Conjuntos Disjuntos y Componentes Conexas. El Problema del Viajante

Algoritmos y Estructuras de Datos Avanzadas 2022-2023

Practica 2

Fecha de entrega: 11 de noviembre de 2022

ÍNDICE

I. TAD Conjunto Disjunto y Componentes Conexas

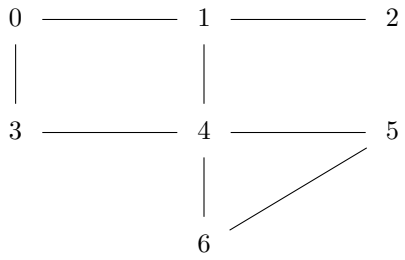
I-A. TAD Conjunto Disjunto

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 o bien de padres o bien de rangos negativos según se ha descrito en clase.

- Escribir una función
  `init_cd(n: int)-> np.ndarray:`
  que devuelve un array con valores -1 en las posiciones `{0, 1, ..., n-1}`.
- Escribir una función
  `union(rep_1: int, rep_2: int, p_cd: np.ndarray)-> int:`
  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 array `p_cd`.
- Escribir una función
  `find(ind: int, p_cd: np.ndarray)-> int:`
  que devuelve el representante del índice `ind` en el CD almacenado en `p_cd` realizando compresión de caminos.
- Aunque eficaz, nuestra implementación de CD hace difícil identificar visualmente los subconjuntos de un CD. Escribir una función
  `cd_2_dict(p_cd: np.ndarray)-> Dict:`
  que reciba un CD en el array `p_cd` y devuelva un diccionario cuyas claves sean los representantes de los subconjuntos del CD y donde el valor de la clave `u` del dict sea una lista con los miembros del subconjunto representado por `u`, incluyendo, por supuesto el propio `u`.

I-B. CDs y Componentes Conexas

Una aplicación del TAD CD es encontrar las componentes conexas de grafos no dirigidos. Para nosotros un tal grafo va a ser un conjunto de vértices V y de ramas E tal y como se muestra en la figura inferior:



Hay diversas formas de definir una estructura de datos para grafos no dirigidos. Aunque no es muy eficaz, vamos a trabajar simplemente con un `int n` que nos indica que V va a venir dado por la lista `[0, ..., n-1]` y una lista `l` con las ramas. Por ejemplo, para el grafo anterior `n=7` y la lista podría ser

```
[(0, 1), (0, 3), (1, 2), (1, 4), (3, 4), (4, 5), (4, 6), (5, 6)]
```

Un grafo se dice **conexo** si hay un camino entre cualesquiera dos vértices. A su vez, las **componentes conexas** (CCs) de un grafo son sus subgrafos conexos maximales. En el grafo anterior solo hay una CC pero, por ejemplo, en el caso extremo de un grafo con `5` vértices y una lista vacía de ramas, habría 5 CCs, una para cada nodo. Queremos desarrollar una función que reciba un grafo no dirigido con la EdD anterior y nos devuelva un dict con sus CCs según describimos a continuación.

- Escribir una función

  `ccs(n: int, l: List)-> Dict:`

  que nos devuelva las componentes conexas de un tal grafo. Para ello la función inicializará un CD vacío, examinará las ramas del grafo contenidas en la lista `l` y si los dos nodos de la rama pertenecen a subconjuntos distintos, unirá estos. Cuando se procesen todas las ramas la función convertirá la tabla que contenga el CD en un dict y devolverá éste.

I-C. Cuestiones sobre CDs y CCs

Contestar razonadamente a las siguientes cuestiones.

1. Sin darnos cuenta, en nuestro algoritmo de encontrar CCs podemos pasar listas con ramas repetidas o donde los vértices coinciden con los de una que ya está aunque en orden inverso. ¿Afectará esto al resultado del algoritmo? ¿Por qué?
2. Argumentar que nuestro algoritmo de encontrar componentes conexas es correcto, esto es, que a su final en los distintos subconjuntos disjuntos se encuentran los vértices de las distintas componentes del grafo dado.
3. El tamaño de un grafo no dirigido viene determinado por el número `n` de nodos y la longitud de la lista `l` de ramas. Estimar razonadamente en función de ambos el coste del algoritmo de encontrar las componentes conexas mediante conjuntos disjuntos.

II. El problema del viajante de comercio

II-A. Algoritmo del Vecino Más Cercano

Vamos a explorar el algoritmo codicioso basado en el vecino más cercano para encontrar un circuito que dé una solución razonable al problema del viajante (Travelling Salesman Problem, TSP).

