Tecnicatura en Procesamiento y Explotación
de Datos Algoritmos y Estructura de Datos

Actividad: Trabajo práctico integrador 2

Título: Informe de análisis Trabajo practico numero 2

Docentes:
Dr. Javier Eduardo
Diaz Zamboni
Bioing. Jordán F. Insfrán
Bioing. Diana Vertiz del Valle
Bruno M. Breggia

Alumno:
Joaquin Frattin

Problema 1

En la gestión de la cola de prioridades en esta simulación, se ha elegido utilizar un Montículo Binario como estructura de datos principal. El Montículo Binario es eficiente, con una complejidad de orden O(log(n)) tanto para las inserciones como para las eliminaciones. Aunque se asemeja conceptualmente a un árbol, se implementa internamente como una lista.

En dicha simulación se genera un paciente al azar el cual será agregado al elemento sala de espera que es una instancia de la clase montículo. El objeto sala de espera irá incorporando nuevos pacientes y mostrando en pantalla los pacientes en cola, además sala de espera tiene la facultad de atender un paciente es decir utilizamos la capacidad de montículo binario para eliminar el objeto de mayor criticidad con la función eliminar min de la clase montículo.

La clase Montículo binario si intento modelar lo más escalable posible es decir que pueda utilizarse para cualquier clase de datos, en este caso se utilizó la clase paciente, la cual se le asignaron los métodos que se consideraron necesarios para que pueda funcionar en los métodos de la clase montículo.

Problema 2

Se realizó la implementación del problema dos utilizando como guía la bibliografía provista por la cátedra. Para eso se implementaron 4 clases, clase nodoArbol , la clase Arbol Binario suqueda, la clase Arbol Avl que es una clase hija de arbolbinariobusqueda, estas tres clases son las que necesitamos para poder construir los algoritmos de la clase temperaturas que serán los utilizados por Kevin Kelvin para su trabajo.

guardar_temperatura(temperatura, fecha):	O(log n)	Posee una complejidad promedio de O(log n)
devolver_temperatura(fecha)	O(log n)	Posee una complejidad promedio de O(log n)
max_temp_rango(fecha1, fecha2):	O(n)	Posee una complejidad de O(log n
temp_extremos_rango(fecha 1, fecha2):	O(log n)	Posee una complejidad O (log n) ya que es una busqueda en un arbol Avl
min_temp_rango(fecha1, fecha2):	O(log n	Posee una complejidad O (log n) ya que es una busqueda en un

		arbol Avl
borrar_temperatura(fecha):	O(log n)	Necesito encontrar el nodo correspondiente a esa fecha recorriendo el árbol luego borrarlo, con una complejidad O (log N) la misma que al insertar.
devolver_temperaturas(fecha 1, fecha2):	O(log n)	Al ser un arbol AVL la complejidad que posee es de O(log n)
cantidad_muestras()	O(1)	Devuelve la cantidad de muestras de la base de datos posee una complejidad O(1) ya que consulta un atributo que contiene esta información.

Problema 3

En este problema se busca resolver el problema que tiene la empresa CasaBella el cual consiste en calcular el máximo peso transportable desde una ciudad a otra, y además calcular el mínimo precio.

Para eso se opta por elegir una estructura de datos del tipo grafo el cual posee como Nodos las ciudades. Y se implementó un algoritmo del tipo para realizar los cálculos necesarios y así poder obtener el peso máximo y calcular el mínimo precio.
Para eso se modelaron las siguientes clases :

Clase Grafo:

Esta clase Grafo representa un grafo dirigido ponderado, donde los vértices son instancias de la clase Vertice y las aristas tienen pesos asociados. A continuación, se las principales características y métodos de la clase:

- listaVertices: Un diccionario que almacena los vértices del grafo. Las claves son las claves de los vértices y los valores son instancias de la clase Vertice.

- numVertices: Un contador que registra la cantidad de vértices en el grafo.

Métodos:

- agregarVertice(clave): Agrega un nuevo vértice al grafo con la clave proporcionada y devuelve la instancia del nuevo vértice.
- obtenerVertice(n): Devuelve la instancia del vértice correspondiente a la clave n si existe en el grafo; de lo contrario, devuelve None.
- __contains__(n): Verifica si la clave n está en el grafo.
- agregarArista(de, a, costo, precio): Agrega una arista al grafo entre los vértices con claves de y a, con pesos opcionales costo y precio.
- obtenerVertices(): Devuelve todas las claves de los vértices en el grafo.
- __iter__(): Permite iterar sobre los objetos de vértice en el grafo.
- maximo_peso_transportado(ciudad_inicio, ciudad_destino): Calcula el máximo peso transportado desde la ciudad_inicio hasta la ciudad_destino utilizando un algoritmo de clase Dijkstra el cual toma como valor de ponderación el peso del producto y calcula el maximo de los mínimos valores.
-
- minimo_precio_envio(start_city, end_city): Calcula el mínimo precio de envío desde start_city hasta end_city utilizando un montículo de prioridad y utilizando un algoritmo de clase Dijkstra el cual toma como valor de ponderación el precio del producto y calcula el mínimo precio.

Clase Vertice:

Esta clase se utiliza para representar los nodos en un grafo ponderado y proporciona métodos para manipular las conexiones entre vértices, así como para obtener información sobre las distancias, predecesores, conexiones y precios asociados a los vértices vecinos.

Atributos de la clase:

- id: Identificador único del vértice.
- conectadoA: Un diccionario que almacena los vértices vecinos como claves y una tupla de ponderación y precio como valores. La ponderación representa el peso máximo y el precio el costo asociado a la arista que conecta el vértice actual con el vértice vecino.
- distancia: Distancia desde un vértice origen hasta este vértice.
- predecesor: Vértice predecesor en el camino más corto desde un vértice origen hasta este vértice.
- precio: Precio asociado al vértice, inicializado como infinito.

Métodos de la clase:

- agregarVecino(vecino, ponderacion, precio): Agrega un vértice vecino al vértice actual con una ponderación y precio dados.

- asignarDistancia(distancia): Asigna la distancia desde un vértice origen hasta este vértice.
- obtenerDistancia(): Obtiene la distancia desde un vértice origen hasta este vértice.
- asignarPredecesor(predecesor): Asigna el vértice predecesor en el camino más corto desde un vértice origen hasta este vértice.
- obtenerPredecesor(): Obtiene el vértice predecesor en el camino más corto desde un vértice origen hasta este vértice.
- __str__(): Devuelve la representación en cadena del identificador del vértice.
- obtenerConexiones(): Devuelve todos los vértices vecinos del vértice actual.
- obtenerPonderacion(vecino): Devuelve la ponderación al vértice vecino.

Además se implementó una clase Montículo la cual utiliza la biblioteca heapq la cual ayuda a modelar un montículo binario necesario para la implementación de la clase grafo. Y se implementó una función la cual carga los datos del archivo txt y devuelve un grafo listo para su utilización.