Introducción a Python. Algoritmos básicos en grafos

Diseño y análisis de algoritmos 2019-2020

Práctica 1

Fecha límite de entrega: domingo 25 de octubre de 2020, 23:59 horas.
Usar únicamente los módulos importados en el script de comprobación de la práctica.

ÍNDICE

I. A Jupyter Notebook Environment

First of all, Jupyter notebooks are much more a tool to communicate results than a development environment. However, when starting with Python and dealing with small projects, notebooks can be useful to accelerate first steps and, in any case, they are a tool that every Python programmer should know about. (However, if you intend to use notebooks for serious development, watch first the I Don't Like Notebooks presentation by Joel Grus.)

Here we will describe a minimal notebook environment for small project programming in Python, Some things we almost always have to do are:

- Tell the notebook to draw pictures.
- Automatically reload some modules we may be working with.
- Move to some directory of our interest or list its contents.

We can do these inside a notebook using IPython's **magic functions** writing in a single cell commands such as:

```
#plot on the notebook
%matplotlib inline

#auotmatic module reloads
%load_ext autoreload
%autoreload 2

#some system commands
%cd D:\practicas_DAA_2017\datos_grafos
%ls
```

Next, we would import the standard modules we will work with and also the modules we are developing; it is also useful to enlarge Python's path with the dirs where our own modules may be. This should be done in another notebook cell with commands such as:

```
import sys
import time
import matplotlib.pyplot as plt
import random
import numpy as np

from sklearn.linear_model import LinearRegression
```

```
sys.path.append(r"D:\practicas_DAA_2017")
import grafos as gr
```

After this we are ready to start working with cells for either code or documentation. In code cells we will

- Edit sentences or functions.
- Execute them with `Ctrl+Intro` .
- Debug, re–edit and re–execute until OK.
- Draw pictures with matplotlib commands.

Text cells have to be marked as Markdown cells with `Esc+m` . In them we can format text with Markdown syntax for headings, lists and other typesetting actions. They also admit formulas with LaTeX notation

Finally, notebooks can be saved as such, downloaded as plain html files or converted to LaTeX using `nbconvert` (and then, say, to pdf). Their Python code can also be downloaded as a `.py` file.

We can find more on Jupyter Notebooks in The Jupyter notebook.

The Jupyter Notebook interface has a number of useful commands. One particularly handy is `Kernel | Restart & Run All` . The reason is that they are not stateless and whatever it is executed in one cell affects all others. Cell numbers help to control this and they should always be in consecutive order.

A simple way to ensure this is to run `Kernel | Restart & Run All` , that restarts the Python interpreter and runs all cells in consecutive order. If this stops before its end, something was wrong somewhere!

II. Multi-graphs

The NetworkX library offers a wide functionality to study graphs and to apply a large set of graph algorithms. It offers data structures for standard graphs in which there is at most a single edge between two nodes but also an expanded one to work with multi graphs, i.e., graphs where there may be several edges between two nodes and even edges connecting a node to itself.

We will work here with a simplified version of this data structure.

II-A. Directed Multi Graphs in NetworkX

NetworkX is a widely used Python library for working with graphs and multigraphs. We won't useit in this practice but it motivates our multigraph data structure.

NetwrokX uses a "dictionary of dictionaries of dictionaries of dictionaries" as the basic multi-graph data structure. More precisely, for a graph `G`:

- 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 which we are going to imitate is the `MultiDiGraph` class for directed multi-graphs. A directed multi-graph is initialized as

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

There are several NetworkX 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)
```

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

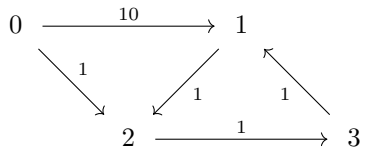
II-B. Our Multigraph Data Structure

To handle multi-graphs, we are going to simplify on NetworkX's multigraph data structure and use as our graph data structure a "dict of a dict of a dict", where for a graph `g`

- There is a dict `g[u]` for a node `u` in the graph which, in turn and if not empty, contains as keys the nodes `v` which can be acceded from `u`.
- Such a dict `g[u][v]` will contain a new dict with keys `0, 1, 2, ...` and where the value `g[u][v][i]` at the key `i` will contain the weight of the `i`–th edge connecting `u` and `v`.

Observe that we also allow for self–pointing edges, i.e., edges whose two extremes coincide. Moreover, in a standard graph (i.e., not a multigraph one), the only possible key at `g[u][v]` is `0`. Notice also that in directed unweighted multigraphs, all edge weights will be 1.

For instance, for the graph



our dict based representation would be