- Escribir una función

  `dist_matrix(n_nodes: int, w_max=10)-> np.ndarray`

  que genere la matriz de distancias de un grafo con `n_nodes` nodos, valores **enteros** con un máximo `w_max` ; observar que dicha matriz debe ser simétrica con diagonal 0. Usar para ello funciones de Numpy como `np.random.randint` o `fill_diagonal` .
- Escribir una función

  `greedy_tsp(dist_m: np.ndarray, node_ini=0)-> List`

  que reciba una matriz de distancias y un nodo inicial y devuelva un circuito codiciosos como una lista con valores entre `0` y el número de nodos menos 1.
- Escribir una función

  `len_circuit(circuit: List, dist_m: np.ndarray)-> int`

  que reciba un circuito y una matriz de distancias y devuelva la longitud de dicho circuito.
- **TSP repetitivo.** Una forma sencilla de mejorar nuestro primer algoritmo TSP codicioso es aplicar nuestra función `greedy_tsp` a partir de todos los nodos del grafo y devolver el circuito con la menor longitud. Escribir una función

  `repeated_greedy_tsp(dist_m: np.ndarray)-> List`

  que implemente esta idea.

- **TSP exhaustivo.** Para grafos pequeños podemos intentar resolver TSP simplemente examinando todos los posibles circuitos y devolviendo aquel con la distancia más corta. Escribir una función

  `exhaustive_tsp(dist_m: np.ndarray)-> List`

  que implemente esta idea usando la librería `itertools`. Entre los métodos de iteración implementados en la biblioteca, se encuentra la función `permutations(iterable, r=None)` que devuelve un objeto iterable que proporciona sucesivamente todas las permutaciones de longitud `r` en orden lexicográfico. Aquí `r` es por defecto la longitud del iterable pasado como parámetro, es decir, se generan todas las permutaciones con `len(iterable)` elementos.

II-B. Cuestiones sobre la solución greedy de TSP

Contestar razonadamente a las siguientes cuestiones.

1. Estimar razonadamente en función del número de nodos del grafo el coste codicioso de resolver el TSP. ¿Cuál sería el coste de aplicar la función `exhaustive_greedy_tsp` ? ¿Y el de aplicar la función `repeated_greedy_tsp` ?
2. A partir del código desarrollado en la práctica, encontrar algún ejemplo de grafo para el que la solución greedy del problema TSP no sea óptima.

III. Material a entregar y corrección

III-A. Material a entregar

Crear una carpeta con el nombre `p2NN` donde `NN` indica el número de pareja y añadir en ella **sólo** los siguientes archivos:

1. Un archivo Python `p2NN.py` con el código de las funciones desarrolladas en la práctica (y NO elementos como scripts de medida de tiempo o dibujo de curvas) así como los `imports` estrictamente necesarios, a saber:

   ```
   import numpy as np
   import itertools

   from typing import List, Dict, Callable, Iterable
   ```

   Los nombres y parámetros de las funciones definidas en ellos deben ajustarse EXACTAMENTE a los utilizados en este documento.
   Además, todas las definiciones de funciones deben incorporar los type hints adecuados.
2. Un fichero `p2NN.html` con el resultado de aplicar el comando `pdoc` del paquete `pdoc3` al módulo Python `p2NN.py` .
3. Un archivo `p2NN.pdf` con una breve memoria que contenga las respuestas a las cuestiones de la práctica en formato pdf. **En la memoria se identificará claramente el nombre de los estudiantes y el número de pareja**. **Si se añaden figuras o gráficos, DEBEN ESTAR SOBRE UN FONDO BLANCO.**

Una vez que se hayan puesto estos archivos, comprimir esta carpeta en un archivo llamado `p2NN.zip` o llamado `p2NN.7z` . **No añadir ninguna estructura de subdirectorios a la carpeta.**
La práctica no se corregirá hasta que el envío siga esta estructura.

III-B. Corrección

La corrección se hará en base a los siguientes elementos:

- La ejecución de un script que importará el módulo `p2NN.py` y comprobará la corrección de su código.
  La práctica no se corregirá mientras este script no se ejecute correctamente, penalizando las segundas presentaciones por esta causa.
- La revisión de la documentación del código contenida en los ficheros html generados por `pdoc`. **En particular, las docstrings deben ser escritas muy cuidadosamente.**
  Además, se recomienda que el código Python esté formateado según el estándar PEP-8. Utilizar para ello un formateador como `autopep8` o `black`.
- La revisión de una selección de las funciones de Python contenidas en el módulo `p2NN.py` .
- La revisión de la memoria con las respuestas a las preguntas anteriores.