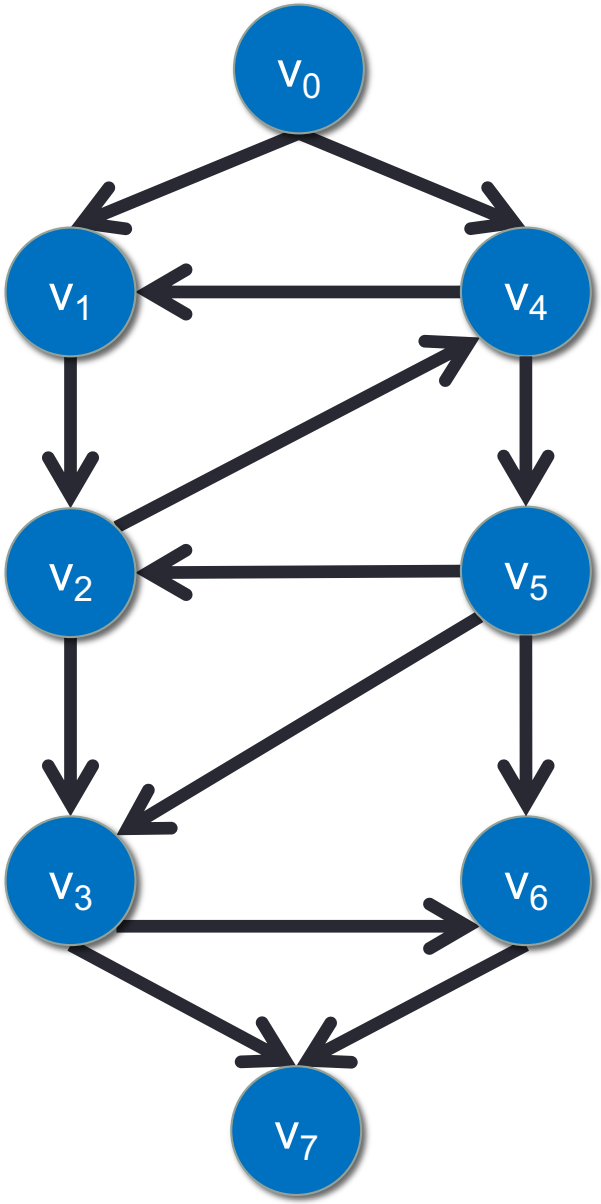
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Recorrido en profundidad (*DFS*)

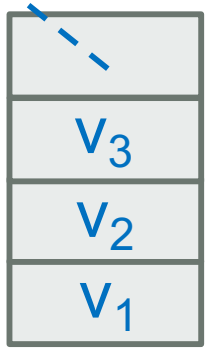
Paso 1



$V_0\ V_4\ V_5\ V_6\ V_7$

Sacamos un elemento de la pila

Pila



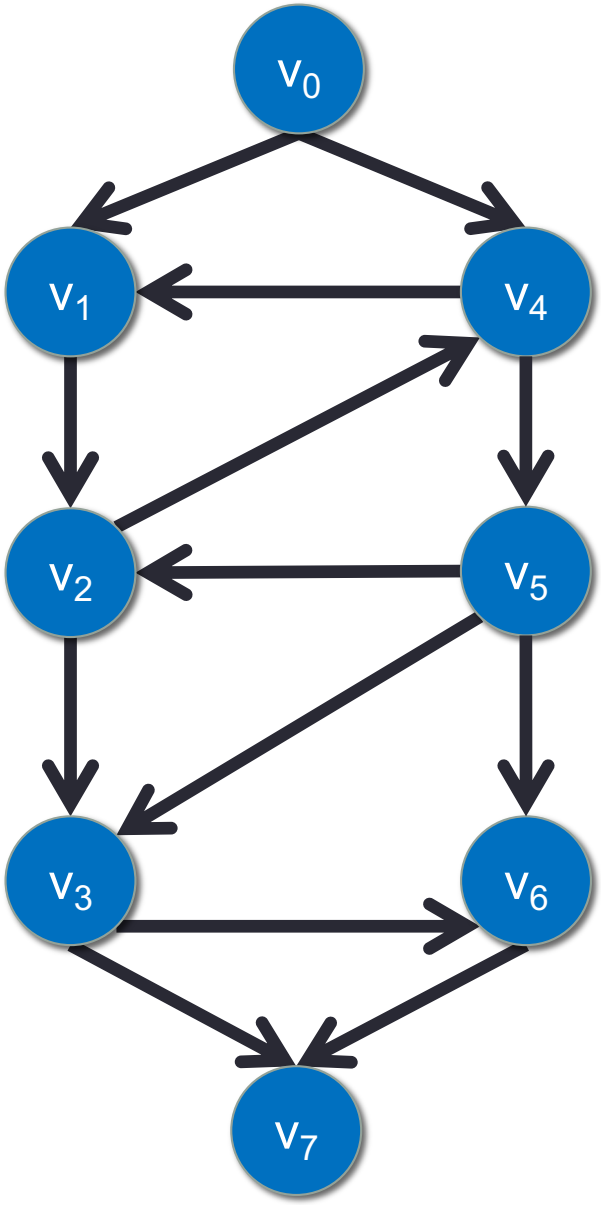
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Recorrido en profundidad (*DFS*)

Paso 2



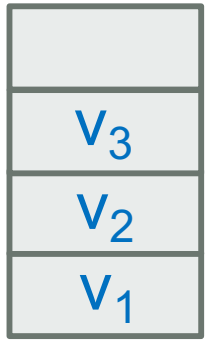
V_0 V_4 V_5 V_6 V_7

Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

<< no hay vecinos no visibles >>

Pila



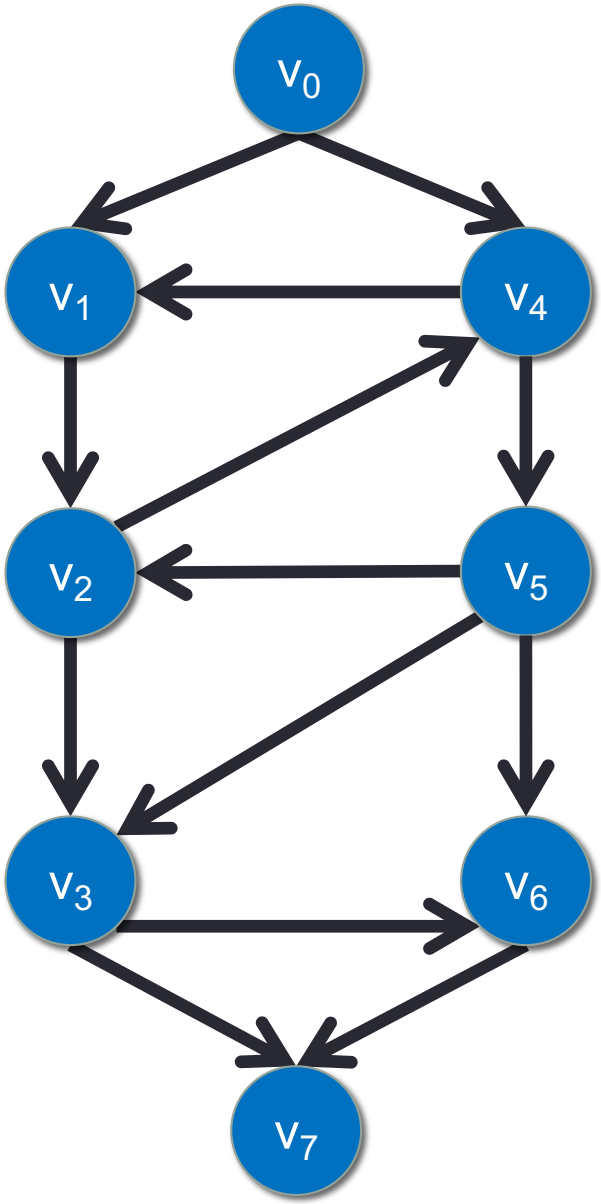
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

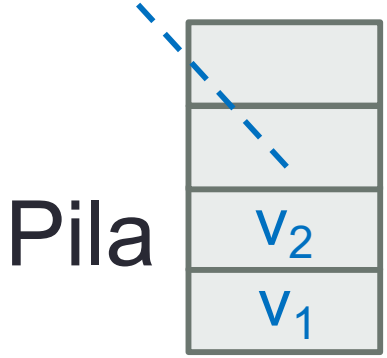
Recorrido en profundidad (*DFS*)

