Facultad de UNER Ingeniería

ALGORITMOS Y ESTRUCTURAS DE DATOS

Trabajo Práctico 2

Integrantes del Grupo: Filchtinsky Tomas y Gabirondo Lucas

Objetivos

- Aplicar conceptos teóricos sobre estructuras jerárquicas, grafos y sus algoritmos asociados.

Consignas

1. Sala de Emergencias

El **triaje** es un proceso que permite una gestión del riesgo clínico para poder manejar adecuadamente y con seguridad los flujos de pacientes cuando la demanda y las necesidades clínicas superan a los recursos. El siguiente proyecto de software posee una sencilla simulación de esta situación: enlace a código.
En este proyecto para simplificar existen pacientes con tres niveles de riesgo para su salud: 1: crítico, 2: moderado, 3: bajo. Actualmente la simulación muestra que los pacientes se atienden según el orden en que llegan al centro de salud.

Se solicita:

a. Seleccionar, programar y aplicar una estructura de datos adecuada para almacenar los pacientes conforme ingresan al centro de salud **de modo tal que cuando se atiende un paciente siempre sea aquel cuyo nivel de riesgo es el más delicado** en comparación con el resto de los pacientes que restan por ser atendidos. Si dos pacientes poseen el mismo nivel de riesgo, adoptar un segundo criterio para seleccionar uno de ellos. Recordar que la estructura de datos debe ser genérica, es decir, debe poder almacenar cualquier tipo de dato, y no ser específica para alojar pacientes (separar implementación de aplicación).
b. Fundamentar en el informe, en no más de una página, la estructura seleccionada indicando el orden de complejidad *O* de inserciones y de eliminaciones en la estructura seleccionada.

Observación: Pueden realizarse todas las modificaciones al código que crean necesarias siempre que se respete el propósito del problema planteado.

RESPUESTAS:

En esta implementación, se ha seleccionado una cola de prioridad que utiliza un montículo binario (también conocido como heap) como estructura subyacente. Un montículo binario es una estructura de datos eficiente para administrar elementos con prioridad, donde se busca acceso rápido al elemento de mayor o menor prioridad (dependiendo del tipo de montículo).
La elección del montículo binario permite que la cola de prioridad tenga un acceso óptimo al elemento de mayor o menor prioridad en la parte superior, con tiempos de ejecución eficientes tanto para inserciones como para eliminaciones.

Inserciones (insertar)

Al insertar un nuevo elemento, este se agrega al final del montículo y luego se ajusta la estructura mediante el proceso de "infiltración hacia arriba" (infiltArriba). Este proceso asegura que se mantenga la propiedad de montículo: el padre siempre es menor que sus hijos en un min-montículo.
Complejidad: el ajuste tiene una complejidad de O(log n), ya que en el peor de los casos el elemento debe moverse desde el último nivel hasta la raíz, recorriendo la altura del montículo, que es O(log n).

Eliminaciones (extraer_mayor_prioridad)

La eliminación del elemento de mayor prioridad se realiza extrayendo la raíz (primer elemento) del montículo y reemplazándola con el último elemento. Después, se aplica el proceso de "infiltración hacia abajo" (infiltAbajo) para restaurar la propiedad del montículo.
Complejidad: similar a la inserción, el proceso de infiltración tiene una complejidad de O(log n), ya que en el peor caso el elemento debe desplazarse desde la raíz hasta una hoja.

2. Temperaturas_DB

Kevin Kelvin es un científico que estudia el clima y, como parte de su investigación, debe consultar frecuentemente en una base de datos la temperatura del planeta tierra dentro de un rango de fechas. A su vez, el conjunto de medidas crece conforme el científico registra y agrega **mediciones** a la base de datos.
Una medición está conformada por el valor de temperatura en ºC (flotante) y la fecha de registro, la cual deberá ser ingresada como “dd/mm/aaaa” (string). (Internamente sugerimos emplear objetos de tipo *datetime*).
Ayude a Kevin a realizar sus consultas eficientemente implementando una base de datos en memoria principal “**Temperaturas_DB**” que utilice internamente un árbol AVL. La base de datos debe permitir realizar las siguientes operaciones (interfaz de Temperaturas_DB):

guardar_temperatura(temperatura, fecha): guarda la medida de temperatura asociada a la fecha.

