Fundamentos de Algoritmos – Trabajo Práctico 2

Master in Management + Analytics, Escuela de Negocios, UTDT

Primer semestre 2022

Consideraciones generales:

- El trabajo se debe realizar en los mismos grupos del TP1.
- Nombrar los archivos con el código fuente como se indica abajo, y subirlos al formulario *TP2: Entrega* en el campus virtual de la materia.
- Los programas entregados deben correr correctamente en `Python3`.
- La fecha límite de entrega es el **domingo 15 de mayo a las 23:55**. Los TPs entregados fuera de término serán aceptados, pero la demora incidirá negativamente en la nota.

Arbolado público de Buenos Aires

De la página `https://data.buenosaires.gob.ar/dataset/arbolado-publico-lineal`, descargar el dataset de arbolado en espacios públicos en la ciudad de Buenos Aires denominado **"Arbolado Público Lineal 2011"**. Este dataset tiene varios valores de cada árbol, incluyendo sus coordenadas geográficas, altura, diámetro, especie, nombre común, barrio y otros. Contiene información de un total de 372.699 árboles.

Atención: ¡Bajar el archivo correcto! Se llama `arbolado-publico-lineal-2011.csv`, es de 2011, está en formato CSV *(comma-separated values)* y pesa 72,3 MB. No confundir con el archivo de 2017-2018, que elegimos no usar porque muchos de sus árboles tienen valores faltantes *(missing values)*.

El objetivo de este Trabajo Práctico es escribir programas para modelar, procesar y efectuar algunas consultas sobre estos datos. En concreto, se pide:

1. Implementar una clase `Arbol` que encapsule el concepto de un árbol individual en este problema y las consultas que se le pueden hacer. Debe tener, al menos, los siguientes métodos:
   - `Arbol(...)`: construye un nuevo objeto de la clase `Arbol`, con los parámetros que se consideren necesarios.
   - `a.longitud()`
   - `a.latitud()`
   - `a.especie()` (Correspondiente a la columna `nombre_cie` del archivo CSV).
   - `a.barrio()`
   - `a.direccion()`, que consiste en la concatenación de los valores de las columnas `calle` y `chapa1`, separadas por un espacio en blanco. Ejemplo: `'Navarro 2625'`.
   - `str(a)`, que devuelve una representación como string del árbol `a`, que debe tener entre paréntesis su especie y su dirección separados por una arroba (esto se logra definiendo el método `__repr__`). Ejemplo: `'(Taxodium distichum@Navarro 2625)'`.

2. Implementar una clase `DataSetArboreo` que encapsule el concepto de una colección de datos de (muchos) árboles y las consultas que se pueden hacer sobre ella. Debe tener, al menos, los siguientes métodos:

   - `DataSetArboreo(filename)`: Construye un dataset a partir del archivo CSV pasado como argumento. Por ejemplo, `DataSetArboreo('arbolado-publico-lineal-2011.csv')` construye un objeto de la clase `DataSetArboreo` conteniendo los 372.699 árboles del archivo CSV indicado. [1]
   - `d.tamano()`: Devuelve la cantidad de árboles en el dataset `d`.
   - `d.especies()`: Devuelve el conjunto de especies de los árboles del dataset `d`.
   - `d.barrios()`: Devuelve el conjunto de barrios de los árboles del dataset `d`.
   - `d.arboles_de_la_especie(especie)`: Devuelve los árboles del dataset `d` que tienen la especie indicada.

     El método `arboles_de_la_especie` debe tener complejidad algorítmica $O(N)$ en el peor caso, donde N es la cantidad de árboles en el dataset.
   - `d.cantidad_por_especie(minimo)`: Devuelve un diccionario que indica la cantidad de ejemplares de cada especie existente en el dataset `d`, pero incluyendo solamente a las especies que tienen como mínimo la cantidad de ejemplares indicada. Por ejemplo, para el dataset completo del archivo CSV, `d.cantidad_por_especie(20000)` devuelve este diccionario:

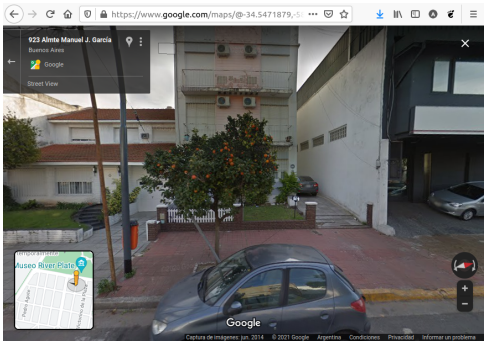
     chequear

     ```
     { 'Melia azeradach': 24558,
       'Platanus x acerifolia': 34786,
       'Ficus benjamina': 23907,
       'Fraxinus pennsylvanica': 141825 }
     ```

     Es decir, hay cuatro especies con más de 20.000 ejemplares en la ciudad de Buenos Aires: *Melia azeradach* (con 24.558 ejemplares), *Platanus x acerifolia* (34.786), *Ficus benjamina* (23.907) y *Fraxinus pennsylvanica* (141.825).