Paso 1



$V_0\ V_4\ V_5\ V_6\ V_7\ V_3$

Sacamos un elemento de la pila



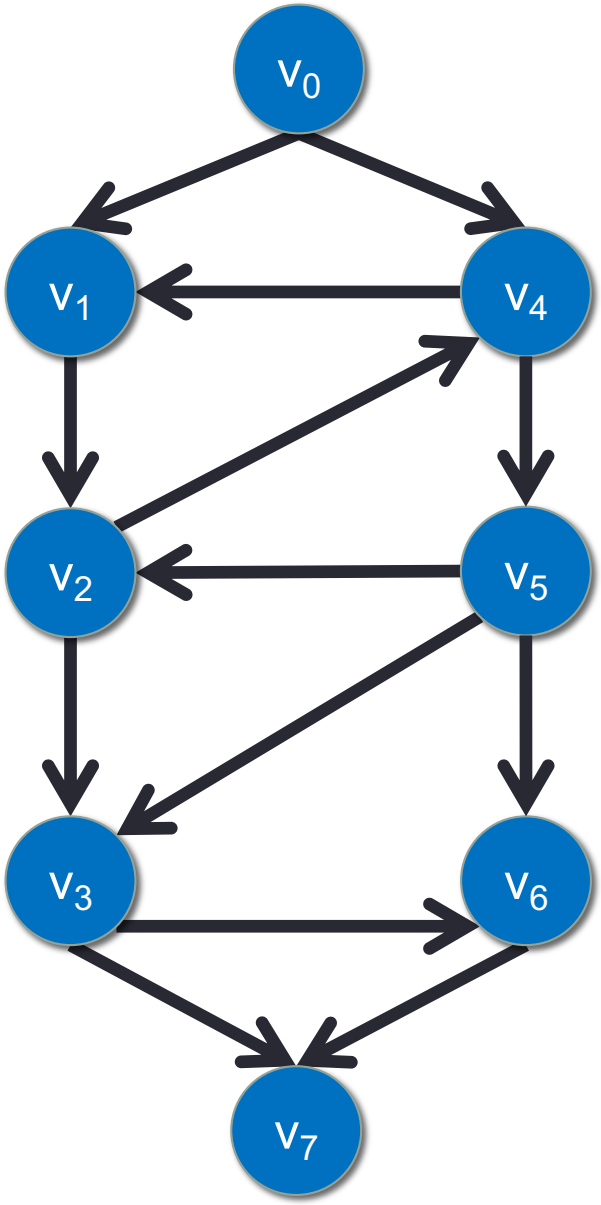
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Recorrido en profundidad (*DFS*)

Paso 2



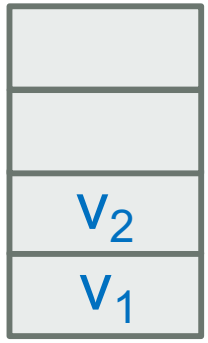
V_0 V_4 V_5 V_6 V_7 V_3

Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

<< no hay vecinos no visibles >>

Pila



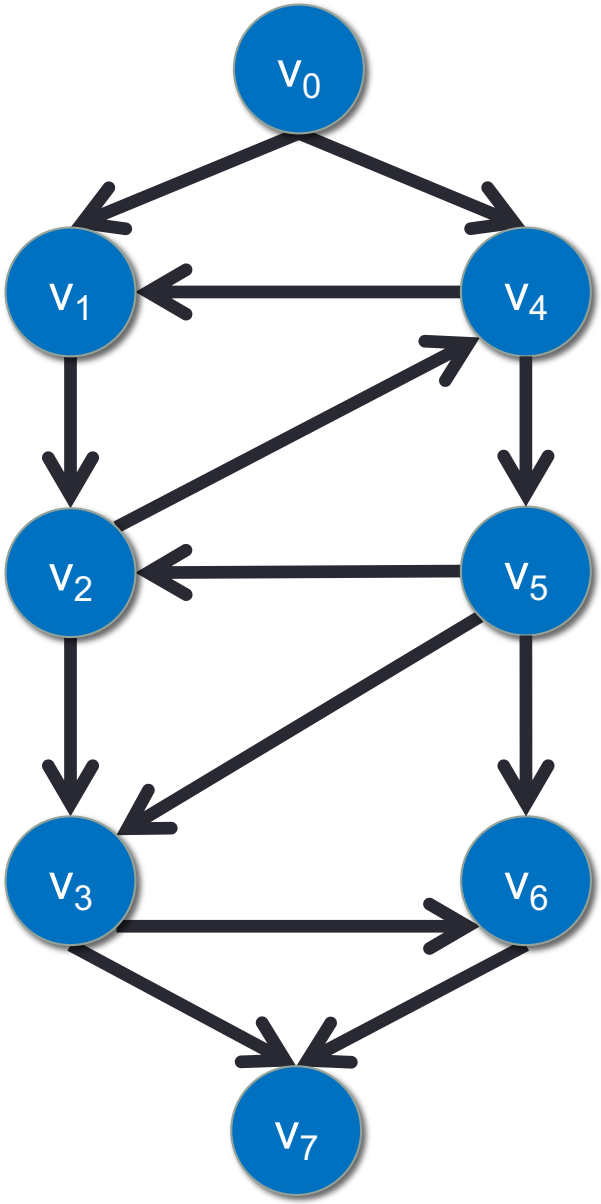
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

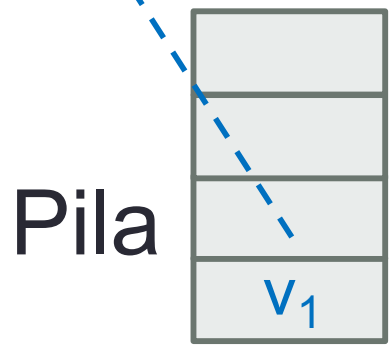
Recorrido en profundidad (*DFS*)

Paso 1



$V_0\ V_4\ V_5\ V_6\ V_7\ V_3\ V_2$

Sacamos un elemento de la pila



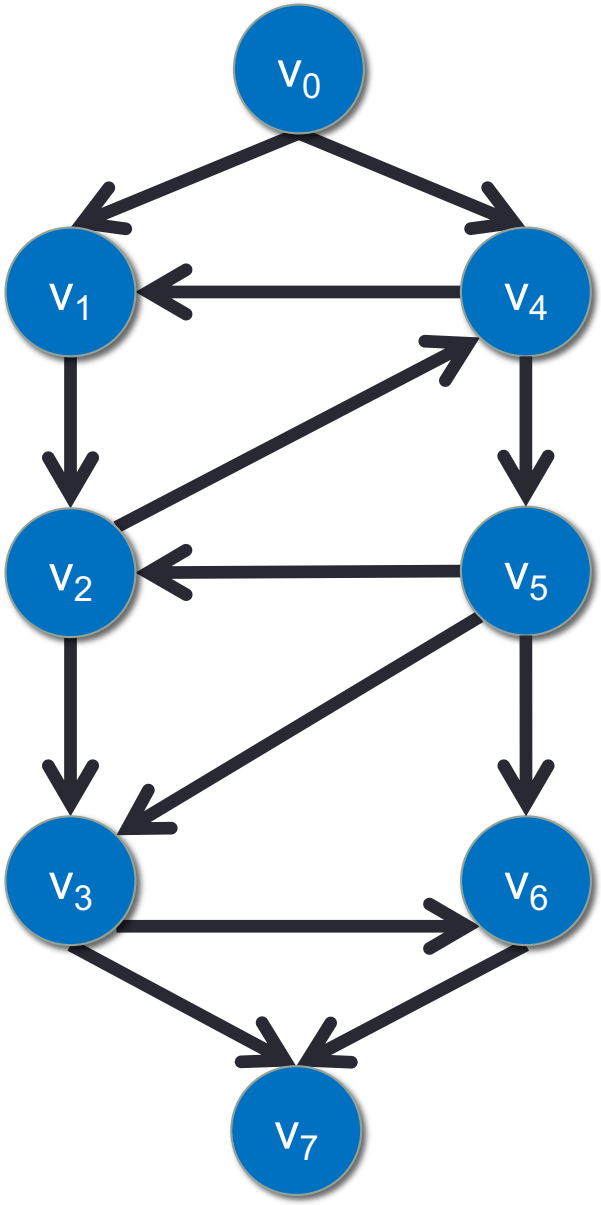
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Recorrido en profundidad (*DFS*)