devolver_temperatura(fecha): devuelve la medida de temperatura en la fecha determinada.

max_temp_rango(fecha1, fecha2): devuelve la temperatura máxima entre los rangos fecha1 y fecha2 inclusive (fecha1 < fecha2). Esto no implica que los intervalos del rango deban ser fechas incluidas previamente en el árbol.

min_temp_rango(fecha1, fecha2): devuelve la temperatura mínima entre los rangos fecha1 y fecha2 inclusive (fecha1 < fecha2). Esto no implica que los intervalos del rango deban ser fechas incluidas previamente en el árbol.

temp_extremos_rango(fecha1, fecha2): devuelve la temperatura mínima y máxima entre los rangos fecha1 y fecha2 inclusive (fecha1 < fecha2).

borrar_temperatura(fecha): recibe una fecha y elimina del árbol la medición correspondiente a esa fecha.

devolver_temperaturas(fecha1, fecha2): devuelve un listado de las mediciones de temperatura en el rango recibido por parámetro con el formato “dd/mm/aaaa: temperatura ºC”, ordenado por fechas.

cantidad_muestras(): devuelve la cantidad de muestras de la BD.

Adicionalmente realizar las siguientes actividades:

- Escriba una tabla con el análisis del orden de complejidad Big-O para cada uno de los métodos implementados para su clase “Temperaturas_DB”, explique brevemente el análisis de los mismos.

RESPUESTAS:

guardar_temperatura	O(log n)	Este método llama a agregar en el árbol AVL, que tiene complejidad O(log n) debido al reequilibrado tras la inserción, manteniendo el árbol balanceado.
devolver_temperatura	O(log n)	obtener en el árbol AVL también tiene O(log n) al buscar un nodo por su clave. Al estar balanceada la estructura es posible que sea log la búsqueda.
max_temp_rango	O(n)	Itera en orden todos los nodos del árbol, y en el peor caso, debe verificar cada nodo en el rango dado para encontrar la temperatura máxima.
min_temp_rango	O(n)	Similar a max_temp_rango, recorre todos los nodos en el rango para encontrar la mínima temperatura, y podría visitar hasta todos los nodos en el peor caso.
temp_extremos_rango	O(n)	Combina las funciones de max_temp_rango y min_temp_rango, por lo que recorre los nodos en el rango para obtener ambas temperaturas extremas.
borrar_temperatura	O(log n)	Utiliza el método eliminar del árbol AVL, que tiene complejidad O(log n) debido al reequilibrio necesario para mantener la estructura balanceada.

devolver_temperaturas	O(n)	Recorre los nodos en orden dentro del rango dado y, en el peor caso, podría visitar todos los nodos, resultando en O(n).
cantidad_muestras	O(1)	Al devolver el tamaño actual almacenado en tamano(tamaño), que es una operación de acceso directo es constante.

3. Palomas mensajeras

Desde la antigüedad y hasta la invención del telégrafo en el año 1835 por Morse, las palomas mensajeras fueron el sistema de comunicación más rápido entre ciudades. La premisa que sustenta este tipo de comunicación es que una paloma mensajera criada en un lugar y liberada en otro, volverá a su lugar de origen; si se aprovecha este viaje haciendo que el animal lleve consigo un mensaje, la información se podrá replicar y transmitir de un lugar a otro.

William, como dueño de la agencia de noticias “Palomas William”, conoce el tema y entiende que hay varios caminos para entregar un mensaje a un lugar. Sin embargo, todavía no está seguro de cómo encontrar la mejor forma de enviar las noticias a las demás aldeas ahorrando recursos. Él tiene a cargo toda la comunicación desde su aldea “Peligros” a un grupo de otras 21 aldeas, todas con un palomar propio. Cada aldea, incluída Peligros, sólo puede enviar mensajes a sus aldeas vecinas de las cuales tiene alguna paloma mensajera. La lista de aldeas vecinas y la distancia entre cada par de ellas está disponible en el archivo aldeas.txt. Cada ruta en la lista es una tupla (S_i, T_i, d_i) donde S_i y T_i son, respectivamente, los nombres (string) de la aldea de inicio y final de cada posible viaje de una paloma mensajera, y d_i representa la distancia en leguas entre esas aldeas (entero positivo).

