Algoritmos y estructura de datos avanzadas. Año Académico 2024-25

Práctica 3

Grafos, Programación dinámica

Fecha de entrega: Grupo 126: **9 de diciembre 2024 a las 23:59**
Grupo 127: **10 de diciembre 2024 a las 23:59**

¡Atención! Muchas de las funciones de esta práctica (y de las demás) se pasarán por un script de corrección automática. Esto quiere decir que tenéis que seguir escupulosamente las indicaciones en lo que se refiere a nombres de ficheros, nombres de funciones, parámetros, valores de retorno, ficheros, etc. Las funciones no deben escribir nada en pantalla (si ponéis algunos `print` para hacer *debuging* que no se os olvide de eliminarlos antes de entregar). Los ficheros que se entregan deben contener sólo las funciones, ningún *script* de prueba ni nada que se ejecute cuando se hace el `import` del fichero.

No seguir estas indicaciones puede resultar en una penalización o, en los casos peores, en la no corrección de la práctica.

I. Grafos: Búsqueda en profundidad y componentes fuertemente conexas

I.A Implementación del TAD Grafo

En esta primera parte implementaremos el Tipo Abstracto de Datos `Grafo` para representar un grafo dirigido no ponderado. Para ello utilizaremos la clase `Graph` definida en el fichero `graph_24.py` cuyas cabeceras se muestran a continuación:

```python
from typing import Set, List, Generator, Tuple, KeysView, Iterable
import os

class Graph:

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


    def add_node(self, vertex) -> None:
        ''' Add a single node if it is not in the graph
        '''
        pass

    def add_edge(self, vertex_from, vertex_to) -> None:
        ''' Add an edge from vertex_from to vertex_to. The nodes will be
            added if they are not already in the graph.
        '''
        pass

    def nodes(self) -> KeysView[str]:
        '''Devuelve las keys de los nodos del grafo'''
        pass

    def adj(self, vertex) -> Set[str]:
        '''Devuelve los nodos adjacentes a vertex '''
```

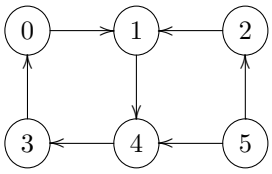
```
        pass

    def exists_edge(self, vertex_from, vertex_to)-> bool:
        ''' Devuelve True/False si vertex_to se encuentra en la
            lista de adyacencia de vertex_from
        '''
        pass

    def _init_node(self, vertex) -> None:
        ''' Set vertex initial values'''

        self._V[vertex] = {'color': 'WHITE', 'parent': None,
                           'd_time': None, 'f_time': None}

    def __str__(self) -> str:
        pass


### Auxiliary functions to manage graphs ########

def read_adjlist(file: str) -> Graph:
    ''' Read graph in adjacency list format from file.
    '''

    G = Graph()
    with open(file,'r') as f:
        for line in f:
            l = line.split()
            if l:
                u = l[0]
                G.add_node(u)
                for v in l[1:]:
                    G.add_edge(u, v)
    return G


def write_adjlist(G: Graph, file: str) -> None:
    '''Write graph G in single-line adjacency-list format to file.
    '''

    file_path = os.path.join(os.path.dirname(__file__), file)
    with open(file_path,'r') as f:
        for u in G.nodes():
            f.write(f'{u}')
            f.writelines([f' {v}' for v in G.adj(u)])
            f.write('\n')


### Driver code

if __name__ == '__main__':
    #G = read_adjlist('./graph.txt')
    #print(G)
```

Como puedes observar en el método `__init__` se utilizan dos estructuras privadas para almacenar el grafo:

- `self._V` Almacena los nodos. Es un diccionario donde cada clave es un nodo y cuyo valor es otro diccionario con los atributos de los nodos que utilizan los diferentes algoritmos que se implementarán en la práctica. Por ejemplo, en el algoritmo de búsqueda en profundidad (dfs) los nodos tienen los atributos: tiempo de descubrimiento (`d_time`), tiempo de finalización (`f_time`), `parent` y `color` (ver el método privado `_init_node(self, vertex)`).
- `self._E` Implementa las listas de adyacencia de los nodos. Es un diccionario donde cada clave es un nodo y cuyos valores son el conjunto de nodos a los que está conectado, `set()`. Por tanto, no se permiten multigrafos.

