Conjuntos Disjuntos y Algoritmo de Kruskal. El Problema del Viajante

Algoritmos y Estructuras de Datos Avanzadas 2023-2024

Práctica 2

Fecha de entrega: 24 de noviembre de 2023

ÍNDICE

I. TAD Conjunto Disjunto

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.

1. Escribir una función
   `init_cd(n: int)-> np.ndarray:`
   que devuelve un array con valores -1 en las posiciones `{0, 1, ..., n-1}`.
2. 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`.
3. 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.

II. Árboles abarcadores mínimos

II-A. Algoritmo de Kruskal

En esta sección vamos a implementar el algoritmo de Kruskal para encontrar un árbol abarcador mínimo de un grafo no dirigido ponderado. Para ello supondremos que en un grafo no dirigido con n vértices, estos vienen dados como índices `0, 1, ..., n-1` y que los vértices están dados por una lista `l_g` cuyas ramas se representan como tuplas `(u, v, w)` , donde `u, v` son dos `ints` que representan la rama y `w` otro `int` que da el peso de la rama `(u, v)` . Por lo tanto, nuestros grafos vendrán representados como una tupla `(n, l_g)` .

El primer paso en el algoritmo de Kruskal es insertar las distintas ramas de un grafo en una cola de prioridad. Para ello vamos a usar la clase `PriorityQueue` del módulo `queue` , que debemos de importar como

```
import queue
from queue import PriorityQueue
```

En una tal cola `pq` se inserta un elemento `item` con prioridad `w` como `pq.put((w, item))` y se extrae con `w, item = pq.get()` .

1. Escribir una función
   `create_pq(n: int, l_g: List)-> queue.PriorityQueue`
   que inserte las ramas del grafo no dirigido dado por el par `n, l_g` en una cola de prioridad y la devuelva.
2. Completar a continuación el desarrollo del algoritmo de Kruskal escribiendo una función
   `kruskal(n: int, l_g: List)-> Tuple[int, List]`
   que devuelve, si lo hay, un árbol abarcador mínimo (AAM) para el grafo `n, l_g` como un nuevo grafo `n, l_t` , donde `l_t` son las ramas de dicho árbol. La implementación de Kruskal debe vaciar la cola de prioridad antes de volver.
   Si no hay un tal árbol, debe devolver `None`

II-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 completos (esto es, donde todos los vértices están conectados entre sí).

1. Escribir una función
   `def complete_graph(n_nodes: int, max_weight=50)-> Tuple[int, List]:`
   que genere un grafo completo con `n_nodes` nodos generando ramas `u, v` con `u < v` , y pesos `w` obtenidos mediante `random.randint(1, max_weight)` . La función devolverá la tupla `n_nodes, l_g` donde en la lista `l_g` insertamos los elementos `u, v, w` que vamos generando.
2. Escribir una función
   `time_kruskal(n_graphs: int, n_nodes_ini: int, n_nodes_fin: int, step: int)-> List`
   que genera `n_graphs` grafos completos con pesos aleatorios según la función anterior y devuelve una lista con los tiempos medios de ejecución de nuestra función `kruskal` correspondientes a cada número de nodos entre `n_nodes_ini, n_nodes_fin` incrementando éste según `step`.
3. 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: int, n_nodes_ini: int, n_nodes_fin: int, step: int)-> List`
   para que mida únicamente los tiempos de ejecución asociados a la gestión del CD.
   Para ello modificar nuestra función `kruskal` para obtener una función `kruskal_2` que además de un AAM, devuelva también los tiempos de ejecución acumulados sobre las primitivas del conjunto disjunto.

II-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.
   Contrastar la discusión anterior con las gráficas a elaborar mediante las funciones desarrolladas en la práctica.
2. El algoritmo de Kruskal puede detectar que ya ha obtenido un árbol abarcador mínimo sin tener que procesar toda la cola de prioridad. Sin embargo, la misma debería vaciarse, para lo que podemos simplemente que nuestra función Kruskal siga procesando la cola de prioridad hasta vaciarla (no se añadirán nuevas ramas al AAM) o bien simplemente dejar de procesar el CD y simplemente vaciar la cola mediante `get` .
   ¿Cuál de las dos alternativas te parece más ventajosa computacionalmente? Argumen tu respuesta analizando los costes de ejecución de ambas opciones.

III. El problema del viajante de comercio

III-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). Para ello usaremos la función siguiente

```
def dist_matrix(n_nodes: int, w_max=10) -> np.ndarray:
    """
    """
    m = np.random.randint(1, w_max+1, (n_nodes, n_nodes))
    m = (m + m.T) // 2
    np.fill_diagonal(m, 0)
    return m
```

que genera la matriz de distancias de un grafo con `n_nodes` nodos, valores **enteros** con un máximo `w_max` ; observar que la función trabaja con arrays de Numpy y nos devuelve una matriz simétrica con diagonal 0.

1. 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.
2. 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.
3. **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.
4. **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.

III-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_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.

IV. Material a entregar y corrección

IV-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.

IV-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 aplicación de la herramienta `pylint` de análisis de código, que comprueba un archivo Python de acuerdo a un archivo de configuración (en nuestro caso el archivo `my_pylintrc` acccesible en Moodle y que copiaremos al directorio de la práctica) y proporciona un listado de los errores encontrados así como un score entre 0 y 10 mediante la orden

  ```
  pylint p2XX.py --rc-file my_pylint
  ```

- La revisión de una selección de las funciones de Python contenidas en el módulo `p2NN.py` , donde se tendrá en cuenta la claridad y calidad del código, su buen formateo y su documentación interna (en particular, comentarios y nombres de variables.
- La revisión de la memoria con las respuestas a las preguntas anteriores.