Algoritmos y estructura de datos avanzadas. Año Académico 2024-25

Práctica 2

Conjuntos disjuntos, árboles abarcadores, Problema del Viajante

Fecha de entrega: Grupo 126: **4 de Noviembre 2024** antes de la clase
Grupo 127: **5 de Noviembre 2024** antes de la clase

¡Atención! Muchas de las funciones de esta práctica (y de las demás) se pasarán por un script de corrección automática. Esto quiere decir que tenéis que seguir escupulosamente las indicaciones en lo que se refiere a nombres de ficheros, nombres de funciones, parámetros, valores de retorno, ficheros, etc. Las funciones no deben escribir nada en pantalla (si ponéis algunos `print` para hacer *debuging* que no se os olvide de eliminarlos antes de entregar). Los ficheros que se entregan deben contener sólo las funciones, ningún *script* de prueba ni nada que se ejecute cuando se hace el `import` del fichero.

No seguir estas indicaciones puede resultar en una penalización o, en los casos peores, en la no corrección de la práctica.

I. CONJUNTOS DISJUNTOS

I.A Implementación de la estructura

En esta primera parte implementaremos las funciones de base de la estructura de conjuntos disjuntos con unión por *rank* y compresión de caminos.

1. Escribir una función

   ```
   ds_init(n)
   ```

   que inicializa una estructura con n elementos, cada uno siendo un conjunto separado (es decir: que devuelve una lista con -1 en cada uno de sus elementos).

2. Escribir una función:

   ```
   ds_union(p_ds, rep_1, rep_2)
   ```

   que recibe una estructura de conjuntos disjuntos y dos *representantes de conjuntos* (dentro de la función hay que averiguar que los dos elementos sean raíces de árboles y devolver `None` en caso contrario). La función hace la unión por rango de los dos conjuntos representados por `rep_1` y `rep_2` y devuelve el representante del conjunto unión.

3. Escribir una función

   ```
   ds_find(p_ds, m)
   ```

   que devuelva el representante del elemento m usando compresión de camino. Verificar que m esté en el rango $0, \ldots, n-1$ y devolver `None` en caso contrario.

* * *

I.B Componentes conexas

En este apartado utilizaremos la estructura de conjuntos disjuntos para encontrar las componentes conexas de un grafo no dirigido.

1. Escribir la función

   ```
   connected(n, e)
   ```

   La función recibe un entero n (el número de nodos del grafo) y una lista de pares de enteros

   $$E = [(u_1, v_1), \ldots, (u_m, v_m)]$$

   con $u_i, v_i \in [0, n)$. Cada elemento de la lista indica que hay un arco (no dirigido) entre los nodos u_i and v_i. La función ejecutará el algoritmo de componentes conexas y devolverá la estructura de conjuntos disjuntos que resulta de la ejecución.

2. Escribir la función

   ```
   connected_count(p)
   ```

   La función recibe en entrada la estructura de conjuntos disjuntos resultante de la llamada a `connected` y devuelve el número de componentes conexas del grafo.

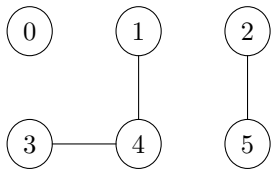
3. Escribir la función

   ```
   connected_sets(p)
   ```

   La función recibe en entrada la estructura de conjuntos disjuntos resultante de la llamada a `connected` y devuelve una lista de listas de enteros: cada una de las listas representa una componente conexa del grafo y contiene los nodos que forman parte de esa componente conexa.

   Ejemplo:

   Dado el grafo:

   representado por la lista

   ```
   lst = [ (1, 4), (3, 4), (2, 5)]
   ```

   Las llamadas:

   ```
   s = connected(6, lst)
   n = connected_count(s)
   ccp = connected_set(s)
   ```

producirá `n=3` y

```
ccp = [
  [0],
  [1, 3, 4],
  [2, 5]
]
```

II. ÁRBOLES ABARCADORES MÍNIMOS

II.A Algoritmo de Kruskal

En este apartado implementaremos el algoritmo de Kruskal para encontrar árboles abarcadores mínimos en un grafo ponderado no dirigido. El grafo en este caso se representa como (n, E), donde n es el número de nodos y E es una lista de triplas:

$$E = [(u_1, v_1, w_1), \ldots, (u_m, v_m, w_m)]$$

La tripla (u_k, v_k, w_k) indica que hay un arco (no dirigido) entre u_k y v_k con peso w_k.

