UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos 2007 **Trabajo Práctico Nro. 7**

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 o de un digrafo.
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**

Ejercicio 4.
Sea la siguiente especificación:

Grafo
- Vertice[] vertices - int cantVertices
+Grafo() +agregarVertice(String dato) +eliminarVertice(String dato) +conectar(String dato1, String dato2): boolean +desconectar(String dato1, String dato2): boolean +esAdyacente(String dato1, String dato2): boolean +esVacio ():boolean +listaDeVertices():Lista +listaDeAdyacentes(String dato): Lista - posición (String dato): int

El constructor **Grafo()** inicializa un grafo vacío, es decir con el véctor de vértices vacío.
El método **agregarVertice(String dato)** crea un vértice con label dato y lo agrega al arreglo de vértices.
El método **eliminar (String dato)** elimina el vértice con label dato del arreglo de vértices.
El método **conectar(String dato1, String dato2): boolean** agrega la arista al grafo. Si alguno de los vértices con label dato1 y dato2 no pertenece al grafo, no los conecta. Devuelve true o false según si se pudo o no realizar la operación.
El método **desconectar(String dato1, String dato2): boolean** elimina la arista del grafo. También devuelve si pudo o no realizar la operación.
El método **esAdyacente(String dato1, String dato2): boolean** determina si los vértices con dato1 y dato2 están conectados o no.
El método **esVacio(): boolean** determina si el grafo tiene vértices o no.
El método **listaDeVertices():Lista** retorna la lista con los datos de todos los vértices del grafo.

El método **listaDeAdyacentes(String dato):Lista** retorna la lista con los datos de los adyacentes del dato pasado como parámetro.
El método posición (String dato): int es un método privado busca en que posición del arreglo se encuentra el dato.

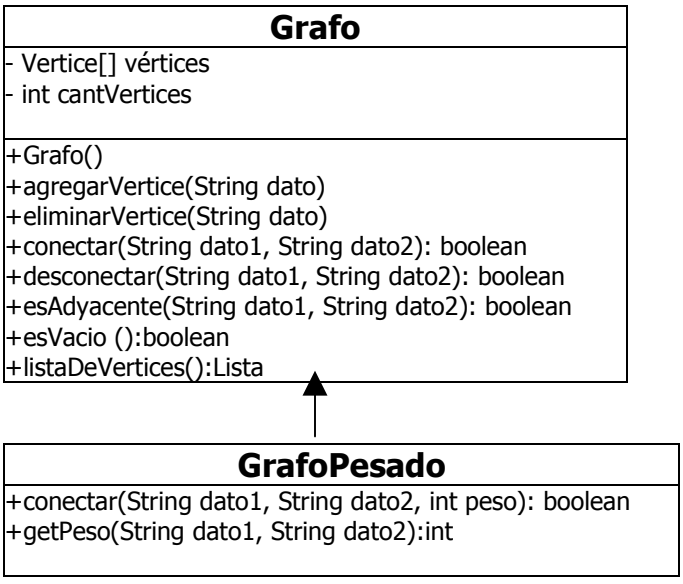
Vértice
- String dato - Lista adyacentes - int posicion
Vertice (String dato) setDato (String dato) getDato(): String setAdyacentes(Lista aristas) getAdyacentes(): Lista setPosicion(int pos) getPosicion(): int conectar(Vertice vDest) conectar(Vertice vDest, int peso)

Arista
- Vertice destino
Arista (Vertice dest) setDestino(Vertice vertice) getDestino(): Vertice

Clase Vértice
La variable de instancia adyacentes es una Lista de objetos de clase Arista.
El método conectar(Vértice vDest) crea una arista con vDest y la agrega a la lista de adyacentes del vértice (Se usa sólo para grafos no pesados).
El método conectar(Vértice vDest, int peso) crea una arista con vDest y peso y la agrega a la lista de adyacentes del vértice. (Se usa sólo para grafos pesados)

a) Implemente las clases Grafo, Vértice y Arista.
b) Estime el orden de ejecución de cada una de las operaciones.
c) Como hace para que los vertices admitan Objects? ¿Que consideraciones debe tener en cuenta al momento de implementar esta modificacion?

Ejercicio 5.
Sea la siguiente especificación:



a) Tenga presente que debe crear la clase AristaPesada (como subclase de Arista) que contenga el peso.

