Puntos de articulación. Árboles abarcadores mínimos.

Diseño y análisis de algoritmos 2019-2020

Práctica 2

ÍNDICE

Fecha límite de entrega: domingo 17 de noviembre de 2019, 23:59 horas

I. MULTI-GRAPHS

I-A. Directed Multi Graphs in NetworkX

NetworkX uses a "dictionary of dictionaries of dictionaries of dictionaries" as the basic multi-graph data structure. More precisely:

- The keys are nodes so `G[u]` returns an adjacency dictionary keyed by neighbor to the edge attribute dictionary.
- The expression `G[u][v]` returns the edge attribute dictionary itself. In its simplest form it contains a dict with keys `0, 1, 2, ...`, one per each multi-edge, containing a dict with edge information (just the key `'weight'` with the edge's weight in the simplest case).

The basic object we are going to work with is the `MultiDiGraph` class for directed multi-graphs. A directed multi-graph is initialized as

```
d_g = nx.MultiDiGraph()
```

There are several methods to add nodes and vertices to `mg`. A particularly simple one is using the method `add_weighted_edges_from(l_edges)`. For instance for the above graph we can define the list of tuples `(i, j, w_ij)`

```
l_e = [(0, 1, 10), (0, 2, 1), (1, 2, 1), (2, 3, 1), (3, 1, 1)]
```

and then apply

```
d_g.add_weighted_edges_from(l_e)
d_g.add_weighted_edges_from(l_e)
```

to obtain a multi-graph with tow copies of each edge. We can check the results, for instance, by `g[0][1]`.

I-B. *Enlarging Our Graph Data Structure*

To handle multi-graphs, we are going to simplify on this and just adapt our graph data structure as a "dict of a dict of a dict", where for a graph `g` the dict `g[u][v]` will contain a new dict with keys `0, 1, 2, ...` and where `g[u][v][i]` will contain the weight of the `i`–th edge connecting `u` and `v`. Observe that we should also allow for self–pointing edges, i.e., edges whose two extremes coincide.

In order to check our multigraph functions it is convenient for us to have auxiliary multigraph conversion and generation functions.

- Write a function `graph_2_multigraph(d_g)` that converts a graph `d_g` that uses our previous dict based data structure to the new one for multi-graphs.
- Write a function
  `rand_weighted_multigraph(n_nodes, prob=0.2, num_max_multiple_edges=1, max_weight=50., decimals=0, fl_unweighted=False, fl_diag=True)`
  that generates a directed multigraph dict with `n_nodes` nodes, a probability `prob` of having links between two nodes, at most `num_max_multiple_edges` between two nodes, maximum weights `max_weight` and with `decimals` float decimal digits. When `fl_unweighted` is `True`, the function will generate only edges with weights 1, and `fl_diag` allows to have self connected edges if `True`.
- Similarly, write a function
  `rand_weighted_undirected_multigraph(n_nodes, prob=0.2, num_max_multiple_edges=1, max_weight=50., decimals=0, fl_unweighted=False, fl_diag=True)`
  that generates an **undirected** multigraph dict with the same structure as in the previous function.

II. Puntos de Articulación

II-A. *Detección de puntos de articulación*

Vamos a implementar una serie de funciones con el fin de detectar los puntos de articulación de un grafo no dirigido conexo.

- Escribir una función
  `o_a_tables(u, d_g, p, s, o, a, c)`
  donde `u, d_g, p, s, o, a, c` son respectivamente un vértice de un grafo, el diccionario del grafo, dicts de previos, vistos, orden de paso y ascenso, y un contador que se usa para actualizar el dict de orden. La función debe implementar una modificación adecuada de BP para actualizar `c` y los dicts `p, s, o, a`, y debe devolver `c`.
- Escribir una función
  `p_o_a_driver(d_g, u=0)`
  que inicialize los dicts `p, s, o, a` y el contador `c`, arranque `o_a_tables` sobre `d_g` desde `u` y devuelva `p, o, a`.
- Escribir una función
  `hijos_bp(u, p)`
  que devuelva una lista con los hijos de `u` en el árbol definido por `p` identificando, por ejemplo, los hijos de manera directa.
- Escribir una función
  `check_pda(p, o, a)`
  que compruebe si el dict `p` es compatible con un gafo conexo. De ser así, devolverá una lista con los puntos de articulación derivados del análisis de los dicts `p, o, a`. Si no es así, devolverá `None`.