Paso 2



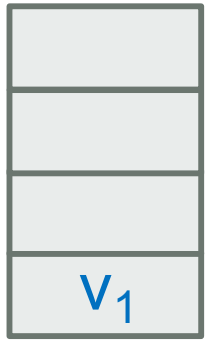
V_0 V_4 V_5 V_6 V_7 V_3 V_2

Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

<< no hay vecinos no visibles >>

Pila



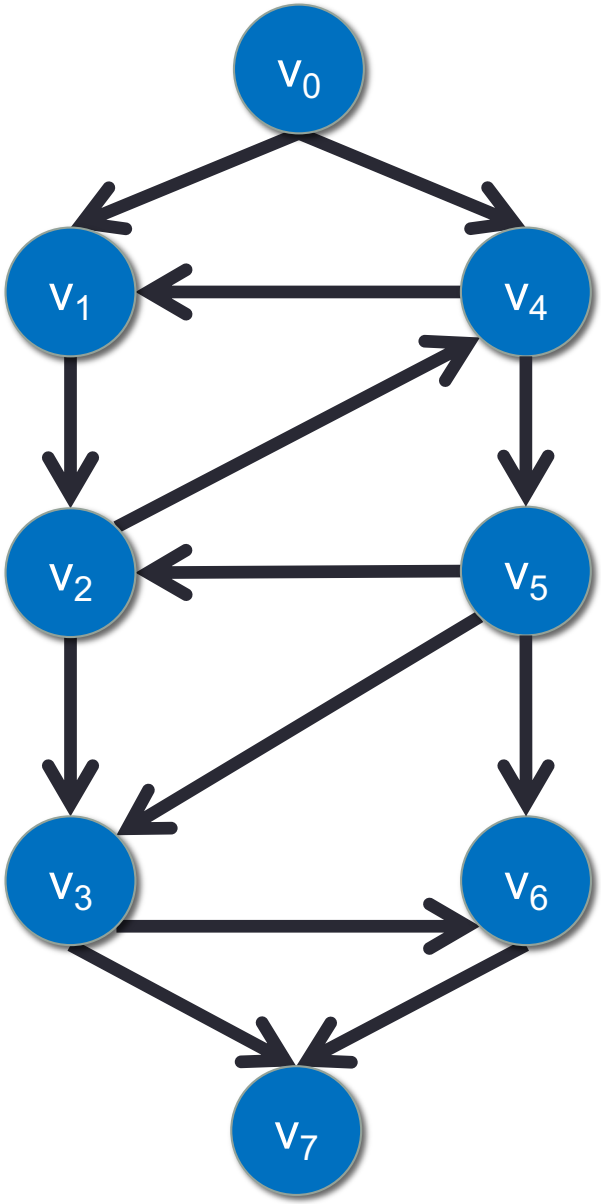
Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

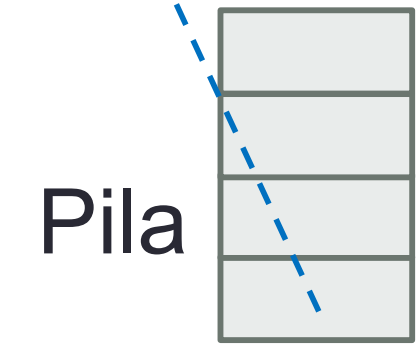
Recorrido en profundidad (*DFS*)

Paso 1



$V_0\ V_4\ V_5\ V_6\ V_7\ V_3\ V_2\ V_1$

Sacamos un elemento de la pila

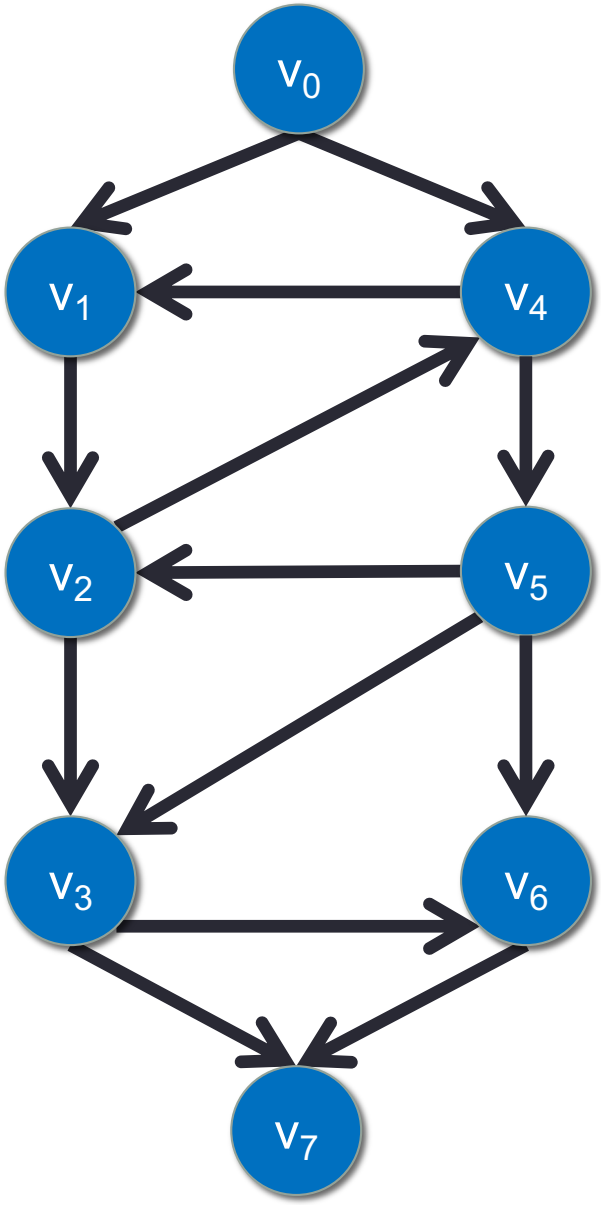


Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Recorrido en profundidad (*DFS*)



V_0 V_4 V_5 V_6 V_7 V_3 V_2 V_1

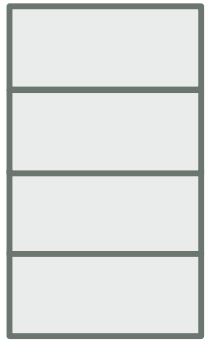
Sacamos un elemento de la pila

… marcamos sus vecinos como visibles:

<< no hay vecinos no visibles >>

… y la pila está vacía.

Pila

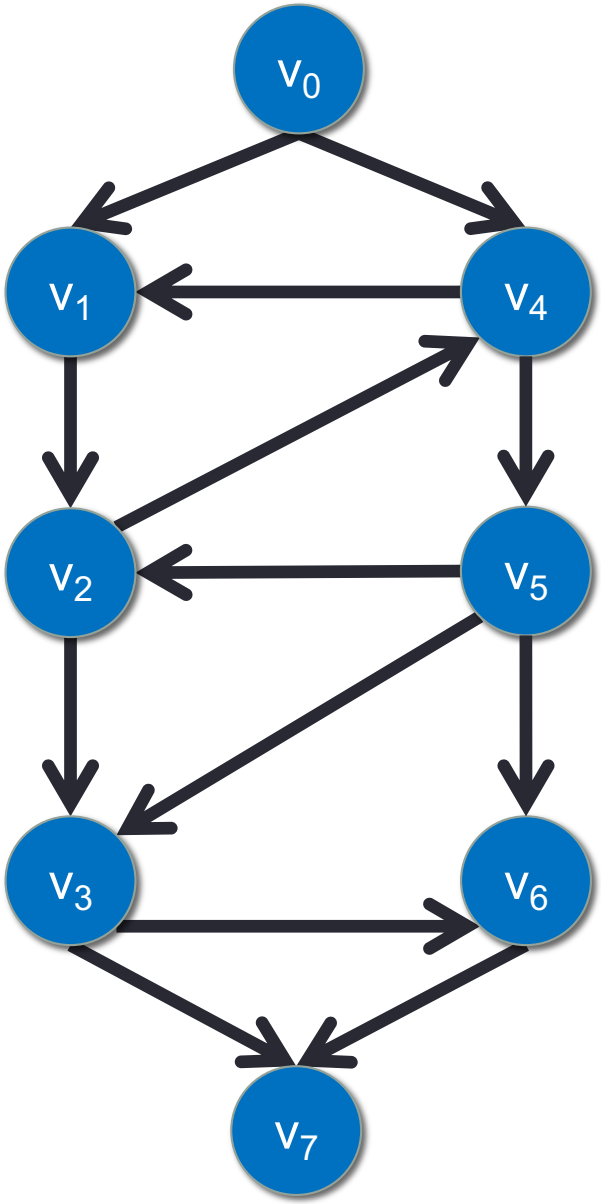


Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Recorrido en profundidad (*DFS*)



$V_0\ V_4\ V_5\ V_6\ V_7\ V_3\ V_2\ V_1$

… recorrido en profundidad

Recorrido en profundidad (DFS)

Depth_First_Search (grafo G, vertice inicial vi)

- S = Stack()
- Marcar vi como visible;
- S.push(vi)

- Mientras not S.isEmpty ():
  - v = S.pop()

  - Para todas las aristas (v,w):
    - Si w no es visible:
      - Marcar w como visible
      - S.push(w)