Debes programar los métodos de la clase `Graph`

Ejemplo: Dado el grafo:



El código:

```
import graph_24 as g

G = g.Graph()

G.add_edge(0, 1)
G.add_edge(2, 1)
G.add_edge(1, 4)
G.add_edge(4, 3)
G.add_edge(5, 4)
G.add_edge(3, 0)
G.add_edge(5, 2)

print(G)
```

producirá la salida

```
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}
```

I.B Recorrido en grafos y componentes fuertemente conexas

1. Programar el método `dfs()` que implementa el algoritmo de búsqueda en profundidad.

```
from collections import deque

def dfs(self, nodes_sorted: Iterable[str] = None) -> List[List[Tuple]]:
    ''' Depth find search driver

        nodes_sorted: Si se le pasa un iterable el bucle principal de DFS
        se iterará según el orden del iterable (eg en Tarjan)

        Devuelve un bosque dfs en la que cada una de las sublistas es un
        árnbol dfs. Cada elemnto del arbol es una tupla (vertex, parent)
    '''
    pass
```

2. Programar la función `graph_conjugated()` que devuelve el grafo conjugado de `G` y el método `tarjan()` de la clase `Graph` que implementa el algoritmo de Tarjan para obtener las componentes fuertemente conexas de un grafo.

```python
def graph_conjugate(G: Graph) -> Graph:
    ''' Devuelve el grafo conjugado de G '''
    pass

def tarjan(self)-> List[List[str]]:
    ''' Return a list with the strong connected components '''
    pass
```

Ejemplo: Cuando `G` es el grafo del apartado anterior el código,

```python
print(f' DFS forest : {G . dfs ()} ')
print()
print(G)
print()
print(f' scc : {G . tarjan ()} ')
```

produciría la salida (cuidado, recuerda que, la salida depende del orden en que se procesen los nodos)

```
DFS forest : [[(3, 4), (4, 1), (1, 0), (0, None)], [(2, None)], [(5, None)]]

Vertices:
0: {'color': 'BLACK', 'parent': None, 'd_time': 1, 'f_time': 8}
1: {'color': 'BLACK', 'parent': 0, 'd_time': 2, 'f_time': 7}
2: {'color': 'BLACK', 'parent': None, 'd_time': 9, 'f_time': 10}
4: {'color': 'BLACK', 'parent': 1, 'd_time': 3, 'f_time': 6}
3: {'color': 'BLACK', 'parent': 4, 'd_time': 4, 'f_time': 5}
5: {'color': 'BLACK', 'parent': None, 'd_time': 11, 'f_time': 12}

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

scc : [[0, 3, 4, 1], [2], [5]]
```

I.C Componente Gigante y umbral de percolación

Un grafo de Erdös-Renyi es un grafo aleatorio con n nodos en el que cada nodo tiene **en media** m vecinos (n y m son los parámetros que definen el grafo). Por tanto, en un grafo Erdös-Renyi la probabilidad de que dos nodos cualesquiera esten conectados es $p = m/n$.

Se ha observado que cuando m es inferor al valor crítico $m_c = 1$ el grafo está muy desconectado: la inmensa mayoría de las componentes conexas del grafo estarán formadas por un único nodo. Sin embargo, si el número promedio de vecinos por nodo supera un valor crítico m_c (llamado umbral de percolación), el grafo *condensa* y aparece *una única componente conexa gigante*. La inmensa mayoría de los nodos pertenecerán a esa única componente conexa gigante mientras que el resto de componentes conexas serán de tamaño muy reducido. En los grafos **no dirigidos** la transición de fase que se observa en el umbral de percolación $m_c = 1$ es muy abrupta.

¿Ocurrirá igual en **grafos dirigidos**?. Para analizar el comportamiento vamos a crear grafos aleatorios todos del mismo tamaño n pero con diferentes valores de m. Se analizará el tamaño de la mayor de las componentes fuertemente conexas (normalizado por n) frente al valor de m.

Discute con rigor los resultados obtenidos.

Nota: Para que tus conclusiones sean representativas debes utilizar grafos de al menos de 10^3 nodos y representar un número apreciable de puntos en tus gráficas. Notad que estamos particularmente interesados en analizar la transición del grafo entre el estado completamente desconectado al estado con un única componente fuertemente conexa, es decir en el entorno del valor umbral de percolación (si este existiese).