Recordamos que el algoritmo de Kruskal supone el uso de una cola de prioridad. Para esto se usará el *min heap* de la práctica anterior, modificada para aceptar como entrada pares (itm, p) donde itm es un elemento (un arco, en este caso) y p es su propridad.

1. Escribir una función

   ```
   kruskal(n, E)
   ```

   que ejecute el algoritmo de Kruskal en el grafo. El algoritmo debe devolver un grafo (un árbol, lo recordamos, también es un grafo) (n, E') donde n es el número de nodos (el mismo que en grafo inicial) y E' es el conjunto de arcos que forman el árbol. Si el árbol no existe (es fácil ver que un grafo no conexto no tiene árbol abarcador), la función debe devolver `None`.

2. Escribir una función

   ```
   k_weight(n, E)
   ```

   que, dada la lista de arcos producidos por la función `kruskal`, devuelve el peso del árbol

II.B Tiempo de Ejecución

1. Un grafo de Erdös-Renyi ponderados y modificado por conexión es un grafo aleatorio con n nodos en que cada nodo tiene en media m vecinos (n y m son los parámetros que definen el grafo). En nuestro grafo consideraremos sólo pesos contenidos en el intervalo $[0, 1]$. El grafo se genera con el método siguiente:

   i) Se crea una lista de arcos vacía

   ii) para $i = 1, \ldots, n-1$, se elige un nodo aleatorio en el intervalo $[0, i-1]$ (sea m este nodo) , se genera un peso aleatorio w y se crea un arco (i, m, w) (estos arcos garantizan que el grafo final es conexo)

ii) Para $i = 0, \ldots n \times (m-1) - 1$ se eligen dos nodos aleatorios u, v que todavía no están conectado, se genera un peso aleatorio w, y se añade el arco (u, v, w) a la lista de arcos.

Se escriba una función

```
erdos_conn(n, m)
```

que, dado los parámetros n y m construya el grafo aleatorio y devuelva la lista de arcos.

Avertencia: para que las cosas funcionen sin problemas es mejor que sea $m \ll n$. Si n y m son parecidos se pueden haber problemas en generar arcos aleatorios que no estén ya generados. Hay maneras de evitar el problemas (¿sabéis encontrar una?) pero para esta práctica se pueden simplemente usar parámetros con $m \ll n$

2. Se escriba la función:

```
time_kruskal(n, m, n_graphs)
```

que, dados los parametros `n` y `m`, construye `n_graphs` grafos aleatorios de Erdös-Renyi modificados, calcula el tiempo de ejecución del algoritmo de Kruskal en cada uno de ellos, y devuelve media y varianza de tal tiempo.

3. Utilizar la función del punto anterior, con valores variados y oportunos de `n` y de `m` para estimar el tiempo de ejecución del algoritmo como función de `n` y `m`. (Sugerencia: buscar cual es el tiempo teórico de ejecución y usarlo como guia.)

III. EL 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). Recordemos que este algoritmo empieza de una ciudad elegida a priori y se desplaza cada vez a la ciudad más cercana entre las que todavía no se han visitado, hasta visitarlas todas. Para probar el problema crearemos una matriz $\mathbf{M}$ de distancias aleatorias entre ciudades, en que m_{ij} es la distancia entre la ciudad i y la ciudad j. La matriz debe ser tal que $m_{ij} > 0$ si $i \neq j$ y $m_{ii} = 0$. También $m_{ij} = m_{ji}$. La función siguiente genera la matriz

```
def dist_matrix(n_cities, w_max = 10):
    M = [ [ random.uniform(0, w_max) for _ in range(n_cities)] for _ in range(n_cities) ]
    for k in range(n_cities):
      M[k][k] = 0
      for h in range(k):
        u = (M[k][h] + M[h][k])/2.0
        M[h][k] = M[k][h] = u
    return M
```

1. Escribir una función

```
greedy_tsp(dist_m, node_ini)
```

que reciba una matriz de distancias y un nodo inicial y devuelva un circuito codiciosos como una lista con valores entre `0` y `n_cities-1` (`n_cities` es el valor usado para la creación de la matriz de distancias), y que representa la secuencias de ciudades a recorrer.

2. Escribir una función

   ```
   len_circuit(circuit, dist_m)
   ```

   que recibe un circuito tal como lo ha generado `greedy_tsp`, una matriz de distancias, y devuelve su longitud. (**Atención**: el circuito que buscamos es cerrado. Una lista que contiene las ciudades no representa un circuito cerrado en cuanto la ciudad final no es igual a la inicial. Una vez llegado a la última ciudad hay que volver a la primera, y esta distancia también hay que contarla.)

