Introducción a Python. Algoritmos básicos en grafos

Diseño y análisis de algoritmos 2019-2020

Práctica 1

Fecha límite de entrega: domingo 13 de octubre de 2019, 23:59 horas

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 Juoyter Notebook interface has a number of useful commands. One particularly useful 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.

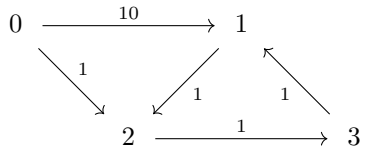
A simple way to ensure this is to run `Kernel | Restart & Run All` , that restarts the Python interpreter and runs all cells in consecutive order. If this stops before its end, something was wrong somewhere!

II. GRAPH BASICS

En esta parte vamos a trabajar con grafos ponderados usando dos EdD:

- Una matriz numpy de adyacencia, donde el elemento `i, j` indica el peso c_{ij} de la rama `(i, j)`
- Un diccionario de diccionarios donde las claves del primer diccionario `G` son índices de nodos, las claves de los diccionarios `G[i]` son los vértices de su lista de adyacencia y un diccionario `G[i][j]` contiene el peso de la rama `(i, j)`.

Por ejemplo, para el gráfico



el código

```
l = [[0, 10, 1, np.inf],
[np.inf, 0, 1, np.inf],
[np.inf, np.inf, 0, 1 ],
[np.inf, 1, np.inf, 0]]
m_g = np.array(l)
```

generaría su matriz de adyacencia según se describe arriba, mientras que

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

generaría su descripción como un dict.

II-A. Generación de grafos aleatorios

La forma más sencilla de generar grafos aleatorios en memoria es como matrices de adyacencia.

- Escribir una función
  `rand_matr_pos_graph(n_nodes, sparse_factor, max_weight=50., decimals=0)`
  que devuelva una matriz de adyacencia de un grafo dirigido ponderado con `n_nodes` nodos, una proporción `sparse_factor` de ramas y `max_weight` como peso máximo. *Sugerencia: empezar con una matriz cuadrada con valores `np.inf` y recorrerla por filas cambiando una proporción `sparseFactor` de las conexiones (i, j) con $i \neq j$.*
- Escribir una función
  `cuenta_ramas(m_g)`
  que devuelva el número de ramas (excluyendo las diagonales) en el grafo dado por la matriz de adyacencia `m_g` para comprobar que la proporción de ramas generadas es la correcta.
- Para reforzar lo anterior, escribir una función
  `check_sparse_factor(n_grafos, n_nodes, sparse_factor)`
  que genere las matrices de `n_grafos` aleatorios con `n_nodes` y un cierto `sparse_factor` y devuelva la media de los `sparse_factor` reales de las matrices generadas.
- Escribir una función
  `m_g_2_d_g(m_g)`
  que devuelva el diccionario de listas de adyacencia del grafo definido por la matriz de adyacencia `m_g`.
- Escribir una función
  `d_g_2_m_g(d_g)`
  que devuelva la matriz de adyacencia del grafo definido por el diccionario de listas de adyacencia `d_g`.

II-B. Guardando y leyendo grafos

II-B1. Guardando y leyendo grafos con `pickle`: El módulo Python `pickle` permite guardar objetos Python manteniendo su estructura y recuperar la misma al leerlos. Además, es posible comprimir dichos objetos al guardarlos y descomprimirlos al leerlos combinando el módulo `pickle` con el módulo `gzip`.

- Escribir una función
  `save_object(obj, f_name='obj.pklz', save_path='.')`
  que utilice los métodos `gzip.open` y `pickle.dump` para guardar un objeto Python `obj` de manera comprimida en un fichero de nombre `f_name`.
- Escribir una función
  `read_object(f_name, save_path='.')`
  que utilice los métodos `gzip.open` y `pickle.load` para devolver un objeto Python guardado en un fichero de nombre `f_name`.

Consultar si es preciso la sección **Files and Modules** de las notas Python.

II-B2. The Trivial Graph Format: Dada la gran información representable en un grafo, hay diversos formatos de mayor o menor riqueza para almacenar grafos en archivos. Las siguiente definición está adpaptada de en.wikipedia.org/wiki/Trivial_Graph_Format es la siguiente:

> *The "Trivial Graph Format"(TGF) is a very simple way to store very simple graphs with at most one label (such as a weight) per graph element. An example of a very simple unweighted graph could be*

```
1
2
3
#
1 2
2 1
1 3
3 2
```

> *The # sign marks the end of the node list (that may be unordered) and the start of the edge list. A similar example of a very simple weighted graph could be*

```
1
2
3
#
1 2 3.
2 1 2.
1 3 -1.
3 2 0.5
```

Vamos a escribir funciones de lectura y escritura de grafos ponderados en archivos que usen el segundo formato. Entre otras cosas, las mismas requieren trabajar con archivos y cadenas Python, por lo que intentaremos usar las funciones pertinentes de los apuntes.

