Algoritmos y Programación

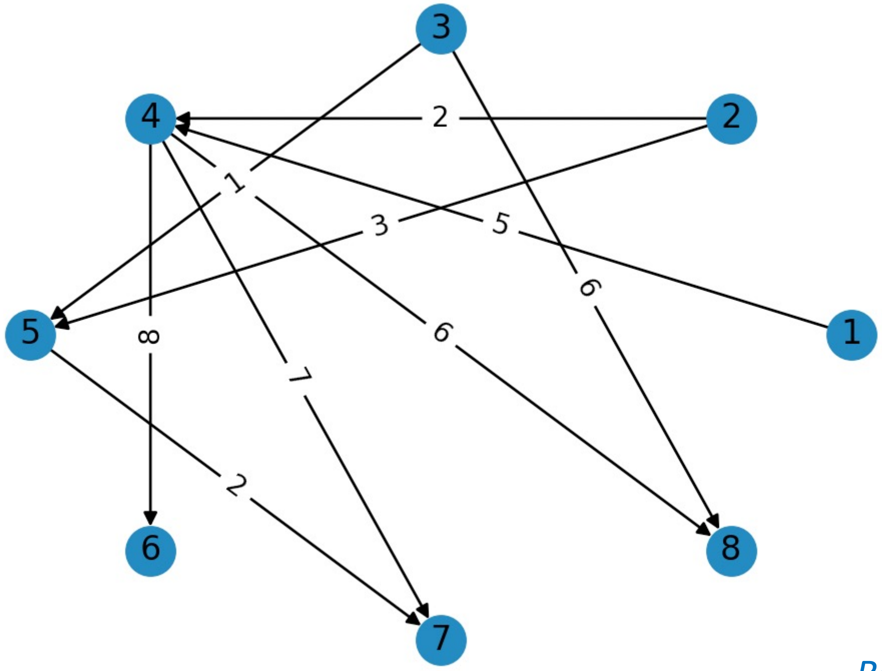
Práctica 3.2: Orden Topológico

Formato del fichero de entrada

- La primera línea es un descriptor: número de vértices, número de aristas
- El resto de las líneas son los vertices con su peso

Ejemplo:

8 9
1 4 5
2 4 2
2 5 3
3 5 1
3 8 6
4 6 8
4 7 7
4 8 6
5 7 2



Reutilizamos la rutina que hicimos en el primer ejercicio de esta semana

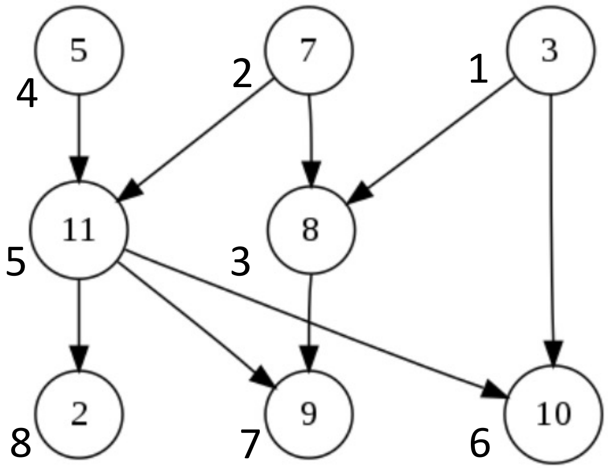
Ejercicio 3.2a

1) Utilizando Python y NetworkX, programa el algoritmo recursivo explicado en clase que calcula 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: cualquier órden topológico válido

VPL 3.2a (1/3)

graph_utils.py

```python
import networkx as nx

def build_digraph_with_weights():

    # Añade aqui la rutina que hiciste en el primer ejercicio
    # de esta semana para crear el grafo dirigido con pesos.
    # ...

    return graph
```

main.py

```python
import networkx  as nx

from graph_utils import *
from solve       import *

graph    = build_digraph_with_weights()
assert nx.is_directed_acyclic_graph(graph)

solution = dfs_topological_sort(graph)
d_swap = {v: k for k, v in solution.items()}

print(dict(sorted(d_swap.items())))
```

VPL 3.2a (2/3)

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

n = número de vertices

Con todos los vertices v:

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

solve.py

```python
import networkx as nx

def dfs_topological_sort(graph):
    """
    Compute one topological sort of the given graph.
    """

    # La solucion que retorna esta función es un diccionario de Python.
    #   * La clave del diccionario es el número del nodo
    #   * El valor es el orden topologico asignado a ese nodo
    #
    # Por ejemplo, si tenemos el siguiente grafo dirigido con 3 vertices:
    #                     3 ---> 2 ---> 1
    # ... el orden topologico es:
    #                El vértice 3 va en la primera posición
    #                El vértice 2 en la segunda posición
    #                El vértice 1 en la tercera posición
    # Con lo que debemos devolver un diccionario con este contenido:
    #     {1: 3, 2: 2, 3: 1}

    N = graph.number_of_nodes()

    visibleNodes = set(); # En este ejercicio utilizamos un set
                          # para recordar los nodos visibles
    order = {}

```

VPL 3.2a (3/3)

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

n = número de vertices

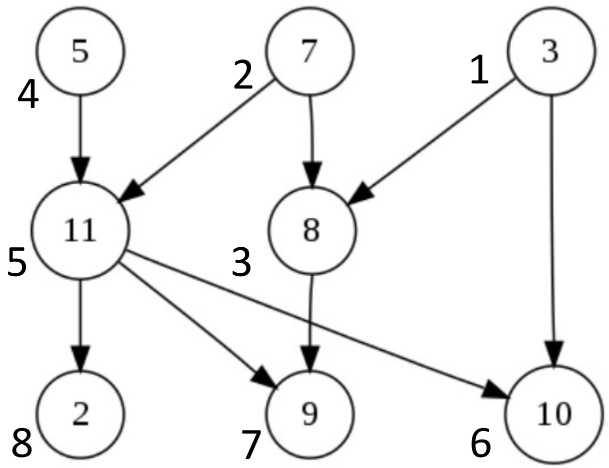
Con todos los vertices v:
- Si v no es visible:
  - DFS (G, v)

solve.py

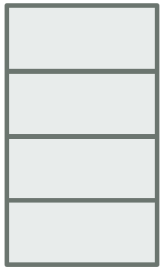
```python
...

# solve it here! ----------------------------------------------

def dfs(u):
    nonlocal N
    # 1. Añade código aqui
    # ...

    return

# 2. Añade código también aqui
# ...

return order
```

Ejercicios 3.2b

2) Utilizando Python y NetworkX, invéntate y programa un algoritmo <u>iterativo</u> que calcule un orden topológico válido en un grafo dirigido.



Pila



Vértices pendientes de procesar

Vértices visibles

Solución: cualquier órden topológico válido

VPL 3.2b (1/4)

graph_utils.py

```python
import networkx as nx

def build_digraph_with_weights():

    # Añade aqui la rutina que hiciste en el primer ejercicio
    # de esta semana para crear el grafo dirigido con pesos.
    # ...

    return graph
```

main.py

```python
import networkx  as nx

from graph_utils import *
from solve       import *

graph    = build_digraph_with_weights()
assert nx.is_directed_acyclic_graph(graph)

solution = dfs_topological_sort(graph)
d_swap = {v: k for k, v in solution.items()}

print(dict(sorted(d_swap.items())))
```


El mismo contenido del primer VPL

VPL 3.2b (2/4)

simple_stack.py

```
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)

```

VPL 3.2b (3/4)

solve.py

```
import networkx as nx

def dfs_topological_sort(graph):
    """
    Compute one topological sort of the given graph.
    """

    # La solucion que retorna esta función es un diccionario de Python.
    #   * La clave del diccionario es el número del nodo
    #   * El valor es el orden topologico asignado a ese nodo
    #
    # Por ejemplo, si tenemos el siguiente grafo dirigido con 3 vertices:
    #                    3 ---> 2 ---> 1
    # ... el orden topologico es:
    #                El vértice 3 va en la primera posición
    #                El vértice 2 en la segunda posición
    #                El vértice 1 en la tercera posición
    # Con lo que debemos devolver un diccionario con este contenido:
    #     {1: 3, 2: 2, 3: 1}

    N = graph.number_of_nodes()

    visibleNodes = set(); # En este ejercicio utilizamos un set
                          # para recordar los nodos visibles
    order = {}

```

El mismo contenido del primer VPL

VPL 3.2b (4/4)

solve.py

```
...
# solve it here! -----------------------------------------------

def dfs_iterative(u):
    nonlocal N
    #  1. Añade código aqui
    #  ...

    return

#  2. Añade código también aqui
#  ...

return order

```

Similar al primer VPL