```
mg = {
      0: {1: {0: 10}, 2: {0:1}},
      1: {2: {0: 1}},
      2: {3: {0: 1}},
      3: {1: {0: 1}}
     }
```

In order to work with our multigraphs we need auxiliary multigraph generation and handling functions.

1. Write a function

   ```
   rand_weighted_multigraph(n_nodes,
                            probability=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, maximum weights `max_weight` and with `decimals` float decimal digits. There is a probability `probability` of having links between two nodes, and `fl_diag` allows to have self connected edges when it is `True`; in either case, when linked, there are at least one and at most `num_max_multiple_edges` between two nodes. Also, when `fl_unweighted` is `True`, the function will generate an unweighted graph, that is, only edges with weights 1.
   Use the appropriate functions from `numpy.random` to generate the random elements and write your choices on the function's docstring.
2. Similarly, write a function

   ```
   rand_weighted_undirected_multigraph(n_nodes,
                                       probability=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.
3. It is not easy to visualize large multigraphs, but for simpler ones we may try to print its adjacency list. Write a function
   `print_adj_list_mg(mg)`
   which prints the graph nodes and their adjacency lists of the multigraph `mg`, using a dash `-` to separate the list members.

III. Distancias Mínimas en Multigrafos

III-A. Programming and Timing Dijkstra

El algoritmo de Dijkstra requiere trabajar con colas de prioridad, para lo que vamos a usar las definidas en la clase `PriorityQueue` del módulo Python `Queue` y sus primitivas `empty, put, get`. En una cola PQ insertaremos tuplas `(d, i)` donde `d` es un float que indica la prioridad e `i` un índice. En Dijkstra `d` será el valor de distancia en un momento dado entre el nodo inicial y el nodo de índice `i`.

1. Escribir una función
   `dijkstra_mg(mg, u)`
   que para un multigrafo `mg` devuelva
   - un diccionario `d_dist` donde `d_dist[v]` contiene la distancia mínima de `u` a `v` y
   - otro diccionario `d_prev` donde `d_prev[v]` contiene el previo de un vértice accesible `v`.
2. Escribir una función
   `min_paths(d_prev)`
   que devuelve un diccionario `d_path` donde `d_path[v]` contains a list with the path from the starting node to node `v`.

El coste teórico del algoritmo de Dijkstra es $O(|E|\log|V|)$ que a su vez, y para grafos estándar (esto es, que no tengan una estructura de multigrafo) es $O(N^2 \log N)$ con $N = |V|$. Vamos a intentar comprobar si el tiempo de ejecución sigue esa pauta. Para ello vamos a generar `n_graphs` grafos estándar con un número de nodos entre `n_nodes_ini` y `n_nodes_fin` con

pasos de tamaño `step` y una cierta probabilidad `prob`, y para cada uno de estos grafos vamos a ejecutar la función `dijkstra_mg` tomando cada uno de sus vértices como nodo inicial.

1. Write a function