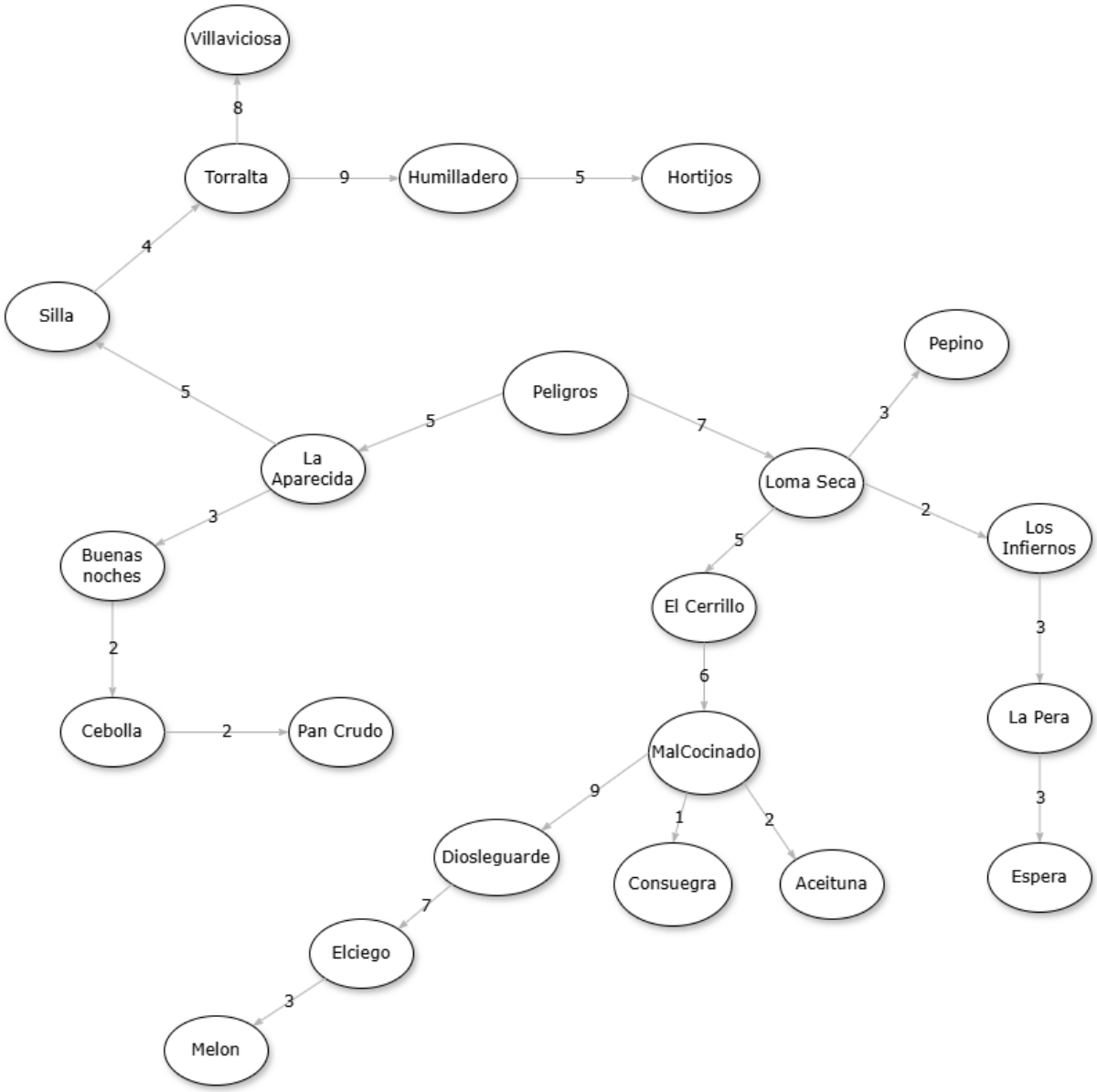
Ayude a William a encontrar la forma más eficiente de llevar un mensaje desde su aldea a todas las demás, donde cada aldea reciba solamente una vez la noticia. Cada aldea, al recibir la noticia, puede replicarla y enviarla a tantas aldeas vecinas como quiera.