Algoritmo iterativo

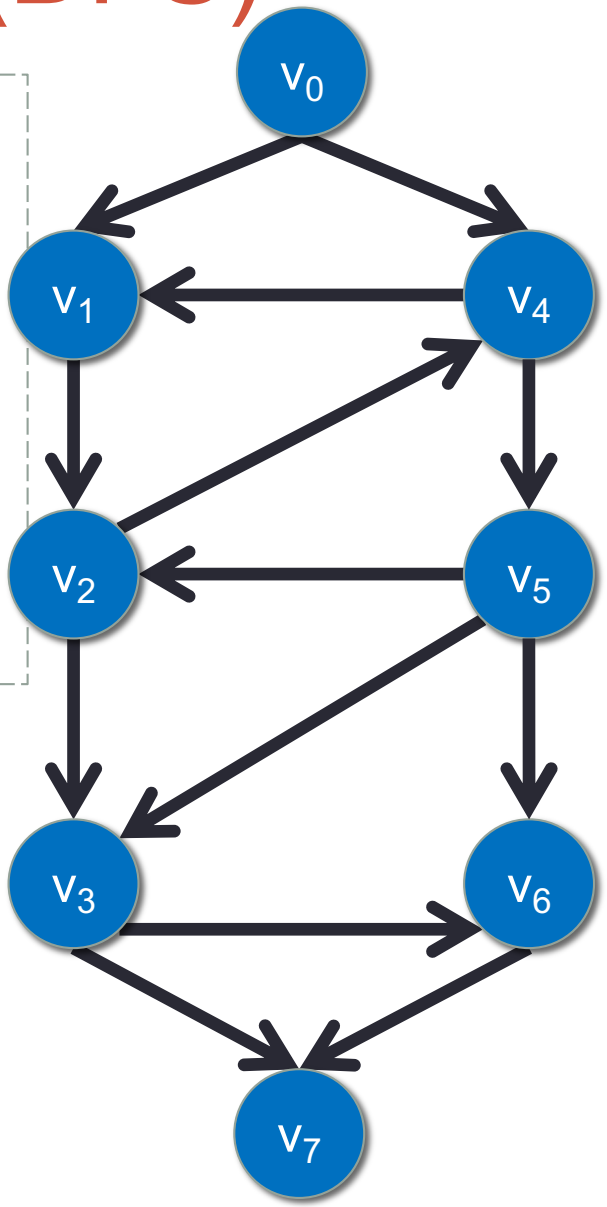
Recorrido en profundidad (DFS)

Depth_First_Search (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - Depth_First_Search (G, w)

Algoritmo recursivo



Estrategias básicas de búsqueda

Recorrido a lo ancho

- *Breadth-First Search (BFS)*
- Exploración por niveles
- Se programa con una cola

Aplicaciones:

- Camino más corto
- Componentes conectados

Recorrido en profundidad

- *Depth-First Search (DFS)*
- Exploración agresiva
- Se programa con una pila (o con recursividad).

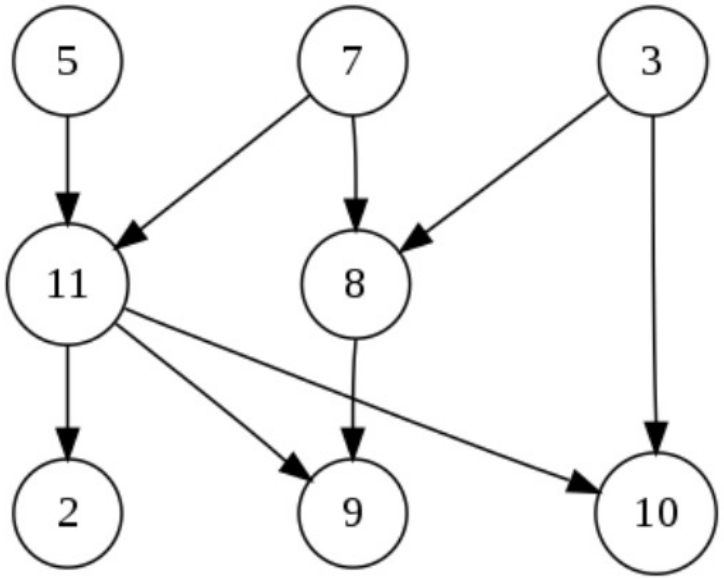
Aplicaciones:

- Orden topológico
- Componentes conectados

Aplicación: Orden topológico

- ***Dado un grafo acíclico dirigido calcular un órden válido de recorrido topológico***

- Definición: el órden topológico de un grafo dirigido es una ordenación lineal en la que cada arista (u,v) establece que u precede v en el orden final.



Soluciones válidas:

- 5, 7, 3, 11, 8, 2, 9, 10
- 3, 5, 7, 8, 11, 2, 9, 10
- 5, 7, 3, 8, 11, 10, 9, 2
- 7, 5, 11, 3, 10, 8, 9, 2
- 5, 7, 11, 2, 3, 8, 9, 10
- 3, 7, 8, 5, 11, 10, 2, 9

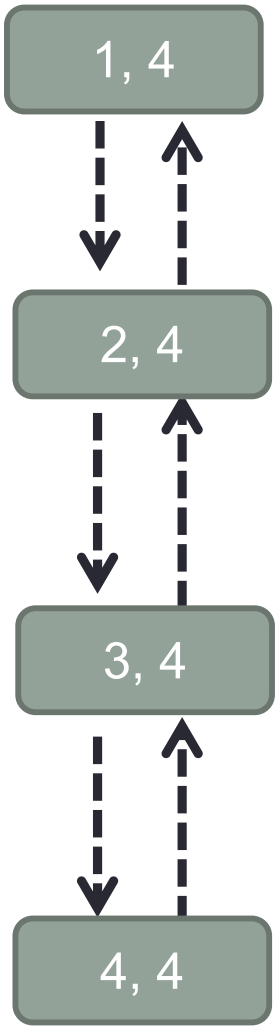
…

https://en.wikipedia.org/wiki/Topological_sorting

Repaso de recursividad

```
def recursiva (n, max):
    if n == max:
        return



    recursiva (n+1, max)


    return
```

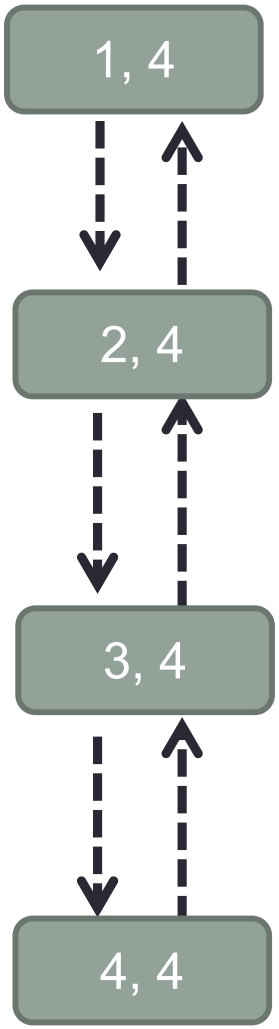
recursiva (1, 4)
return



Repaso de recursividad

```
def recursiva (n, max):
    if n == max:
        return

    numero_guiones = 2 * n

    print('-' * numero_guiones, n)
    recursiva (n+1, max)
    print('-' * numero_guiones, n)
    return
```

recursiva (1, 4)
return



Repaso de recursividad

```
def recursiva (n, max):
    if n == max:
        return

    numero_guiones = 2 * n

    print('-' * numero_guiones, n)
    recursiva (n+1, max)
    print('-' * numero_guiones, n)
    return
```

```
recursiva (1, 4)
return
```

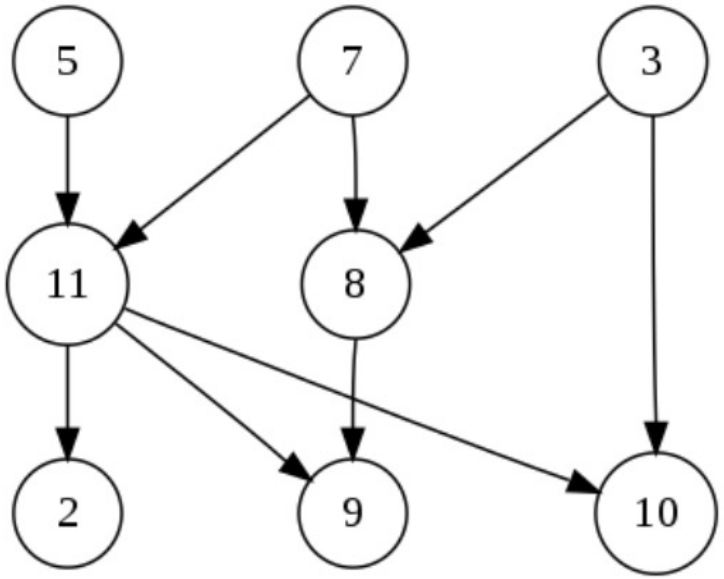
Ejecución del programa

```
-- 1
---- 2
------ 3
------ 3
---- 2
-- 1
```

Aplicación: Orden topológico

- ***Dado un grafo acíclico dirigido calcular un orden válido de recorrido topológico***

- Definición: el órden topológico de un grafo dirigido es una ordenación lineal en la que cada arista (u,v) establece que u precede v en el orden final.



