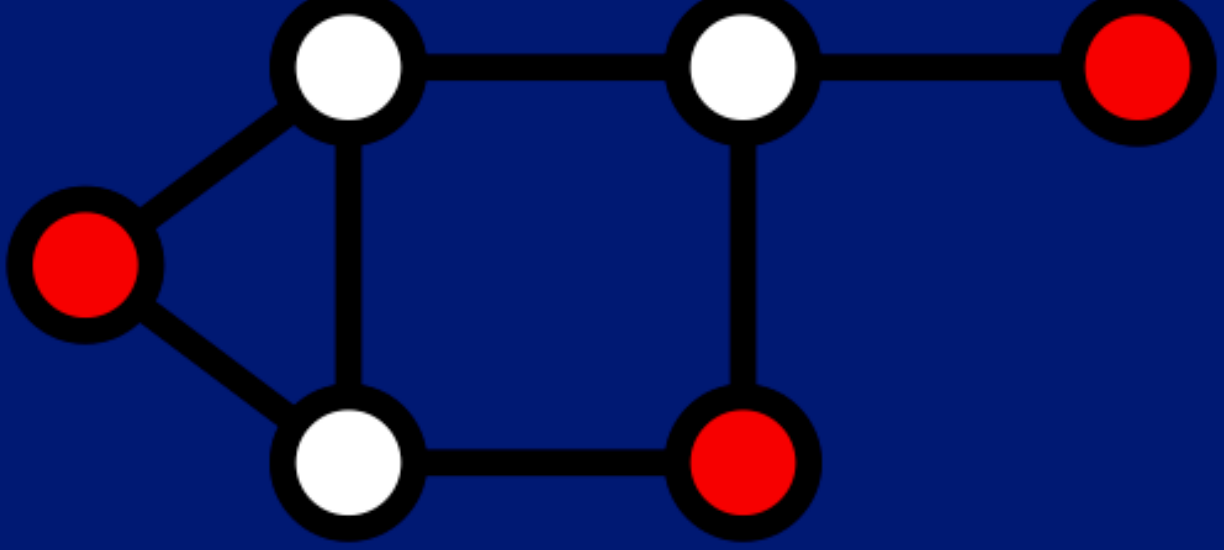
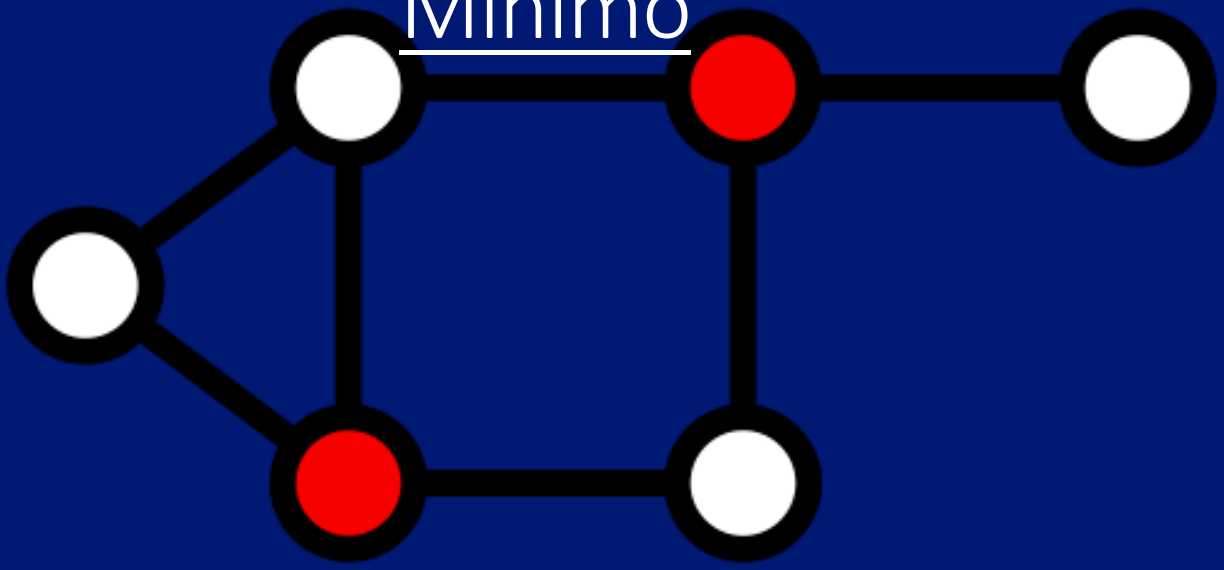
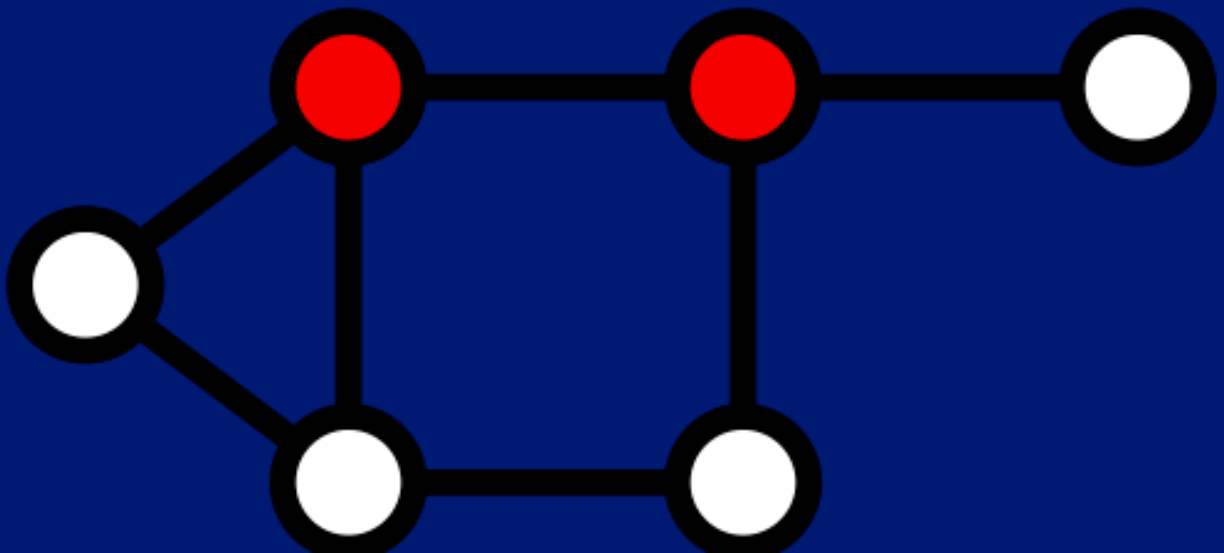