- Escribir una función
  `d_g_2_TGF(d_g, f_name)`
  que reciba la lista de adyacencia de un grafo ponderado y guarde dicho grafo en un archivo de nombre `f_name` en formato TGF.
- Escribir una función
  `TGF_2_d_g(f_name)`
  que lea un grafo ponderado guardado en el archivo `f_name` en formato TGF y devuelva la lista de adyacencia del mismo.

II-C. Cuestiones sobre guardado de grafos

Responder a las siguientes cuestiones.

1. Describir qué se entiende por serializar un objeto Python.
2. `json` es otro formato de serialiación de objetos. Comentar brevemente posibles diferencias entre `pickle` y `json`.
3. ¿Qué ventajas e inconvenientes tendrían las funciones `pickle` sobre las construidas mediante el formato TFG? *Responder algo pertinente y no con lugares comunes.*

III. Distancias Mínimas en Grafos

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`.

- Escribir funciones
  `dijkstra_d(d_g, u), dijkstra_m(m_g, u)`
  que para un grafo dado respectivamente por un dict y por una matriz devuelvan
  - 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`.
- 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 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 con un número de nodos entre

`n_nodes_ini` y `n_nodes_fin` con pasos de tamaño `step` y para cada uno de estos grafos vamos a ejecutar Dijkstra tomando cada uno de sus vértices como nodo inicial.

- Write a function

  `time_dijkstra_m(n_graphs, n_nodes_ini, n_nodes_fin, step, sparse_factor=.25)`

  that generates the above described graphs and applies Dijkstra over adjacency matrices and uses the `time()` method to return a list with the times needed at each step.
- Write now a second function

  `time_dijkstra_d(n_graphs, n_nodes_ini, n_nodes_fin, step, sparse_factor=.25)`

  that does the same as the previous one but working over graph dicts. Compare the previous results with those obtained using this new function. Is there any difference between the resulting times? If so, why?
- Fit a linear model $An^2 \log n + B$ to the times in the returned lists and plot the real and fitted times discussing the results.

III-B. Cuestiones sobre Dijkstra

Responder a las siguientes cuestiones incluyendo gráficas cuando sea necesario.

1. ¿Cuál es el coste teórico del algoritmo de Dijkstra? Justificar brevemente dicho coste.
2. Expresar el coste de Dijkstra en función del número de nodos y el sparse factor 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 listas de adyacencia de 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.
3. ¿Cuál es el coste teórico del algoritmo de Dijkstra iterado para encontrar las distancias mínimas entre todos los vértices de un grafo?
4. ¿Cómo se podrían recuperar los caminos mínimos si se utiliza Dijkstra iterado?

IV. The NetworkX Library

IV-A. Directed Graphs

Here we are briefly going to explore the NetworkX Library.

NetworkX uses a "dictionary of dictionaries of dictionaries" as the basic graph/network 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.

NetworkX graphs provide two interfaces to the edge data attributes: adjacency and edges. So `G[u][v]['width']` is the same as `G.edges[u, v]['width']`.

A complete user guide is available at NetworkX Reference, Release 2.1.

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

```
g = nx.DiGraph()
```

There are several methods to add nodes and vertices to `g`. 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

```
g.add_weighted_edges_from(l_e)
```

We can check the results, for instance, by `g[0]` and `g[0][1]`.

- Write a function

  `d_g_2_nx_g(d_g)`

  that receives a graph in our `dict` format and returns the equivalent NetworkX graph.
- Write a function

  `nx_g_2_d_g(nx_g)`

  that receives a NetworkX graph and returns the equivalent graph in our `dict` format.

The function `single_source_dijkstra(nx_g, u)` applies Dijkstra's algorithm to the graph `nx_g` starting from the node `u`. If so called, it returns a tuple of two dictionaries keyed by target nodes:

- The first dictionary stores distance to each target node.
- The second stores the path to each target node.

As done before for our Dijkstra algorithm,

- Write a function
  `time_dijkstra_nx(n_graphs, n_nodes_ini, n_nodes_fin, step, sparse_factor=.25)`
  that generates the above described graphs and applies NetworkX's Dijkstra over adjacency matrices and uses the `time()` method to return a list with the times needed at each step.
- As done above, fit a linear model $An^2 \log n + B$ to the times in the returned lists and plot the real and fitted times discussing the results.

IV-B. Cuestiones

Responder a las siguientes cuestiones incluyendo gráficas cuando sea necesario.

1. Show graphically the growth of the execution times of Dijkstra's algorithm as a fuction of the number of nodes and the sparsity factor using the NetworkX library. Work with graphs with 100 nodes and sparse factors `0.1, 0.3, 0.5, 0.7, 0.9`.
2. usando la librería NetworkX y trabajando sobre grafos con un número fijo de 100 nodos y sparse factors Measure and show graphically the execution times of iterated Dijkstra to find the minimum distances between all the vertices of a graph using the NetworkX library and working on graphs with a fixed number of 25 nodes and sparse factors `0.1, 0.3, 0.5, 0.7, 0.9`.

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.
- 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.