3. **TSP repetitivo.** Una forma sencilla de mejorar nuestro primer algoritmo TSP codicioso es aplicar la función `greedy_tsp` a partir de todos los nodos del grafo y devolver el circuito con la menor longitud. Escribir una funcioón

   ```
   repeated_greedy_tsp(dist_m)
   ```

   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 máscorta. Escribir una función

   ```
   exhaustive_tsp(dist_m)
   ```

   que implemente esta idea. La idea es crear todas las posibles permutaciones de la list $[0, \ldots, n-1]$, donde n es el número de ciudades (`n=len(dist_m)`). Cada permitación corresponde a un posible recorrido del viajante. Usando la función `len_circuit` podemos determinar la longitud de cada recorrido y quedarnos con el más corto.

   Para generar las permitaciones, es posible usar la libreria `itertools`. Entre los métodos de esta librería 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.

IV. OPCIONALES (PARA LOS ATREVIDOS)

Cada uno de estos problema se puntuerá con un punto extra añadido a la nota final (pero el límite de 10 sigue vigente...sorry)

IV.A Tiempo de ejecución de la cola de prioridad

En el algoritmo de Kruskal, la mayor parte del tiempo se pasa en sacar arcos de la cola de prioridad. Modificar la función `kruskal` de manera tal que mida sólo el tiempo de ejecución de las operaciones de la cola y lo compare con el tiempo de ejecución de la función total. Producir una gráfica en que se mide el porcentaje del tiempo total que se pasa en la cola.

Esta medida no es inmediata. Por un lado, hay que medir los dos tiempos separadamente: la función modificada en general no será apta para medir el tiempo total de ejecución. Se utilizará la función sin modificar, la del apartado I

de la anterior práctica, para medir el tiempo total y la función modificada para lmedir el tiempo de ejecución de las operaciones de cola.

Por otro lado, en las operaciones de cola tenemos el mismo problema que en las otras medidas: una sola ejecución no tarda lo suficiente como para medir el tiempo de ejecución con buena precisión. Por otro lado, no podemos llamar, por ejemplo, 1000 veces la función `heap_extract`, en cuanto cada ejecución modifica el *heap*.

Aquí está el desafío... a vosotros recogerlo.

IV.B Permutaciones
En el enunciado se usa una librería para generar todas las permutaciones de una lista. Pero no es tan difícil hacerlo "a mano" usando una función recursiva. Escribir una función:

```
permute(lst)
```

que recibe en entrada una lista y devuelve una lista de lista, donde cada una de esas lista es una permutación de `lst`. Por ejemplo:

```
>>> permute( [1, 2, 3] )
    [ [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1] ]
>>>
```

V. QUE HAY QUE ENTREGAR

Un fichero con nombre

`AEDATA-2024-<Apellido1>-<Apellido2>.{tar.gz|zip}`

comprimido con **tar+gzip** o **pkzip** (**evitar** por favor formados raros tales como `.rar` o `.7z`) que contenga, *en una sola carpeta* lo siguiente:

1. Un fichero llamado `AEDATA_code.py` (cuidado: respetad las mayúsculas, el Linux es importante) con las funciones que se piden en los apartado I-III. El fichero debe contener sólo las funciones: en el momento en que se hace un `import` no debe ejecutar nada. Las funciones deben devolver los valores que se piden *sin imprimir en ningún caso nada en la pantalla.*
2. Todos los ficheros `.py` auxiliares que hagáis creado para el correcto funcionamiento de esas funciones, es decir, todo los ficheros vuestros que se incluyen en el código principal con un `import`. Para comprobar que todo esté bien: cuando desde un fichero se haga `import AEDATA_code` todo se debe importar correctamente. Esto incluye los ficheros con las funciones para la medida del tiempo que habéis implementado para la primera práctica.
3. Si se ha implementado el opcional, un fichero llamado `AEDATA_optional.py` con todo el código implementado para la parte IV. Valen para este fichero las mismas normas que para el fichero `AEDATA_code.py`

4. Un fichero `AEDATA_time.pdf` con las gráficas, debudamente analizadas y comentadas, de las medidas de tiempo de ejecución que se hayan hecho. Si se ha implementado la parte `IV.A` opcional, esas gráficas, comentadas, también irán en este fichero.