Soluciones válidas:

- 5, 7, 3, 11, 8, 2, 9, 10
- 3, 5, 7, 8, 11, 2, 9, 10
- 5, 7, 3, 8, 11, 10, 9, 2
- 7, 5, 11, 3, 10, 8, 9, 2
- 5, 7, 11, 2, 3, 8, 9, 10
- 3, 7, 8, 5, 11, 10, 2, 9

…

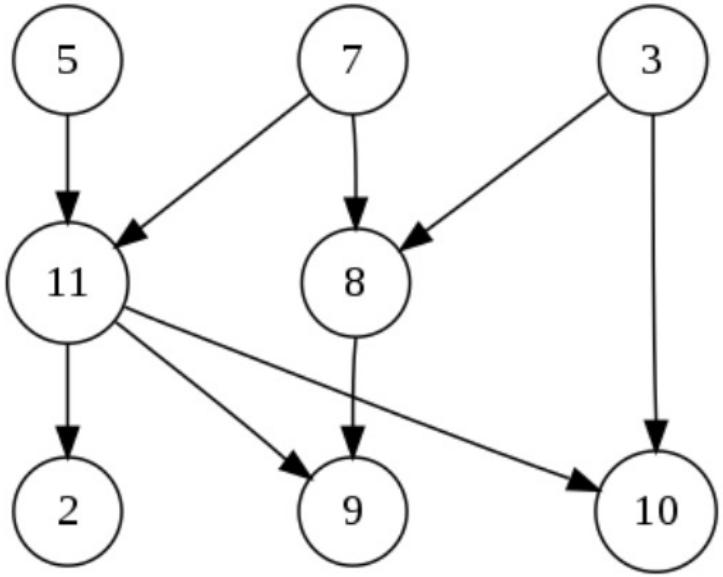
https://en.wikipedia.org/wiki/Topological_sorting

Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)



DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

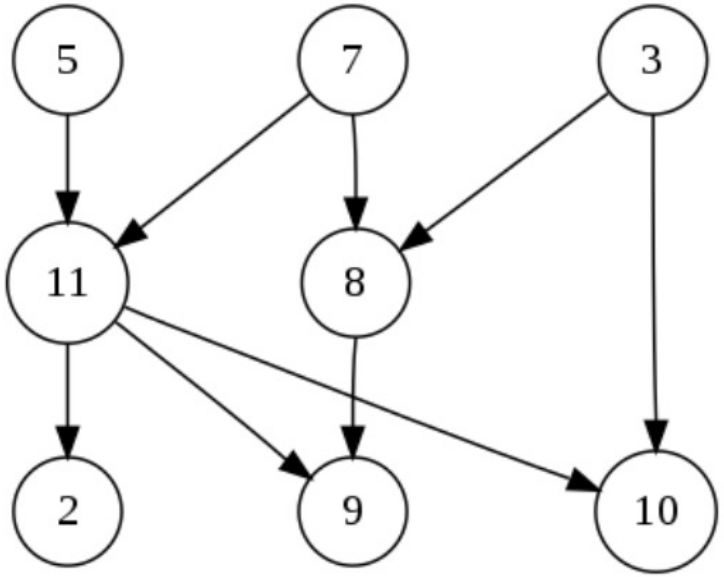
return

Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)



DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=8

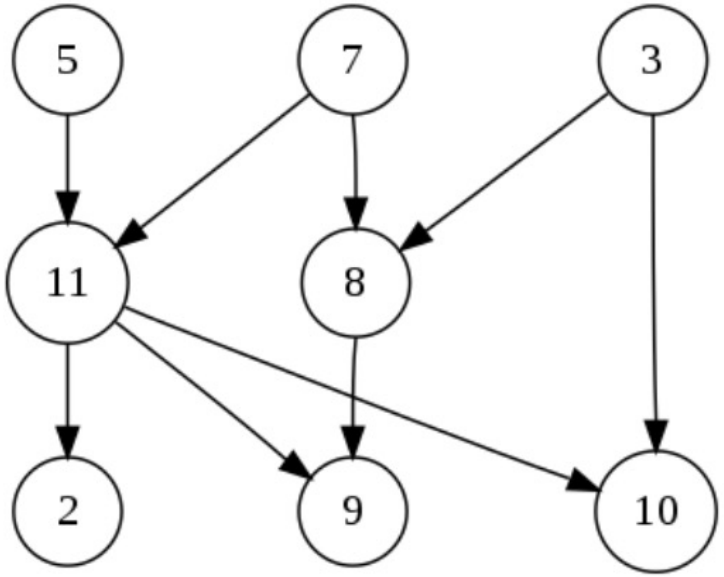
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=8

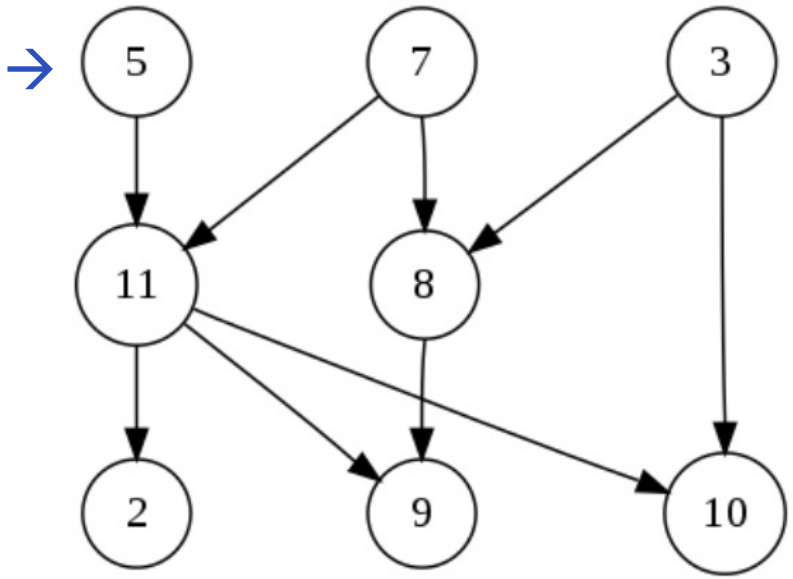
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



DFS (grafo G, vertice v)

- Marcar v como visible
- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)
- **orden(v) = n**
- **n = n – 1**

return

n=8

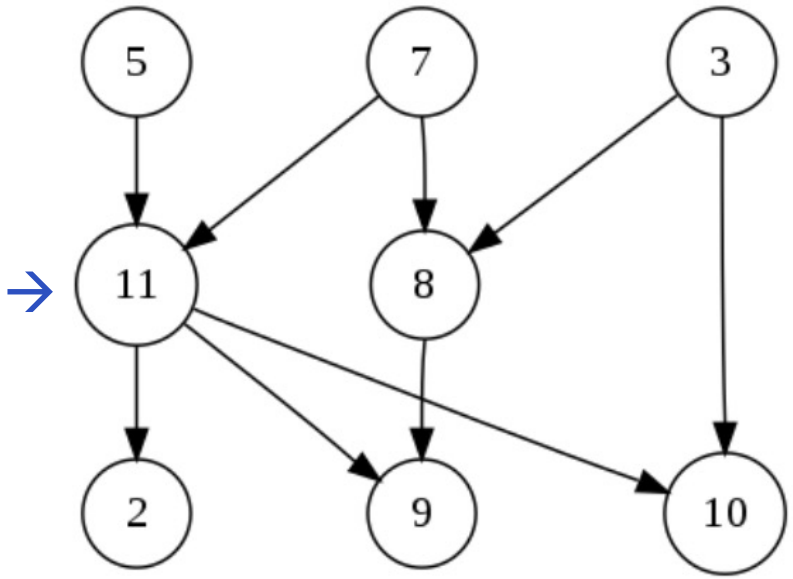
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



DFS (grafo G, vertice v)

- Marcar v como visible

Para todas las aristas (v,w):
- Si w no es visible:
  - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=8

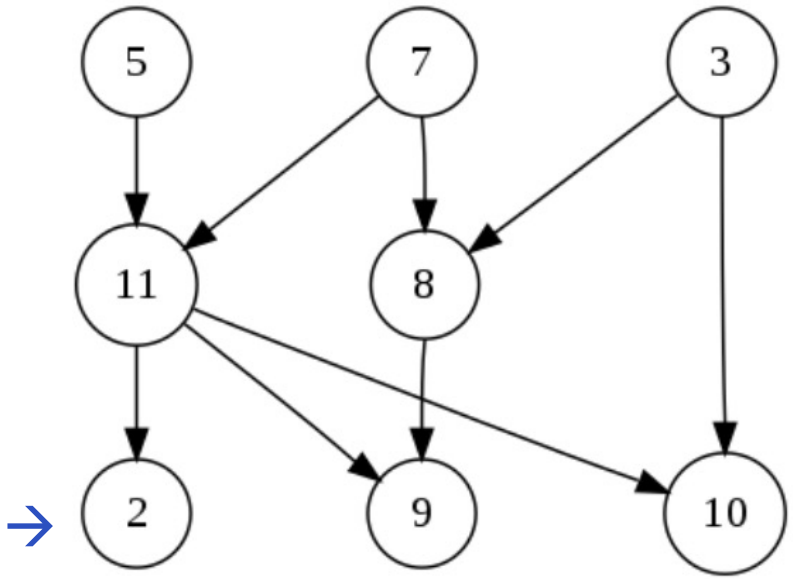
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



