

Algoritmos y Programación

Creación de un grafo
dirigido con pesos

NetworkX

La práctica de esta semana tiene tres ejercicios.

En el primer ejercicio se debe crear un grafo **dirigido con pesos** a partir de un fichero de entrada. Las principales funciones a utilizar son:

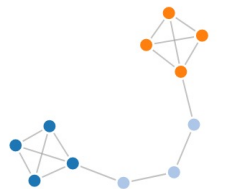
- `nx.DiGraph()`
- `G.add_node()`
- `G.add_edge()`



<https://networkx.org/documentation/stable/tutorial.html>

<https://networkx.org/documentation/stable/reference/index.html>

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

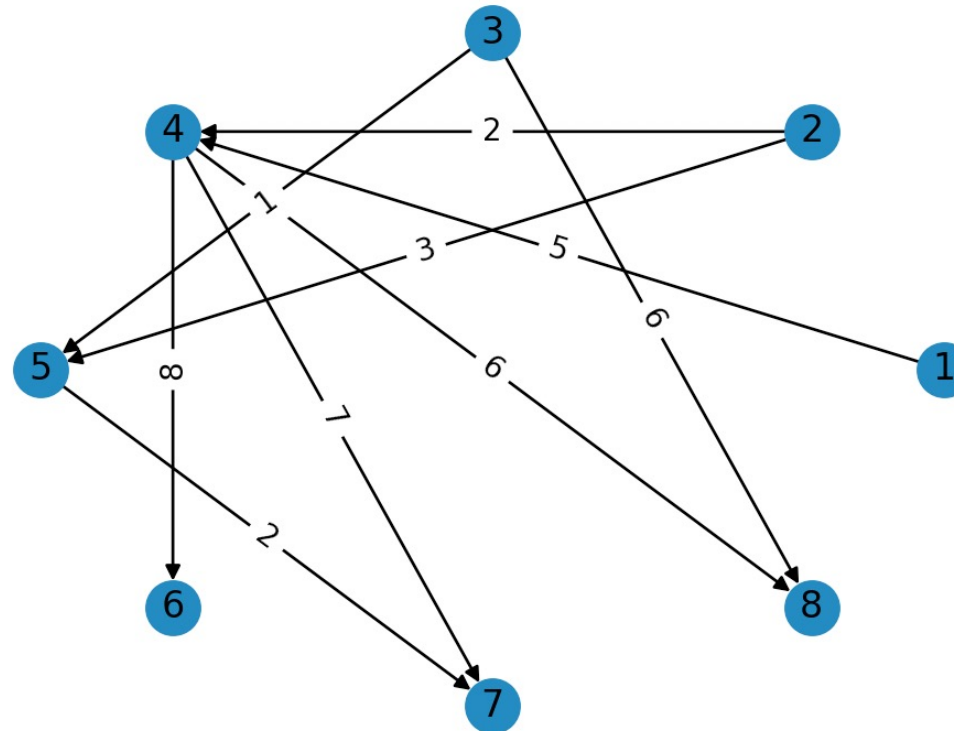


Formato del fichero de entrada

- La primera línea es un descriptor: número de vértices, número de aristas
- El resto de las líneas son las aristas 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
```



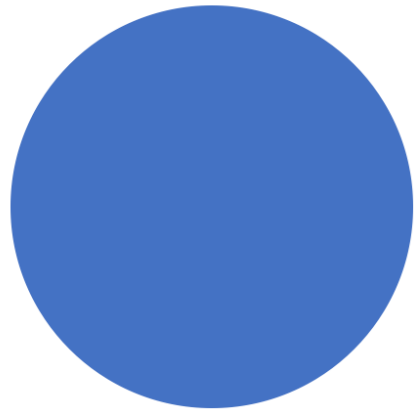
VPL: Ejercicio 4a

main.py

```
1 import networkx as nx
2 from solve import *
3
4 graph = build_digraph_with_weights()
5
6 print("Number of nodes: " + str(graph.number_of_nodes()))
7 print("Nodes: ", graph.nodes())
8 print("Number of edges: " + str(graph.number_of_edges()))
9 print("Edges: ", graph.edges(data=True))
10
11 # Paso 3 (Opcional): Utilizando PyCharm añade aquí el código
12 # necesario para mostrar gráficamente el grafo.
13 # ...
14
```

solve.py