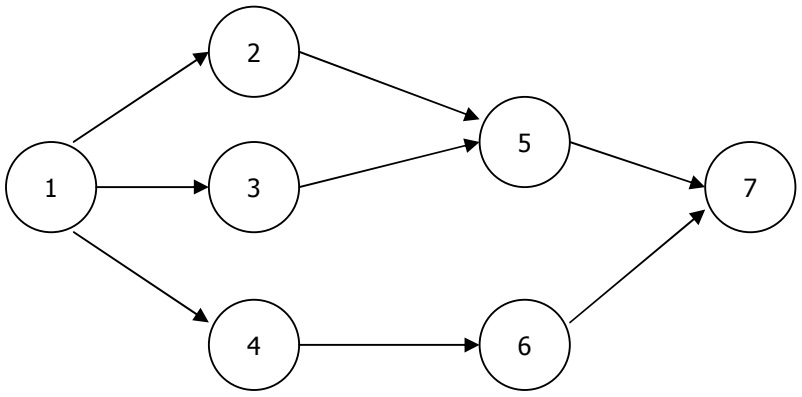
b) Definir la clase **GrafoPesado** como subclase de **Grafo** e implementarlo. Tenga presente que el mensaje getPeso (String dato1, String dato2): int devuelve el peso de la conexión entre ambos datos. Si no existiera la conexión devuelve -1.
c) Analice el comportamiento del método **conectar(String dato1, String dato2): boolean**, que hereda de la clase Grafo.

Ejercicio 6.
a) Agregue a la clase Grafo los siguientes métodos:
dfs(): Lista. Devuelve una lista con el recorrido en profundidad.
bfs(): Lista. Devuelve una lista con el recorrido en amplitud.
tieneCiclo():boolean.

b) **Estimar** los órdenes de ejecución de los métodos anteriores.

Ejercicio 7.
a) Calcule el tiempo de ejecución para el algoritmo de **dijkstra** sin utilizar Heap.
b) Implemente el método **dijkstra (Vértice v): Lista** que se ejecuta usando una Heap y devuelve una Lista de Costos. Siendo Costo un objeto que contiene un Vertice, el costo de accederlo y el Vertice por el cual hay que pasar previamente
c) Calcule el tiempo de ejecución para el algoritmo de b).
d) Mostrar mediante un ejemplo que el algoritmo de Dijkstra falla para cuando existen en el grafo aristas de costo negativo.

Ejercicio 8.
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.
a) Para el siguiente DAG muestre un orden topológico:



b) implemente el método **ordenTopologico(): Lista.**

Ejercicio 9.
Sea la siguiente especificación:

Mapa
- Grafo mapaCiudades
+ Mapa() + existeCamino (String ciudad1, String ciudad2): boolean + devolverCamino (String ciudad1, String ciudad2): Lista + devolverCaminoExceptuando (String ciudad1, String ciudad2, Lista ciudades): Lista + caminoMasCorto(String ciudad1, String ciudad2): Lista + caminoMasCortoCargandoCombustible (String ciudad1, String ciudad2): Lista

Implemente las clases **Mapa**

El método **existeCamino (String ciudad1, String ciudad2): boolean** determina si se puede llegar de ciudad1 a ciudad2.
El método **devolverCamino (String ciudad1, String ciudad2): Lista** retorna la lista de ciudades que se deben atravesar para ir de ciudad1 a ciudad2.
El método **devolverCaminoExceptuando (String ciudad1, String ciudad2, Lista ciudades): Lista** devuelve la lista de ciudades que forman un camino desde ciudad1 a ciudad2, sin pasar por las ciudades que están contenidas en la lista ciudades pasada por parámetro.
El método **caminoMasCorto(String ciudad1, String ciudad2): Lista** devuelve la lista de ciudades que forman el camino mas corto para llegar de una ciudad a otra (las rutas poseen la longitud).
El método **caminoMasCortoCargandoCombustible (String ciudad1, String ciudad2): Lista** debe devolver la lista de ciudades que forman el camino más corto en función del consumo de combustible, suponiendo que se desplaza con automóvil que posee un tanque de combustible de tamaño conocido y un consumo (litros/distancia) también conocido por el algoritmo. El auto no se debe quedar sin combustible en medio de una ruta. El automóvil puede completar su tanque al llegar a cualquier ciudad.