     El método `cantidad_por_especie` debe tener complejidad algorítmica $O(N^2)$ en el peor caso, donde N es la cantidad de árboles en el dataset.
   - `d.arbol_mas_cercano(especie, lat, lng)`: Devuelve el árbol de la especie indicada que está más cercano al punto $\langle$`lat`, `lng`$\rangle$ en el dataset `d`, usando la distancia euclidiana[2]. Este método tiene como precondición que exista al menos un árbol de dicha especie en el dataset `d`. Por ejemplo, considerando el dataset completo del archivo CSV, `d.arbol_mas_cercano('Citrus aurantium', -34.55472, -58.44583)` devuelve el naranjo más cercano a la Di Tella, el cual tiene número identificador 55001143 y dirección "Garcia, Manuel J., Alte. 923".

     Si buscamos manualmente esa dirección en Google Maps y vamos a Street View, encontramos el naranjo en cuestión: ☺



[1] Al crear el dataset, se debe tener en cuenta que en el archivo de entrada el carácter coma (`,`) puede aparecer no solo para separar datos, sino también como parte de los datos correspondientes a un árbol. En estos casos, el valor aparece entre comillas dobles. Por ejemplo, el valor correspondiente al campo `calle` del primer registro de `arbolado-publico-lineal-2011.csv` es `"Calvo, Carlos"`.

[2] La distancia eulidiana entre dos puntos $\langle x_1, y_1 \rangle$ y $\langle x_2, y_2 \rangle$ se define como $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

El método `arbol_mas_cercano` debe tener complejidad algorítmica $O(N)$ en el peor caso, donde N es la cantidad de árboles en el dataset.

chequear

- `d.exportar_por_especie_y_altura(filename, especie, alt_min, alt_max)`: Genera un archivo con nombre `filename` que incluye el listado de los árboles del dataset `d` de la especie indicada cuya altura es mayor o igual a `alt_min` y menor o igual a `alt_max`. Los árboles aparecen uno por línea y ordenados por altura en forma ascendente. Cada línea muestra primero la altura del árbol, a continuación un punto y coma y finalmente la representación como `str` del árbol (Ejemplo: `15;(Taxodium distichum@Navarro 2625)`).
  En la implememntación de este método está permitido (y sugerido) usar `sort()` de la clase `list` de Python.

Se deben entregar los siguientes archivos:

- **`arbol.py`**, con la definición de la clase `Arbol` (este archivo no debe contener código principal).
- **`dataset.py`**, con la definición de la clase `DataSetArboreo` (este archivo no debe contener código principal).
- **`informe.pdf`**, un documento en el cual se explique por qué los métodos `arboles_de_la_especie`, `cantidad_por_especie` y `arbol_mas_cercano` cumplen con los órdenes de complejidad requeridos. Incluir también cualquier aclaración adicional que se considere necesaria sobre cualquier parte del trabajo. Se espera que este documento sea conciso, de tres o cuatro páginas, y en formato PDF.

La carpeta adjunta `templates` contiene archivos de ejemplo para usar como referencia.

Observaciones:

- El código fuente debe estar bien comentado.
- Definir todas las funciones auxiliares que se consideren necesarias.
- No está permitido importar bibliotecas. Solo pueden usarse elementos de Python vistos en clase. (Ante la duda, consultar.)
- Los archivos `arbol.py` y `dataset.py` no deben tener código principal; solamente definiciones de clases. El objetivo es que dichos archivos sean importados y usados por otros programas, como parte de un proyecto más grande.
- Para los cálculos de complejidad algorítmica considerar los órdenes de peor caso documentados en `https://wiki.python.org/moin/TimeComplexity`.
- Durante el desarrollo y para probar las implementaciones, se recomienda **evitar trabajar sobre el archivo CSV completo**. Su lectura demora tiempo en cada ejecución y resulta difícil saber si los resultados son correctos. En cambio, conviene crear archivos CSV pequeños, conteniendo pocos registros tomados del archivo original, para tener mayor control de las ejecuciones.
- En la evaluación del trabajo, se tendrá en cuenta no solamente que el código sea correcto, sino también que sea claro, ordenado y modular y que cumpla los órdenes de complejidad requeridos (con las justificaciones correspondientes).
- Puede suponerse que el usuario siempre usará los objetos y sus métodos de manera correcta. Es decir, si hay errores en la cantidad, tipo o valor de los argumentos, no se espera ningún comportamiento particular (la ejecución podría colgarse o terminar en un error, por ejemplo).