```
1 import networkx as nx
2
3 def build_digraph_with_weights():
4     """
5     Read data from the standard input and build the corresponding
6     directed graph with weights. Nodes numbering starts with number
7     1 (that is, nodes are 1,2,3,...)
8     """
9
10    first_line = input().split()
11    num_nodes = int(first_line[0])
12    num_edges = int(first_line[1])
13
14    # Paso 1: Crear grafo direccional con num_nodes
15
16
17    # Paso 2: Añadir los vértices del grafo
18
19
20    return graph
```



Algoritmos y Programación

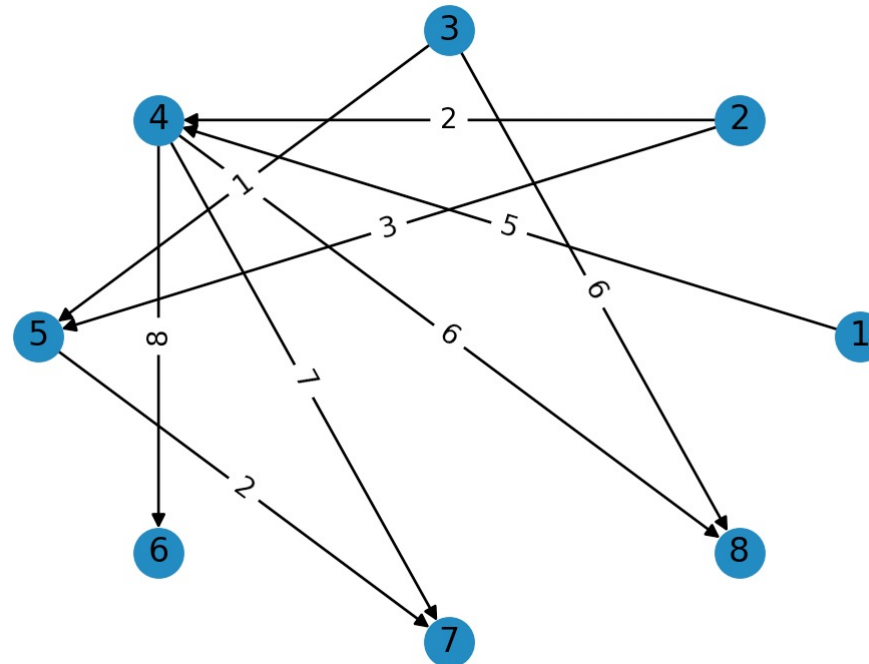
Orden Topológico

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 ejercicio anterior*

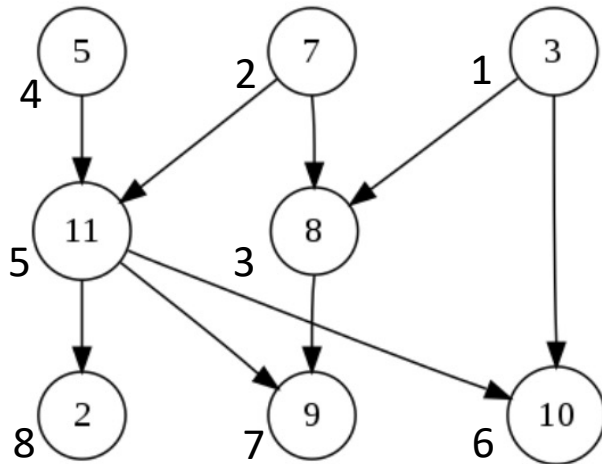
Ejercicio 4b

- 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 orden topológico válido

VPL 4b (1/3)

graph_utils.py

```
1 import networkx as nx
2
3 def build_digraph_with_weights():
4
5     # Añade aquí la rutina que hiciste en el primer ejercicio
6     # de esta semana para crear el grafo dirigido con pesos.
7     # ...
8
9     return graph
```

main.py

```
1 import networkx as nx
2
3 from graph_utils import *
4 from solve import *
5
6 graph = build_digraph_with_weights()
7 assert nx.is_directed_acyclic_graph(graph)
8
9 solution = dfs_topological_sort(graph)
10 d_swap = {v: k for k, v in solution.items()}
11
12 print(dict(sorted(d_swap.items())))
```


VPL 4b

(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)
 - $\text{orden}(v) = n$
 - $n = n - 1$
- return

n = número de vertices

Con todos los vertices v:

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

solve.py

