Trabajo Práctico 2 - Algoritmos y Estructuras de Datos

Grupo 1: Martina Boero y Aníbal Córdoba

Ejercicio 1: sala de emergencias

Para la resolución de este problema se decidió utilizar la estructura de montículo binario de mínimos (útil para colas de prioridades), implementada en el archivo *monticulo.py.* De esta manera, se establece una dinámica que prioriza el estado de riesgo del paciente (siendo 1 el más crítico a 3, bajo), y en caso de pacientes con igual riesgo, anteponiendo a quien haya llegado antes. En esta implementación se utiliza de TAD básico una lista de Python representando el árbol de manera interna.

Cambios realizados en el código original:

- Se añade el atributo “lugar” a cada paciente para representar el orden de llegada
- Se implementan las comparaciones entre pacientes con __gt__ y __lt__ para comparar entre nivel de riesgo o, en caso de ser iguales, por orden de llegada o “lugar”.

Análisis de orden de complejidad (O-grande)

- **Insertar:** se agrega con el método append (complejidad O(1)) al final de la lista un elemento k. Luego, se infiltra hacia arriba hasta el lugar que le corresponda según su nivel de riesgo, comparándolo con su nodo padre iterativamente con el método InfiltArriba, que, en el peor caso, intercambiará A veces, siendo A la altura del árbol. En un árbol binario completo, O(A) = O(log (n)), siendo n la cantidad de nodos en el árbol. De esta manera, la complejidad total será de O(1) + O(log(n)) = O(log(n)), mejorando la eficiencia de inserción de una lista, que es de O(n).

- **EliminarMin:** en esta estructura de cola de prioridades, siempre buscaremos eliminar el elemento mínimo, ubicado en la raíz o posición 1. Para eso, en primer lugar, se lo reemplaza con el nodo ubicado en la última posición, en una operación realizada en tiempo constante. Extraer el último nodo (ya siendo intercambiados, el nodo ubicado al final es el de mayor prioridad) mediante la función pop() también se realiza en complejidad O(1). El paso más lento es el de infiltrar hacia abajo el nodo que se estableció en la raíz, siendo, en el peor caso, intercambiado A veces. Esta operación, en un árbol binario completo de n nodos tiene complejidad de O(log(n)). De la misma manera que en el método anterior, la complejidad total estará dada por O(log(n)).

Ejercicio 2: Temperaturas_DB

Para la resolución de este ejercicio se plantea un árbol AVL, con fechas como claves, ingresadas en formato aaaammdd y con carga útil o valores, temperaturas correspondientes a esas fechas.

- **__init__:** inicializa el registro como un árbol AVL vacío
- **guardar_temperatura:** mediante el método “agregar” del árbol AVL añade una nueva temperatura al registro. Como cada nodo se agrega como hoja y se intercambia recursivamente como máximo A veces, siendo A la altura del árbol. Esto genera, en el peor de los casos, un recorrido de profundidad logarítmica con base dos, ya que cada nivel tiene el doble de elementos que el anterior. Además, las rotaciones ejecutadas en el método agregar se realizan en tiempo constante. Por lo tanto, el orden de complejidad será de $\log_2(n)$.
- **devolver_temperatura:** para buscar una temperatura también se deberá recorrer en profundidad el árbol, utilizando el método obtener del árbol AVL. Este también tiene complejidad de $\log_2(n)$, pero sin realizar rotaciones.
- **borrar_temperatura:** de la misma manera que el guardar temperatura, este método debe recorrer el árbol buscando el elemento a borrar y reequilibrar el árbol tras realizar esta operación. De esta manera, el orden de complejidad de las deleciones es de $\log_2(n)$.
- **max_temp_rango:** este método busca la temperatura máxima dado un rango de fechas, estén incluidas en el registro o no. En la implementación que realizamos, se recorren todos los n nodos buscando aquellos que integren el rango, ya que no encontramos manera más eficiente de realizarlo. Los únicos nodos que no se recorren son aquellos mayores a la segunda fecha ingresada. En el peor de los casos, el método tiene un orden de complejidad O(n).
- **min_temp_rango:** efectuando el mismo análisis que en el método anterior, esta función tiene un orden de complejidad de O(n).
- **temp_extremos_rango:** en este caso, al darle uso a ambos métodos anteriores, recorrerá el árbol dos veces, en un orden de complejidad lineal de O(n) y un tamaño. Sin embargo, consideramos que se puede realizar de manera más eficiente, recorriendo una única vez el árbol, aunque no fue implementado.
- **mostrar_temperaturas:** de la misma manera que las funciones que evalúan un rango, en el peor caso se deben visitar los N nodos del árbol, con un orden O(n).
- **mostrar_cantidad_muestras:** este método accede al atributo longitud del árbol en tiempo constante O(1).

Ejercicio 3: servicio de transporte

Para la representación de este problema se implementaron las clases Vértice y Grafo a partir del código incluido en la bibliografía. Para resolver la consigna de definir y mostrar la ruta con menor costo y mayor cuello de botella se utilizaron algoritmos de Dijkstra y Dijkstra modificado respectivamente, implementando para esto las clases de ColaPrioridadMin y ColaPrioridadMax, basados en el montículo de mínimos que realizamos en el ejercicio 1.

En el archivo *rutas.py* se encuentran las funciones que se encargan de crear los grafos según se solicite desde el main y obtener consecutivamente la ruta con mayor cuello de botella o con menor costo.

En el main se aplicó una interfaz con el usuario vía consola, donde se le solicita ingresar el archivo donde se alojan las rutas, la ciudad de origen y el destino, y la opción que desee: 1 si quiere ver el mayor peso con el que se puede viajar hasta el destino o 2 si desea el menor costo monetario. El programa devolverá no solo el mayor cuello de botella/menor costo, también indicará la ruta que se debe tomar para cumplirlo.

Por cuestiones de tiempo no se ha implementado el algoritmo de Warshall para determinar las ciudades a las que se puede viajar, simplemente se realiza un control para indicar que la ruta es inexistente.