P3 – Grafos. Programación Dinámica

Juan José Sánchez Peña

Requisitos de entrega (1/2)

- Crear una carpeta con el nombre p3**NN**(1) e incluir los siguientes ficheros:
  - p3**NN**.py: incluirá el código de las funciones implementadas en la práctica y los `imports` estrictamente necesarios.
  - p3**NN**_optional.py: contendrá el código implementado para la parte opcional III.
  - p3**NN**.pdf: memoria que contenga las respuestas a las cuestiones de la práctica.
  - Los ficheros .py auxiliares que hayáis creado para el correcto funcionamiento de toda la práctica.

(1) **NN** indica el número de pareja.

Requisitos de entrega (2/2)

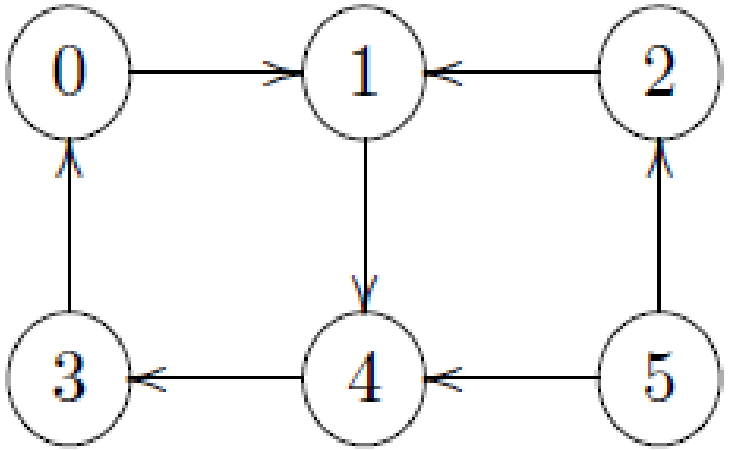
- Observaciones
  - Los nombres y parámetros de las funciones definidas en p3**NN**.py deben ajustarse EXACTAMENTE a lo definido en el enunciado.
  - En la memoria se identificará claramente el nombre de los estudiantes y el número de pareja. Si se añaden figuras o gráficos, deben realizarse sobre fondo blanco.
  - Para la entrega, comprimir la carpeta en un fichero llamado p3**NN**.zip. No añadir ninguna estructura de subdirectorios a dicha carpeta.
  - **La práctica no se corregirá hasta que el envío no siga esta estructura**.

Corrección

- Ejecución del script que importará p3**NN**.py, que comprobará la corrección de dicho código. **¡¡IMPORTANTE!! La práctica no se corregirá mientras este script no se ejecute correctamente, penalizando las segundas entregas**.
- Revisión de ciertas funciones implementadas en p3**NN**.py.
- Revisión de la memoria con las respuestas a las cuestiones planteadas.

TAD: grafo dirigido no ponderado (1/6)

- Grafo dirigido: conjunto de vértices (V) y ramas (E) que conectan dichos vértices en un sentido.
- Grafo dirigido no ponderado: las ramas no tienen peso.



TAD: grafo dirigido no ponderado (2/6)

```
def __init__(self):
    self._V = dict()                # Dictionary with G nodes: Dict[str, Dict[str]]
    self._E = dict()                # Dictionary with G edges: Dict[str, Set]
```

- `self._V`: es un diccionario donde se almacenan los nodos. La clave es un nodo y el valor es otro diccionario con los atributos de los nodos que se utilizarán en los diferentes algoritmos que implementaremos en la práctica.
- `self._E`: es un diccionario que implementa las listas de adyacencia de los nodos. La clave es un nodo y el valor es un conjunto de nodos a los que está conectado.

TAD: grafo dirigido no ponderado (3/6)

```
def add_node(self, vertex) -> None:
```

- Inicializar _V: llamar a la función `_init_node(self, vertex)`.
- Inicializar _E: inicializar la lista de adyacencia.

```
def add_edge(self, vertex_from, vertex_to) -> None:
```

- Añade un arco entre `vertex_from` a `vertex_to`. Los nodos se añaden si no existen en el grafo.
- Añadir nodo `vertex_from`.
- Añadir nodo `vertex_to`.
- Añadir a la lista de adyacencia.

TAD: grafo dirigido no ponderado (4/6)

```
def nodes(self) -> KeysView[str]:
```

- `KeysView`: tipo de vista que proporciona acceso a las claves de un diccionario de manera dinámica.

```
def adj(self, vertex) -> Set[str]:
```

- Devuelve los nodos adyacentes al nodo `vertex`.

```
def exists_edge(self, vertex_from, vertex_to)-> bool:
```

- Devuelve `True`/`False` si el nodo `vertex_to` se encuentra en la lista de adyacencia de `vertex_from`.

TAD: grafo dirigido no ponderado (5/6)

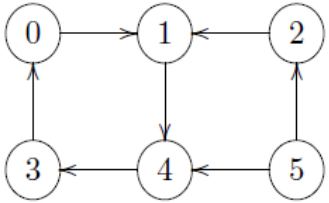
```
def __str__(self) -> str:
```

- Imprime por pantalla las diferentes salidas.

```
Vertices:
0: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
1: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
2: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
4: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
3: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
5: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}

Aristas:
0: {1}
1: {4}
2: {1}
4: {3}
3: {0}
5: {2, 4}
```

TAD: grafo dirigido no ponderado (6/6)



- Ejemplo:

```
print('Nodes:')
print(G.nodes())
```

```
Nodes:
dict_keys([0, 1, 2, 4, 3, 5])
```

```
print('Adyacentes nodo 5')
print(G.adj(5))
```

```
Adyacentes nodo 5
{2, 4}
```

```
print('¿Es 2 adyacente de 5?')
print(G.exists_edge(5,2))
print('¿Es 3 adyacente de 5?')
print(G.exists_edge(5,3))
```

```
¿Es 2 adyacente de 5?
True
¿Es 3 adyacente de 5?
False
```