```
1 import networkx as nx
2
3 def dfs_topological_sort(graph):
4     """
5     Compute one topological sort of the given graph.
6     """
7
8     # La solución que retorna esta función es un diccionario de Python.
9     # * La clave del diccionario es el número del nodo
10    # * El valor es el orden topológico asignado a ese nodo
11    #
12    # Por ejemplo, si tenemos el siguiente grafo dirigido con 3 vertices:
13    #           3 ---> 2 ---> 1
14    # ... el orden topológico es:
15    #           El vértice 3 va en la primera posición
16    #           El vértice 2 en la segunda posición
17    #           El vértice 1 en la tercera posición
18    # Con lo que debemos devolver un diccionario con este contenido:
19    #           {1: 3, 2: 2, 3: 1}
20
21    N = graph.number_of_nodes()
22
23    visibleNodes = set(); # En este ejercicio utilizamos un set
24                        # para recordar los nodos visibles
25    order = {}
26
```

VPL 4b

(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**

solve.py

...

27

28

29 ▾

30

31

32

33

34

35

36

37

38

39

40

solve it here! -----

def dfs(u):

nonlocal N

1. Añade código aquí

...

return

2. Añade código también aquí

...

return order

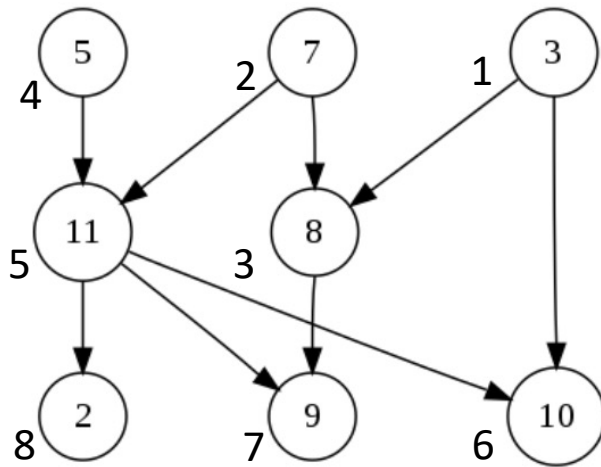
n = número de vertices

Con todos los vertices v:

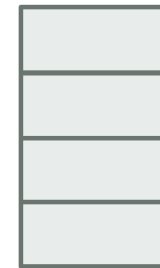
- Si v no es visible:
 - DFS (G, v)

Ejercicio 4c

- 2) Utilizando Python y NetworkX, diseña un algoritmo iterativo que calcule un orden topológico válido en un grafo dirigido.



Pila



Vértices pendientes de procesar

Vértices visibles



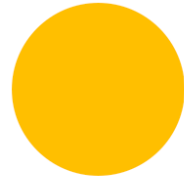
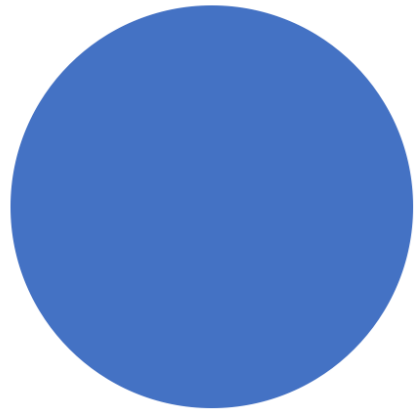
Solución: cualquier orden topológico válido



VPL 4c

simple_stack.py

```
1 class Stack:
2     def __init__(self):
3         self.items = []
4
5     def isEmpty(self):
6         return self.items == []
7
8     def push(self, item):
9         self.items.append(item)
10
11    def pop(self):
12        return self.items.pop()
13
14    def peek(self):
15        return self.items[len(self.items)-1]
16
17    def size(self):
18        return len(self.items)
19
```



Algoritmos y Programación

Problema del submarino

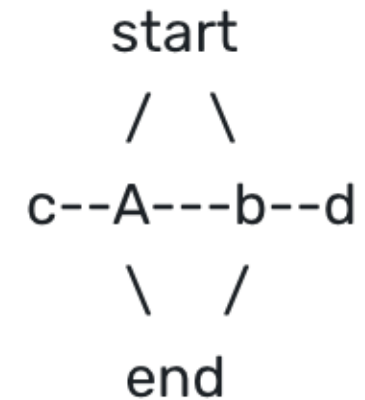
Problema del Submarino

Estamos atrapados en un submarino averiado en una red de cuevas submarinas, y tu objetivo es **calcular todas las rutas posibles para escapar**.

Debes moverte desde una cueva de inicio ("start") hasta la cueva de salida ("end").

Las cuevas están conectadas por tuneles, y hay dos tipos de cuevas:

- Cuevas grandes, representadas por letras MAYÚSCULAS.
Estas cuevas son lo suficientemente amplias como para visitarlas cuantas veces desees durante una ruta.
- Cuevas pequeñas, representadas por letras minúsculas.
Estas cuevas son más estrechas, por lo que sólo se pueden visitar una vez en cada ruta.



```
['start', 'A', 'end']
['start', 'b', 'end']
['start', 'A', 'b', 'end']
['start', 'b', 'A', 'end']
['start', 'A', 'b', 'A', 'end']
['start', 'A', 'c', 'A', 'end']
['start', 'A', 'c', 'A', 'b', 'end']
['start', 'b', 'A', 'c', 'A', 'end']
['start', 'A', 'b', 'A', 'c', 'A', 'end']
['start', 'A', 'c', 'A', 'b', 'A', 'end']
```

Formato del fichero de entrada

- La primera línea indica el número de tuneles
- El resto de las líneas son los tuneles

Ejemplo:

7
start-A
start-b
A-c
A-b
b-d
A-end
b-end

start
/ \
c--A---b--d
\ /
end

Salida

['start', 'A', 'end']
['start', 'b', 'end']
['start', 'A', 'b', 'end']
['start', 'b', 'A', 'end']
['start', 'A', 'b', 'A', 'end']
['start', 'A', 'c', 'A', 'end']
['start', 'A', 'c', 'A', 'b', 'end']
['start', 'b', 'A', 'c', 'A', 'end']
['start', 'A', 'b', 'A', 'c', 'A', 'end']
['start', 'A', 'c', 'A', 'b', 'A', 'end']
10