UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos 2006 **Práctica 9**

Grafos

Objetivos
- Representar grafos, implementar las operaciones de la abstracción y describir soluciones utilizándolos

Ejercicio 1.
Indicar para cada uno de los siguientes casos si la relación se representa a través de un grafo no dirigido, de un digrafo o es indistinto para cualquiera de las dos estructuras.
a) Vértices: países. Aristas: es limítrofe.
b) Vértices: países. Aristas: principales mercados de exportación.
c) Vértices: dispositivos en una red de computadoras. Aristas: conectividad.
d) Vértices: variables de un programa. Aristas: relación "usa". (Decimos que la variable ***x*** usa la variable ***y***, si ***y*** aparece del lado derecho de una expresión y ***x*** aparece del lado izquierdo, por ejemplo **x = y**).

Ejercicio 2.
a) Para un grafo no dirigido de *n* vértices, ¿cuál es el mayor número de aristas que puede tener? **Fundamentar**.
b) Si en cambio el grafo es dirigido, y no tiene aristas que vayan de un nodo a sí mismo, ¿cuál es el mayor número de aristas que puede tener? **Fundamentar**.

Ejercicio 3.
¿en función de qué parámetros resulta apropiado realizar la estimación del orden de ejecución para algoritmos sobre grafos densos? ¿Y para algoritmos sobre grafos dispersos? Fundamentar las respuestas.

Ejercicio 4.
Sea la siguiente especificación:

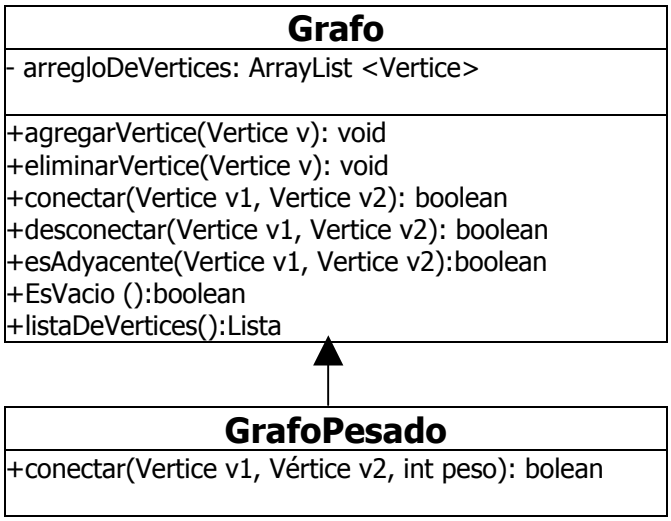
Grafo
- arregloDeVertices: ArrayList <Vertice>
+agregarVertice(Vertice v): void +eliminarVertice(Vertice v): void +conectar(Vertice v1, Vertice v2): boolean +desconectar(Vertice v1, Vertice v2): boolean +esAdyacente(Vertice v1, Vertice v2):boolean +EsVacio ():boolean +listaDeVertices():Lista

La descripción de algunos de los métodos es la siguiente:
conectar(Vertice v1, Vertice v2): boolean //Si v1 y v2 son vértices del grafo, entonces agrega la arista y retorna true, si no retorna false.
desconectar(Vertice v1, Vertice v2): boolean //Si (v1, v2) no es arista del grafo, entonces no modifica nada y retorna false; si no elimina del grafo la arista y retorna true.
listaDeVertices():Lista //Retorna la lista de vértices del grafo.

Vertice
- nombre: String // Nombre del vértice - listaDeAristas: Lista // Lista de aristas
+ Vertice (String nombre) + setNombre (String nombre): void + getNombre (Vertice v): String + getListaDeAristas():Lista + setListaDeAristas(Lista lista): void

Arista
- destino: Vertice - costo: int
+ Arista (Vertice destino, int costo) + setDestino(Vertice vertice): void + getDestino(): Vertice + setCosto(int costo): void + getCosto(): int

a) Implementar la clase Grafo en el paquete **estructutrasdedatos.grafos** de acuerdo a la especificación dada.
b) Estimar el orden de ejecución de cada una de las operaciones.
c) Definir como subclase de Grafo, la clase **GrafoPesado** en el paquete **estructutrasdedatos.grafos**, de acuerdo a la especificación dada a continuación:



Ejercicio 5.
Agregue a la clase Grafo los siguientes métodos
a) **dfs(): Lista** // devuelve una lista con el recorrido en profundidad
b) **bfs(): Lista** // devuelve una lista con el recorrido en amplitud
c) **tieneCiclo():boolean**
d.1) Mostrar mediante un ejemplo que el algoritmo de Dijkstra falla para cuando existen en el grafo aristas de costo negativo.
d.2) **dijkstra (Vértice v): ArrayList<Costo>** //se ejecuta usando una Heap. Siendo Costo un objeto que contiene un Vertice, el costo de accederlo y el Vertice por el cual hay que pasar previamente
e) **ordenTopologico(): Lista** //La organización topológica (o "sort topológico") de un DAG (grafo acíclico dirigido) es un proceso de asignación de un orden lineal a los vértices del DAG de modo que si existe una arista (v,w) en el DAG, entonces v aparece antes de w en dicho ordenamiento lineal
f) **Estimar** los órdenes de ejecución de los métodos anteriores.

Ejercicio 6.
Implemente una clase llamada **Mapa** que mantiene la información de ciudades y de las rutas que las conectan. La estructura usada para representar esta información es un Grafo. La clase pertenece al paquete **estructurasdedatos.aplicacion.tp9**

La interface pública de la clase Mapa es la siguiente:

a) existeCamino (Ciudad c1, Ciudad c2): boolean
b) devolverCamino (Ciudad c1, Ciudad c2): Lista

c) devolverCaminoExceptuando (Ciudad c1, Ciudad c2, Lista ciudades): Lista // devuelve la lista de ciudades que forman un camino desde c1 a c2, sin pasar por las ciudades que están contenidas en la lista ciudades.
d) caminoMasCorto(Ciudad c1, Ciudad c2): Lista //Como las rutas poseen un costo de peaje para transitarlas, debe encontrar el camino mas económico para llegar de una ciudad a otra. Es decir, no importa los kilómetros recorridos, solo debe preocuparse para minimizar el dinero que necesita para pagar los peajes.
e) caminoMasCortoCargandoCombustible (Ciudad c1, Ciudad c2, Auto a1): Lista //Suponiendo que se desplaza con automóvil que posee un tanque de combustible de tamaño conocido y un consumo (litros/longitud) también conocido, y las rutas tienen una longitud, el auto no se debe quedar sin combustible en medio de una ruta. El automóvil puede completar su tanque al llegar a cualquier ciudad. Debe devolver el camino más corto en función del consumo de combustible