II-B. *Coste de la detección de puntos de articulación*

Vamos a estudiar el rendimiento del algoritmo de detección de puntos de articulación midiendo su tiempo de ejecución sobre grafos conexos trabajando de nuevo con grafos "bastante"densos, en el sentido de que se han generado con valores de `prob` cercanos a 1. Escribir para ello una función

`time_pda(n_graphs, n_nodes_ini, n_nodes_fin, step, prob)`

que genere grafos no dirigidos aleatorios y devuelva una lista con los tiempos medios de ejecución de `p_o_a_driver` correspondientes a cada número de nodos entre `n_nodes_ini, n_nodes_fin` incrementando éste según `step`.

Dicha función visitará los grafos a partir del vértice 0 y sólo tendrá en cuenta los tiempos de aquellos grafos que sean conexos, devolviendo una lista de tiempos vacía si para algún número de nodos ningún grafo considerado es conexo.

II-C. *Cuestiones sobre puntos de articulación*

1. Tomando como base el código de las funciones `o_a_tables` y `p_o_a_driver`, dar razonadamente una estimación teórica del coste de detectar si un grafo conexo tiene o no puntos de articulación. Contrastar este análisis con las gráficas a elaborar mediante la función `time_pda` considerando únicamente grafos con `prob` 0.7 y 0.9.
2. ¿Tiene sentido el concepto de punto de articulación en multigrafos? Si crees que sí, arguméntalo. ¿Cómo los definirías? ¿Y qué habría que cambiar en las funciones anteriores para que funcionen en multigrafos?

III. TAD Conjunto Disjunto

Vamos a implementar un conjunto disjunto (CD) `S` sobre un conjunto universal `{0, 1, ..., N-1}` con `N` índices utilizando como estructura de datos un array de rangos negativos según se ha descrito en clase.

- Escribir una función
  `init_cd(d_g)`
  que devuelve un dict con las claves de `d_g` y valores -1.
- Escribir una función
  `union(rep_1, rep_2, d_cd)`
  que devuelve el representante del conjunto obtenido como la unión por rangos de los representados por los índices `rep_1, rep_2` en el CD almacenado en el dict `d_cd`.
- Escribir una función
  `find(ind, d_cd, fl_cc)`
  que devuelve el representante del índice `ind` en el CD almacenado en en el dict `d_cd` sin realizar o realizando compresión de caminos según `fl_cc` sea `False` o no.

IV. Árboles abarcadores mínimos

IV-A. Algoritmo de Kruskal

El primer paso en el algoritmo de Kruskal es insertar las distintas ramas de un grafo en una cola de prioridad.

- Utilizando el modelo de cola de prioridad de la primera práctica, escribir una función
  `insert_pq(d_g, q)`
  que inserte en la cola de prioridad `q` las ramas del grafo no dirigido `d_g` para las que `u < v`, con el fin de no tener copias equivalentes de la misma rama.
- Completar a continuación el desarrollo del algoritmo de Kruskal escribiendo una función
  `kruskal(d_g, fl_cc=True)`
  que devuelve, si lo hay, un árbol abarcador mínimo como un grafo sobre un diccionario con las mismas claves que `d_g` y que efectúa o no CC según el valor del flag. La implementación de Kruskal debe vaciar la cola de prioridad antes de volver.
  Si no hay un tal árbol, debe devolver `None`

IV-B. Coste de Kruskal

Vamos a comparar el rendimiento del algoritmo de Kruskal de acuerdo a diferentes variantes en su implementación, trabajando con grafos no dirigidos "bastante"densos, en el sentido de que se han generado con valores de `prob` cercanos a 1, de tal manera de que tengan una probabilidad alta de ser conexos.