Entregables:

- Mostrar la lista de aldeas en orden alfabético.
- Para cada aldea, mostrar de qué vecina debería recibir la noticia, y a qué vecinas debería enviar réplicas, siendo que se está enviando el mensaje de la forma más eficiente a las 21 aldeas. Tomar en cuenta que desde Peligros solamente se envían noticias a una o más aldeas vecinas.
- Para el envío de una noticia, mostrar la suma de todas las distancias recorridas por todas las palomas enviadas desde cada palomar.

RESPUESTAS:

Gráfico del grafo:



Explicación de Implementación:

Para que pudiéramos cumplir con las consignas impartidas para el tercer proyecto, tuvimos que implementar:

- Una clase Grafo
- Una clase Vértice
- Una clase MonticuloBinario (la cual le da cuerpo a la cola de prioridad)
- Una función Prim (la cual pone en función al Algoritmo de Prim)
- Una aplicación principal (Main) para resolver las consignas

El principal y más importante es el algoritmo de Prim el cual refactorizamos para que funcionara con nuestro MonticuloBinario implementando los siguientes cambios:

```
You, 11 hours ago | 2 authors (tomasfilchy and one other)
from modules.min_heap import monticuloBinario
import sys

def prim(G, inicio):
    cp = monticuloBinario()  # Se crea una cola de prioridad
    arbol_expansion = set()  # Para mantener registro de las aristas del árbol
    flujo_mensajes = []  # Lista para almacenar el flujo de mensajes
    visitados = set()  # Conjunto para evitar duplicados
    aldeas_recorridas = set()  # Conjunto para almacenar aldeas recorridas

    # Inicializar todos los vértices en inf y None asignandoles distancia 0 al vertice de inicio
    for v in G:
        v.asignar_distancia(sys.maxsize)
        v.asignar_predecesor(None)
    inicio.asignar_distancia(0)
    cp.construir_monticulo([(v.obtener_distancia(), v) for v in G]) # Construimos el monticulo

    while not cp.esta_vacia():
        verticeActual = cp.eliminarMin()

        # Solo procesar si no ha sido visitado
        if verticeActual.obtener_id() not in visitados:
            visitados.add(verticeActual.obtener_id())
            aldeas_recorridas.add(verticeActual.obtener_id())

            # Si tiene predecesor, registrar el flujo de mensajes
            if verticeActual.obtener_predecesor():
                flujo_mensajes.append({
                    'emisor': verticeActual.obtener_predecesor().obtener_id(),
                    'receptor': verticeActual.obtener_id(),
                    'distancia': verticeActual.obtener_ponderacion(verticeActual.obtener_predecesor())
                })

        # Si el vértice tiene predecesor, agregar la arista al árbol
        if verticeActual.obtener_predecesor():
            arbol_expansion.add((verticeActual.obtener_predecesor(), verticeActual))

        for verticeSiguiente in verticeActual.obtener_conexiones():
            # Solo procesar vértices no visitados
            if verticeSiguiente not in visitados:
                ponderacion = verticeActual.obtener_ponderacion(verticeSiguiente)
                if verticeSiguiente in cp and ponderacion < verticeSiguiente.obtener_distancia():
                    verticeSiguiente.asignar_predecesor(verticeActual)
                    verticeSiguiente.asignar_distancia(ponderacion)
                    cp.decrementar_clave(verticeSiguiente, ponderacion)

    return  flujo_mensajes, aldeas_recorridas      tomasfilchy, 13 hours ago • ultimos detalles
```

Este código implementa el algoritmo de Prim para encontrar el árbol de expansión mínima en un grafo, específicamente diseñado para un sistema de comunicación entre aldeas. Comienza creando varias estructuras de datos (una cola de prioridad, conjuntos para el árbol de expansión y vértices visitados) y asigna distancias iniciales infinitas a todos los vértices excepto el inicial. Luego, en un bucle principal, va seleccionando el vértice con menor distancia de la cola de prioridad y, si no ha sido visitado, lo procesa: lo marca como visitado, registra el flujo de mensajes si tiene un predecesor, añade la arista al árbol de expansión, y revisa todos sus vértices adyacentes no visitados para actualizar sus distancias si encuentra un camino más corto. Todo este proceso continúa hasta que la cola de prioridad está vacía, momento en el que retorna el flujo de mensajes y las aldeas recorridas, efectivamente encontrando la forma más eficiente de conectar todas las aldeas en la red.