DFS (grafo G, vertice v)

- Marcar v como visible
- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)
- **orden(v) = n**
- **n = n – 1**

return

n=8

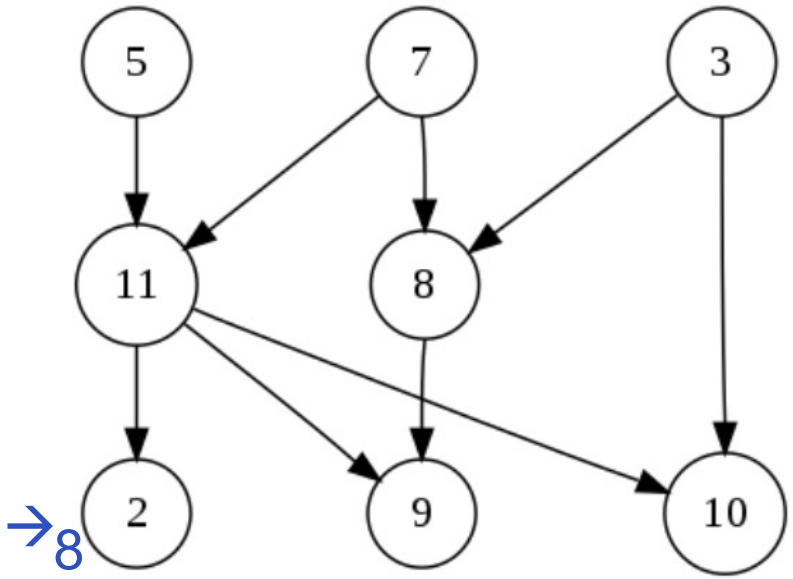
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=7

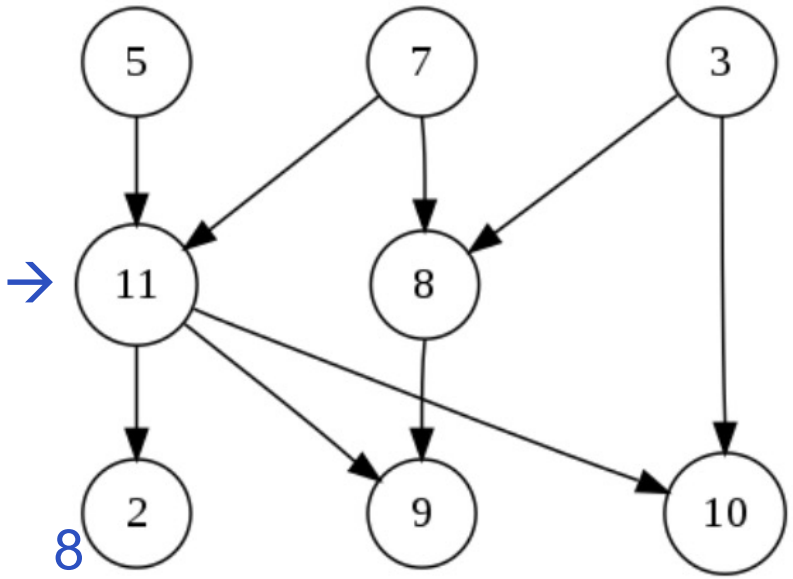
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



8

DFS (grafo G, vertice v)

- Marcar v como visible

Para todas las aristas (v,w):
- Si w no es visible:
  - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=7

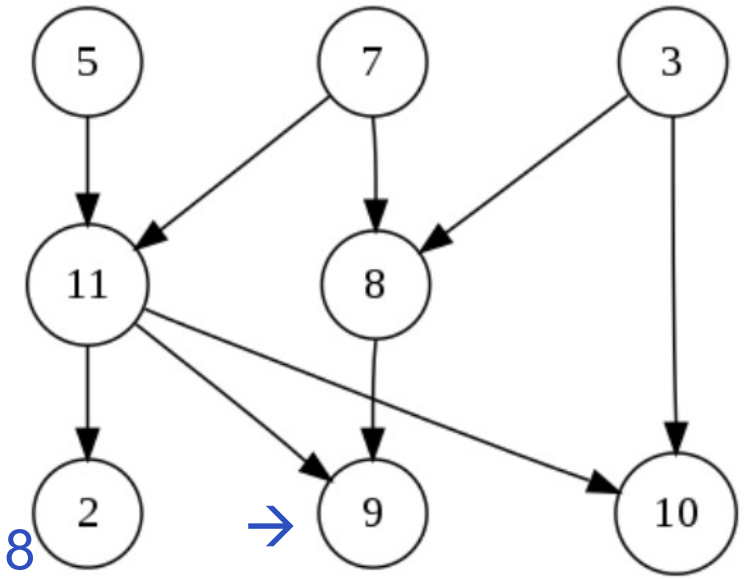
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



8 → n=7

DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

DFS (grafo G, vertice v)

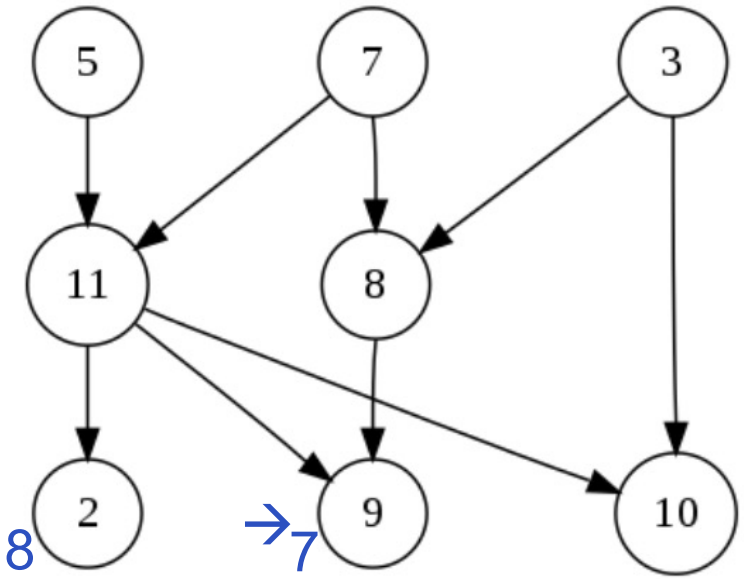
- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

dfs()



n=6

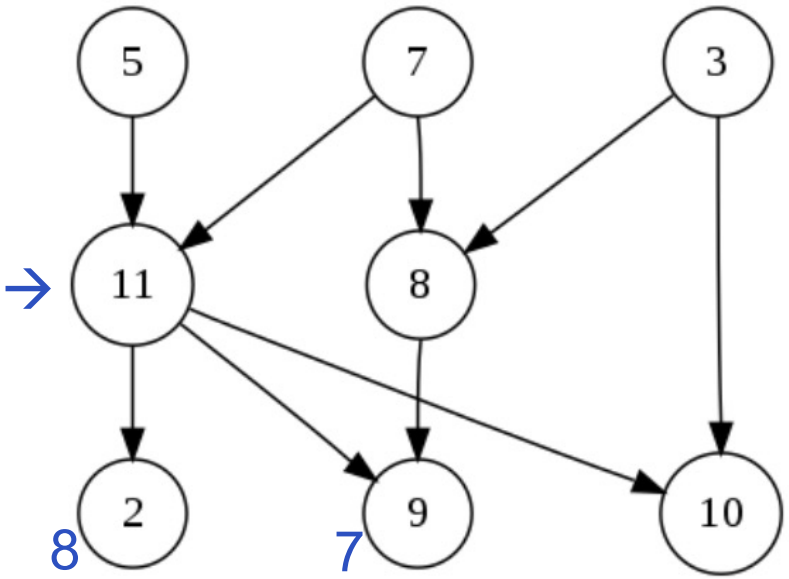
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



DFS (grafo G, vertice v)

- Marcar v como visible

Para todas las aristas (v,w):
- Si w no es visible:
  - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=6