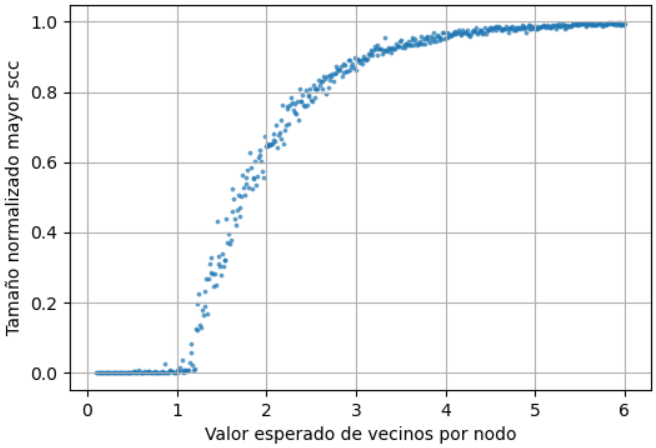
Se deberán crear, **al menos** las funciones siguientes y el programa que genera los puntos de la gráfica.

1. ```
   import python_24 as g

   def erdos_renyi(n: int, m: float = 1.) -> g.Graph:
       ''' Devuelve un grafo aleatorio dirigido
           n: numero de nodos del grafo
           m: numero medio de vecinos de un nodo

           El número de vecinos de cada nodo del grafo se obtiene
           a partir de una muestra de una distribución binomial de
           probabilidad p = m/n.
       '''
   ```

2. ```
   def size_max_scc(n: int, m: float) -> Tuple[float, float]:
       ''' Genera un grafo dirigo aleatorio de parametros n y m.
           Calcula el tamaño de la mayor de las componentes fuertemente conexas.
           Devuelve una tupla con el tamaño de la mayor scc del grafo normalizada por n
           y el valor de m
       '''
   ```

La siguiente función se ha utilizado para representar los datos

```
import matplotlib.pyplot as plt

def grafica(points, file='percolation.png') -> None:
    ''' Genera una grafica en el fichero file '''

    y, x = zip(*points)          # Desempaqueta los points

    fig, ax = plt.subplots(1, 1, figsize=(6, 4))
    ax.scatter(x, y,  alpha=0.6, s=3)
    ax.set_ylabel (f'Tamano normalizado mayor scc')
    ax.set_xlabel (f'Valor esperado de vecinos por nodo')
    ax.grid()
    plt.savefig(file)
    #plt.show()                  # descomenta para grafico interactivo
```



II. PROGRAMACIÓN DINÁMICA

II.A Distancias entre cadenas y subcadena común mśs larga

Implementar las siguientes funciones de Programación Dinámica (PD)

1. Escribir una función

   ```
   def edit_distance(str1: str_1, str_2: str) -> int:
       pass
   ```

   que devuelva la distancia de edición entre las cadenas `str_1` y `str_2` utilizando la menor cantidad de memoria posible.

2. Escribir una función

   ```
   def max_subsequence_length(str_1: str, str_2: str)-> int:
       pass
   ```

   que devuelva la longitud de una subsecuencia común a las cadenas `str_1` y `str_2` no necesariamente consecutiva. Dicha función deberá usar la menor cantidad de memoria posible.

3. Escribir una función

   ```
   def max_common_subsequence(str_1: str, str_2: str) -> str:
       pass
   ```

   que devuelva una subcadena común a las cadenas `str_1` y `str_2` aunque no necesariamente consecutiva.

II.B Multiplicación de matrices

Aplicar PD para determinar el mínimo número de productos necesarios para multiplicar una lista de matrices.