```
time_dijkstra_mg(n_graphs,
                 n_nodes_ini,
                 n_nodes_fin,
                 step,
                 num_max_multiple_edges=1,
                 prob=0.2)
```

   that generates the above described graphs and applies `dijkstra_mg` on them starting from all the graph's nodes and uses the `time()` method to return a list with the average times that have been needed at each step.
   Use the appropriate functions from `numpy` whenever needed.

III-B. Cuestiones sobre Dijkstra

Responder a las siguientes cuestiones incluyendo gráficas cuando sea necesario.

1. Tanto el problema de distancias mínimas como el algoritmo Dijkstra se aplican en principio a grafos estándar, esto es, sin ramas que auto-apunten a un vértice y con a lo sumo una rama entre dos nodos distintos. ¿Cómo se definiría el problema de distancias mínimas en multigrafos? ¿Cómo habría que modificar el algoritmo de Dijkstra si fuera necesario?
2. For a probability `prob=0.5`, apply the function `time_dijkstra_mg`, fit a linear model $An^2 \log n + B$ to the times in the returned list and plot the real and fitted times discussing the results. To fit the the model, use the template in the Scikit-learn section of the Python notes.
3. Para un grafo estándar generado con una probabilidad ρ, argumentar que podemos esperar que $|E| \simeq \rho|V|^2$. Expresar el coste de Dijkstra sobre un grafo estándar en función del número de nodos $|V|$ y la probabilidad ρ del grafo en cuestión. Para un número de nodos fijo **adecuado**, ¿cuál es el crecimiento del coste de Dijkstra en función de ρ? Ilustrar este crecimiento ejecutando Dijkstra sobre grafos con un número fijo de nodos y sparse factors `0.1`, `0.3`, `0.5`, `0.7`, `0.9` y midiendo los correspondientes tiempos de ejecución.
4. ¿Cuál es el coste teórico del algoritmo de Dijkstra iterado para encontrar las distancias mínimas entre todos los vértices de un grafo estándar?
5. ¿Cómo se podrían recuperar los caminos mínimos si se utiliza Dijkstra iterado?

IV. All Pairs Minimum Distances

IV-A. Dijkstra vs Floyd–Warshall

Vamos a comparar el rendimiento del algoritmo de Floyd–Warshall contra la aplicación de Dijkstra sobre todos los vértices trabajando con dicts de adyacencia de grafos estándar "bastante"densos, en el sentido de que se han generado con valores de `prob` relativamente cercanos a 1.

1. Escribir una función

   `dijkstra_all_pairs(g)`

   que reciba un multigrafo y devuelva una matriz cuya fila i contenga las distancias mínimas d_{ij} entre el nodo i y los demás nodos j del grafo.
2. Escribir una función

   `dg_2_ma(g)`

   que reciba un multigrafo y devuelva una matriz de adyacencia considerando solo la primera rama entre dos nodos en el caso de que haya varias.
3. Escribir una función

   `floyd_warshall(ma_g)`

   que reciba la matriz de adyacencia de un grafo y devuelva una matriz cuya fila i contenga las distancias mínimas d_{ij} entre el nodo i y los demás nodos j del grafo.
4. Write a function

```
time_dijkstra_m_all_pairs(n_graphs,
                          n_nodes_ini,
                          n_nodes_fin,
                          step,
                          prob=0.5)
```

   that generates random graphs as described in Section III graphs and applies all pairs Dijkstra returning a list with the times needed at each step.
5. Write a similar function

```
time_floyd_warshall(n_graphs,
                    n_nodes_ini,
                    n_nodes_fin,
                    step,
                    prob=0.5)
```

that now applies Floyd–Warshall to the adjacency matrices of graphs generated as in item (4) and returns a list with the times needed at each step.

IV-B. Cuestiones sobre Dijkstra iterado y Floyd–Warshall

1. ¿Cuál es el coste teórico del algoritmo Floyd–Warshall para la misma tarea?
2. Suponiendo que se va a trabajar con grafos con una probabilidad ρ, ¿para qué valores de ρ y del número de nodos N debería ser Floyd–Warshall más competitivo que Dijkstra iterado? Contrastar las conclusiones obtenidas con ejemplos y gráficas concretos obtenidos mediante grafos de diferentes tamaños N y $\rho = 0{,}5$.
3. ¿Cómo se podrían recuperar los caminos mínimos si se utiliza Floyd–Warshall?

V. Material a entregar y corrección

V-A. Material a entregar

Crear una carpeta de nombre `p1NN` donde `NN` indica el número de pareja e incorporar a la misma **únicamente** los siguientes archivos:

1. Archivo del módulo Python `grafosNN.py`.
   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 `pydoc`.
3. Archivo `memoP1NN.html` o `memoP1NN.pdf` con una breve memoria que contenga las respuestas a las cuestiones en formato html o pdf.

Comprimir dicha carpeta en un archivo `.zip` o `.7z` de nombre `p1NN` **No añadir a la carpeta ninguna subestructura de subdirectorios.**

No se corregirá la práctica hasta que la entrega siga esta estructura.

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. Asimismo, el código Python debe formatearse de acuerdo al estándar PEP-8. Usar para ello un formateador como `autopep8` o `black`.
- 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.