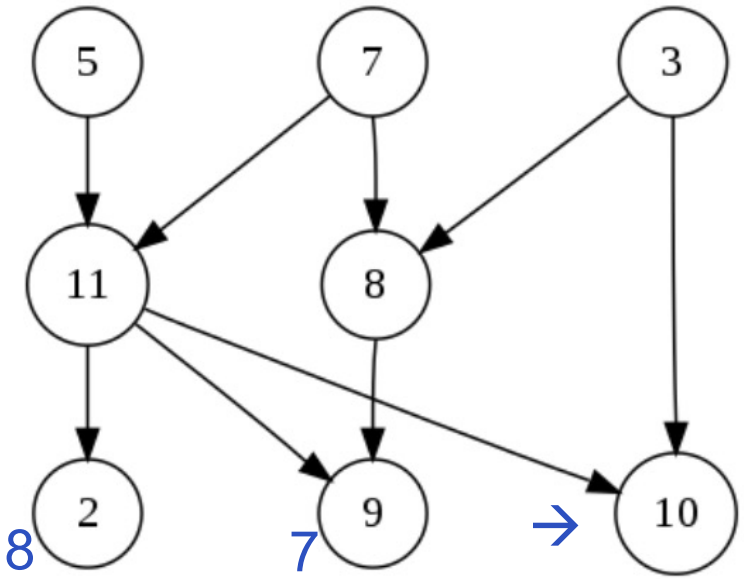
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



8 7 →

DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=6

Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

DFS (grafo G, vertice v)

- Marcar v como visible

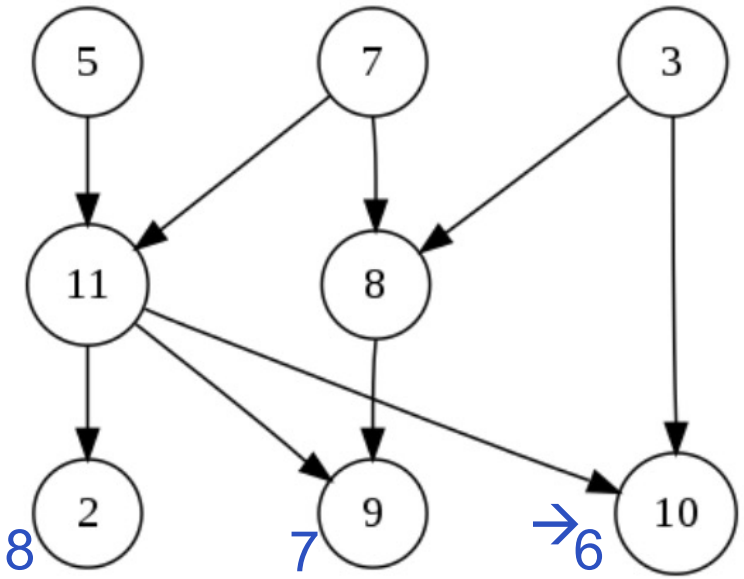
- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=5

dfs()



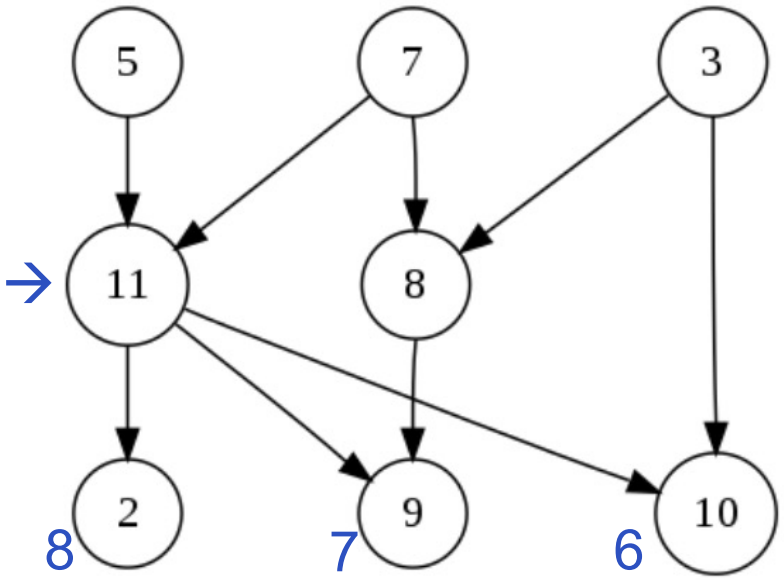
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



DFS (grafo G, vertice v)

- Marcar v como visible
- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)
- **orden(v) = n**
- **n = n – 1**

return

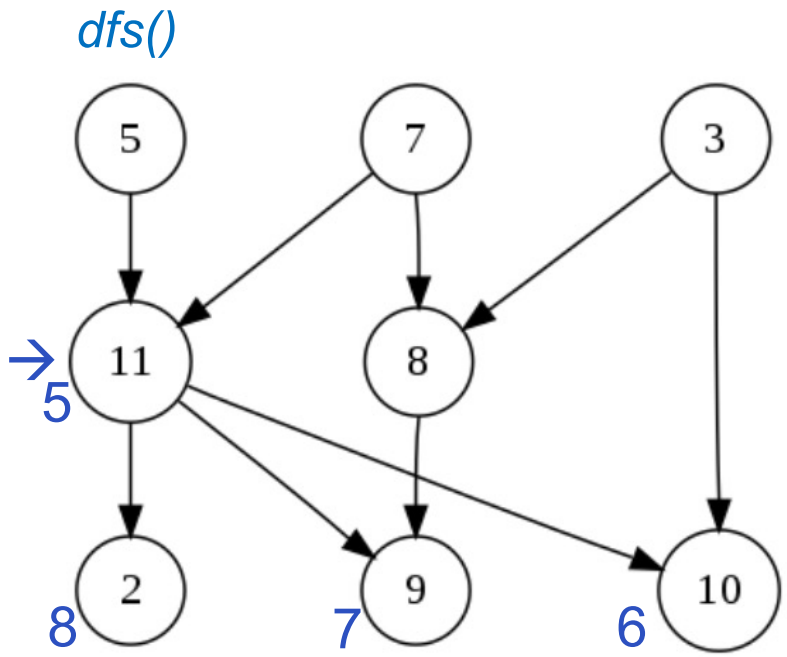
n=5

Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)



DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

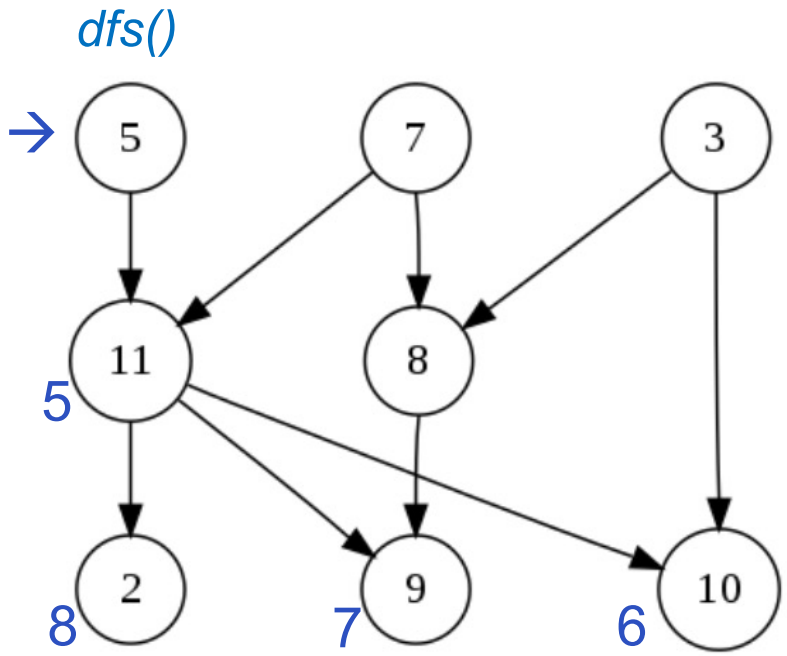
n=4

Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)



DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

n=4

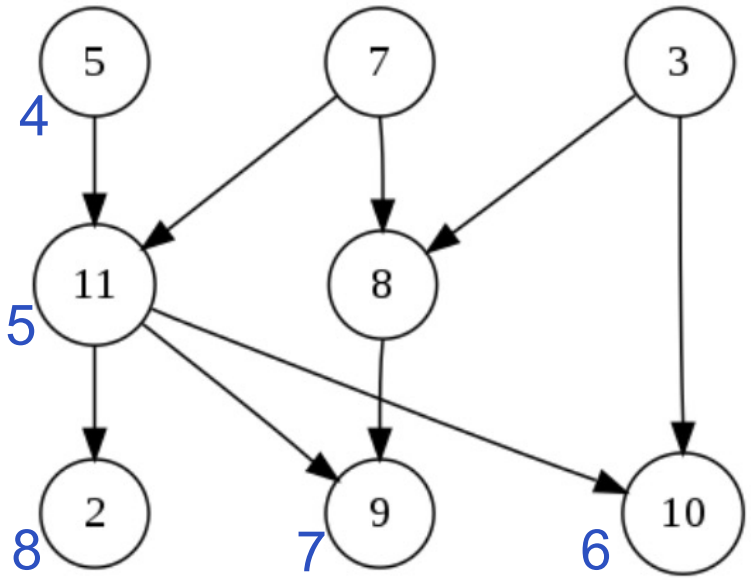
Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)

dfs()



DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

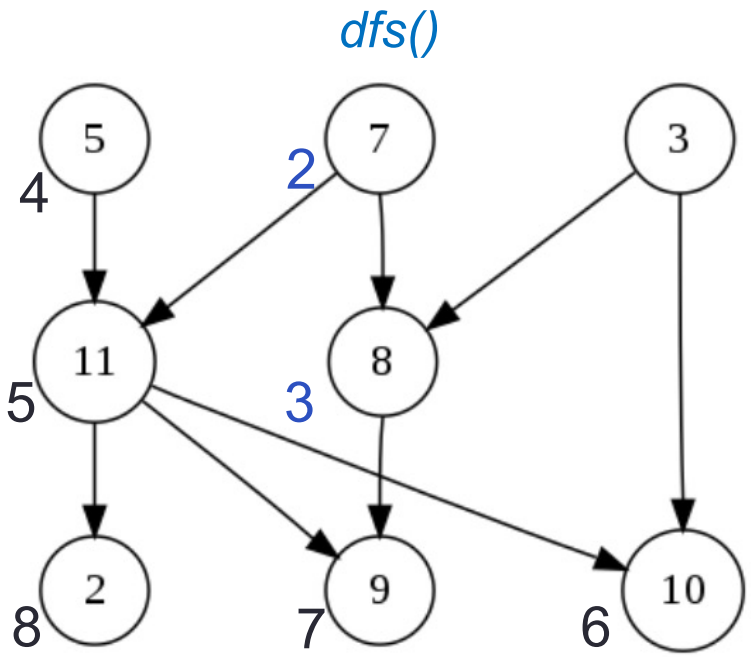
n=3

Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)



DFS (grafo G, vertice v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

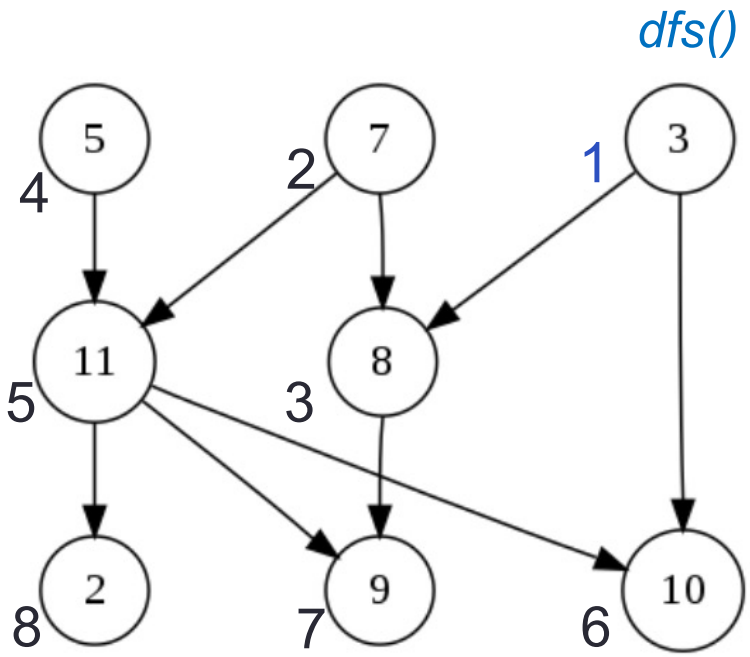
n=1

Aplicación: Orden topológico con DFS

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)



DFS (grafo G, vertice v)

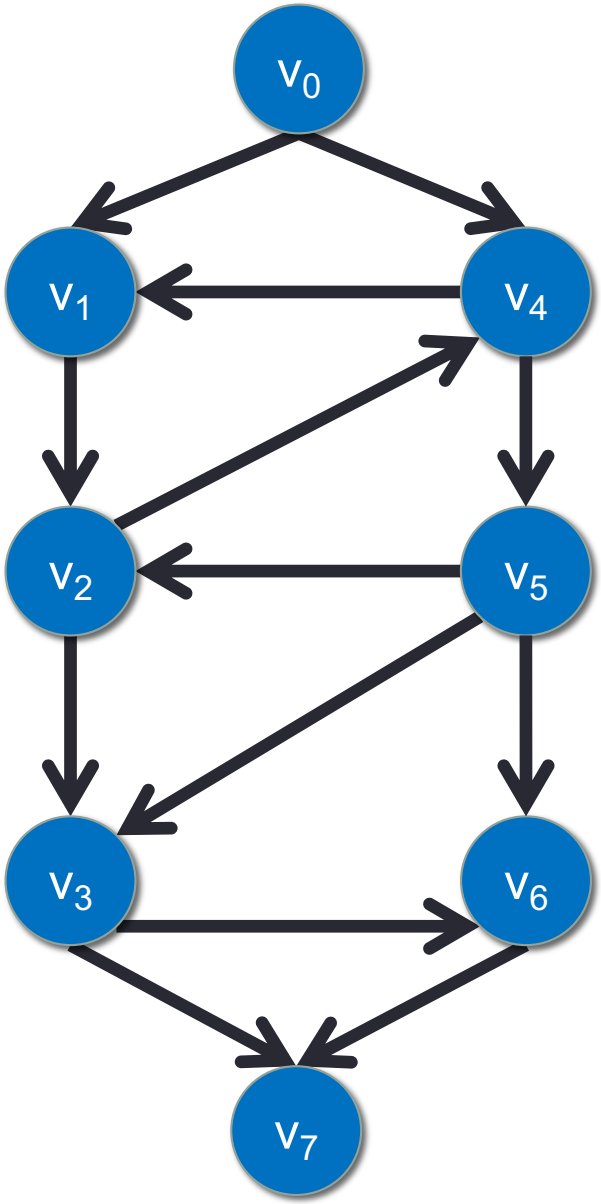
- Marcar v como visible
- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)
- **orden(v) = n**
- **n = n – 1**

return

n=0

Solución: 3 7 8 5 11 10 9 2

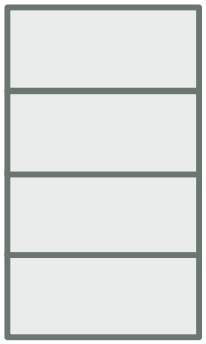
Recorridos en Grafos



- Para recorrer un grafo no siempre es necesario construir el grafo.
- Podemos recorrerlo utilizando sólamente las estructuras de datos auxiliares

Cola

Pila



Vértices pendientes de procesar

Vértices visibles

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

¡ Importante para recorrer grafos grandes !

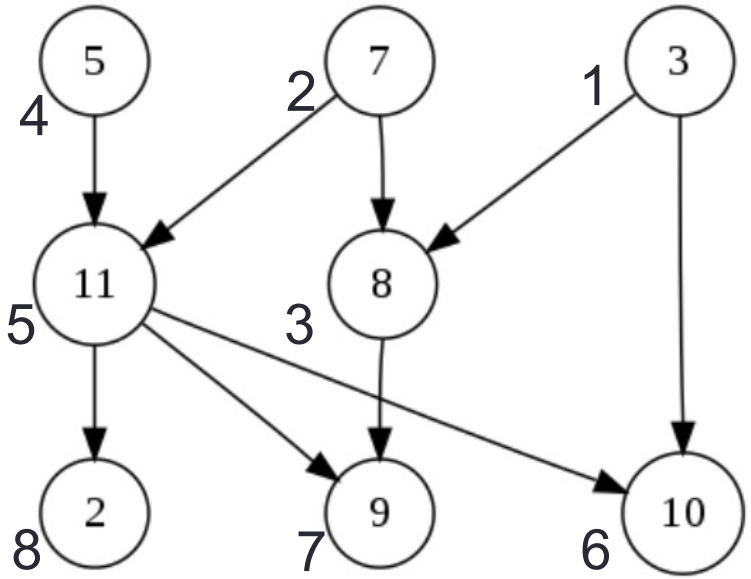
Práctica Semana 3

Utilizando Python y NetworkX, programaremos algoritmos que calculen un orden topológico válido en un grafo dirigido.

n = número de vertices

Con todos los vertices v:

- Si v no es visible:
  - DFS (G, v)



DFS (grafo G, vertice inicial v)

- Marcar v como visible

- Para todas las aristas (v,w):
  - Si w no es visible:
    - DFS (G, w)

- **orden(v) = n**
- **n = n – 1**

return

Solución: 3 7 8 5 11 10 9 2