1. Escribir una función

   ```
   def min_mult_matrix(l_dims: List[int])-> int:
       pass
   ```

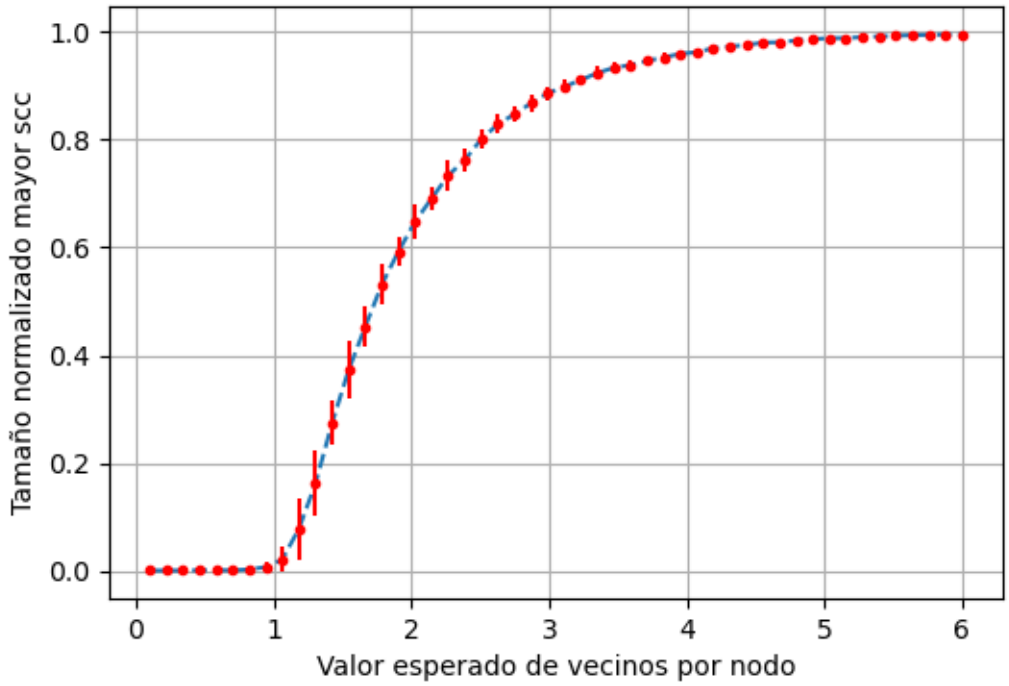
que devuelva el mínimo número de productos para multiplicar `n` matrices cuyas dimensiones están contenidas en la lista `l_dims` con `n+1 ints`, el primero de los cuales nos da las filas de la primera matriz y el resto las columnas de todas ellas.

III. OPCIONALES (PARA LOS/AS ATREVIDOS/AS)

Cada uno de estos problemas se puntuará con un punto extra añadido a la nota final (pero el límite de 10 sigue vigente...sorry)

El tamaño de la mayor de las componentes fuertemente conexas de un grafo dirigido es una variable aleatoria: si generásemos `n_rep` grafos invocando `n` veces a la función `erdos_renyi()` con el mismo valor de los parámetros `n` y `m` obtendríamos diferentes valores. Por tanto, cuando en la gráfica anterior se representaba el tamaño de la mayor de las componentes fuertemente conexas frente a `m`, se debería indicar algún intervalo de confianza. Piensa en algún intervalo de confianza que sea representativo de la variabilidad y vuelve a representar los datos y discute rigurosamente los resultados.

Ejemplo:



IV. QUÉ HAY QUE ENTREGAR

Un fichero comprimido con **tar+gzip** o **pkzip** (**evitar** por favor formatos raros tales como `.rar` o `.7z`) cuyo nombre sea

`p3NN.{tar.gz|zip}`

donde **NN** indica el número de pareja. El contenido del fichero contendrá *en una sola carpeta* lo siguiente:

1. Un fichero llamado `p3NN.py` (cuidado: respetad las mayúsculas, el Linux es importante) con las funciones que se piden en los apartados I-II. El fichero debe contener sólo las funciones: en el momento en que se hace un `import` no debe ejecutar nada. Las funciones deben devolver los valores que se piden *sin imprimir en ningún caso nada en la pantalla.*
2. Todos los ficheros `.py` auxiliares que hayáis creado para el correcto funcionamiento de esas funciones, es decir, todo los ficheros vuestros que se incluyen en el código principal con un `import`. Para comprobar que todo esté bien: cuando desde un fichero se haga `import p3NN` todo se debe importar correctamente.
3. Si se ha implementado el opcional, un fichero llamado `p3NN_optional.py` con todo el código implementado para la parte III. Valen para este fichero las mismas normas que para el fichero `p3NN.py`
4. Un archivo `p3NN.pdf` con una breve memoria que contenga las respuestas a las cuestiones planteadas de la práctica.