- Escribir una función
  `time_kruskal(n_graphs, n_nodes_ini, n_nodes_fin, step, prob, fl_cc)`
  que genera grafos no dirigidos aleatorios y devuelve una lista con los tiempos medios de ejecución correspondientes a cada número de nodos entre `n_nodes_ini, n_nodes_fin` incrementando éste según `step`.
  Dicha función sólo tendrá en cuenta los tiempos de aquellos grafos que sean conexos y devolverá una lista vacía si para algún número de nodos ningún grafo considerado es conexo.
- El tiempo de ejecución de Kruskal está dominado por la inserción en la cola de prioridad. Modificar la función anterior a una nueva
  `time_kruskal_2(n_graphs, n_nodes_ini, n_nodes_fin, step, prob, fl_cc)`
  para que mida únicamente los tiempos de ejecución del bucle de construcción del árbol abarcador.
  Utilizar para ello una función `kruskal_2` modificando de manera adecuada la función `kruskal` anterior. Como antes, esta nueva función sólo tendrá en cuenta los tiempos de aquellos grafos que sean conexos.

IV-C. Cuestiones sobre Kruskal

Contestar razonadamente a las siguientes cuestiones, incluyendo gráficas si fuera preciso.

1. Discutir la aportación al coste teórico del algoritmo de Kruskal tanto de la gestión de la cola de prioridad como la del conjunto disjunto. Intentar llegar a la determinación individual de cada aportación.
2. Contrastar la discusión anterior con las gráficas a elaborar mediante las funciones desarrolladas en la práctica.
3. ¿Tiene sentido el concepto árbol abarcador mínimo en multigrafos? Si crees que sí, ¿cómo los definirías? ¿Y qué habría que cambiar en las funciones anteriores para que funcionen en multigrafos?

V. Material a entregar y corrección

V-A. *Material a entregar*

Crear una carpeta de nombre `p2NN` donde `NN` indica el número de pareja e incorporar a la misma **únicamente** los siguientes archivos:

1. Archivo del módulo Python `grafosNN.py` que contenga las funciones desarrolladas en la práctica.
   Dicho archivo contendrá **únicamente** los `import` imprescindibles para la comprobación de la práctica (por ejemplo, NO DEBE importar `matplotlib`.
   La corrección de la práctica se efectuará en un shell Linux como Ubuntu o similares, por lo que se debe asegurar que `grafosNN.py` se pueda usar sin errores en dicho entorno.
   Los nombres y parámetros de las funciones definidas en ellos deben ajustarse EXACTAMENTE a los usados en este documento.
2. Archivo `grafosNN.html` con el resultado de aplicar al módulo Python la herramienta `pydoc3`.
3. Archivo `memoP2NN.pdf` con una breve memoria que **únicamente** contenga las respuestas a las cuestiones en formato pdf. El archivo debe incluir el número de pareja y los nombres de sus miembros.

Comprimir dicha carpeta en un archivo `.zip` o `.7z` de nombre `p2NN` **No añadir a la carpeta ninguna subestructura de subdirectorios.**

No se corregirá la práctica hasta que la entrega siga esta estructura y se podrán penalizar reenvíos debidos a no tenerse esto en cuenta.

V-B. *Corrección*

La corrección de la práctica se va a efectuar en función de los siguientes elementos:

- Ejecución de un script o notebook que reciba unos datos (parámetros, grafos concretos) para comprobación de la corrección del código en los módulos Python. Los mismos se situarán en Moodle antes de la entrega de práctica.
  Es muy importante que los nombres de funciones y argumentos, así como los valores devueltos por las distintas funciones que componen la práctica se ajusten a lo indicado en los distintos apartados anteriores de este guión.
  No se corregirá la práctica mientras estos scripts no se ejecuten correctamente, penalizándose segundas entregas debidas a esta causa.
- Revisión de la documentación del código contenida en los archivos html generado mediante `pydoc` con los docstrings y otros elementos de los módulos a entregar.
  Las docstrings deben cuidarse particularmente.
- Revisión de una pequeña selección de las funciones Python contenidas en los módulos.
- Revisión de la memoria de resultados con las respuestas a las cuestiones anteriores.