Trabajo Práctico 3: Conjunto Dominante Mínimo



Profesores: Patricia Bagnes – Ignacio Sotelo – Gabriel Carrillo

Integrantes: Lucas Aguilar
Federico Pardo
Lautaro Roca Vilte



Introducción:

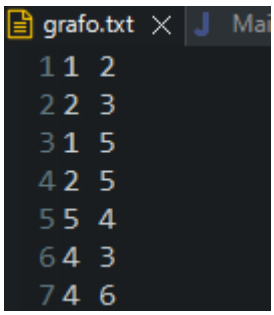
El problema del conjunto dominante mínimo en teoría de grafos se refiere a encontrar un subconjunto de vértices en un grafo, de manera que cada vértice del grafo esté en el conjunto o sea adyacente (vecino) a al menos un vértice en el conjunto. En otras palabras, se busca el conjunto más pequeño de vértices que cubra o domine todos los vértices del grafo. Este conjunto se denomina "conjunto dominante mínimo" y se considera mínimo cuando ningún subconjunto propio de él sigue siendo un conjunto dominante.

El problema del conjunto dominante mínimo pertenece a la clase de problemas NP en la teoría de complejidad computacional. La resolución de este problema implica encontrar una solución que minimice la cantidad de vértices en el conjunto dominante, lo que a menudo requiere enfoques algorítmicos eficientes y heurísticas para grafos de gran tamaño. La esencia del problema siempre será consistente: encontrar el conjunto más pequeño de vértices que garantice la dominancia de todos los vértices en un grafo dado.

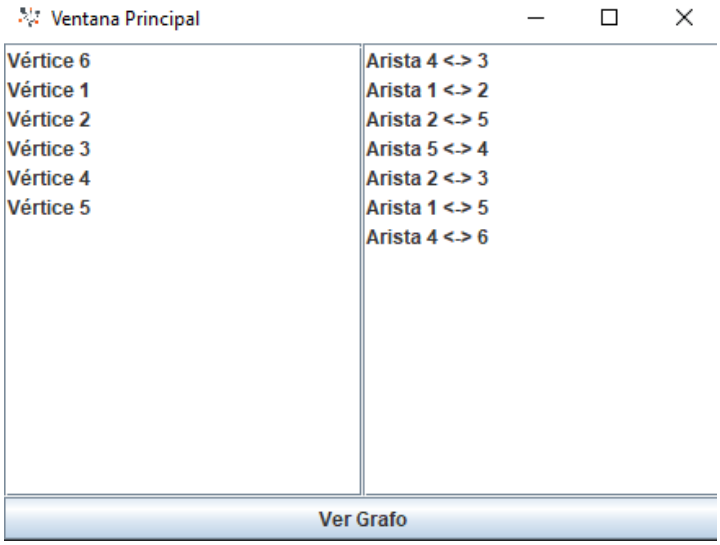
Como en los trabajos anteriores, seguimos implementando las técnicas de buena programación que fueron inculcadas en clase y las correcciones que fueron puestas previamente.

Descripción de la aplicación:

Antes de empezar, para ingresar el grafo, es necesario modificar el archivo de texto “grafo.txt” de la siguiente manera:

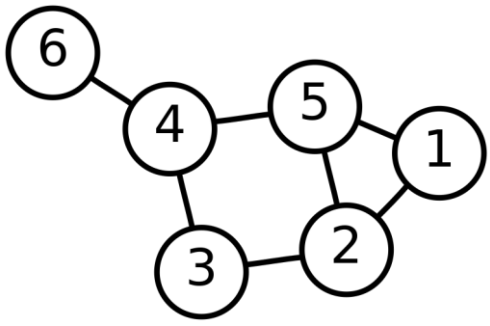


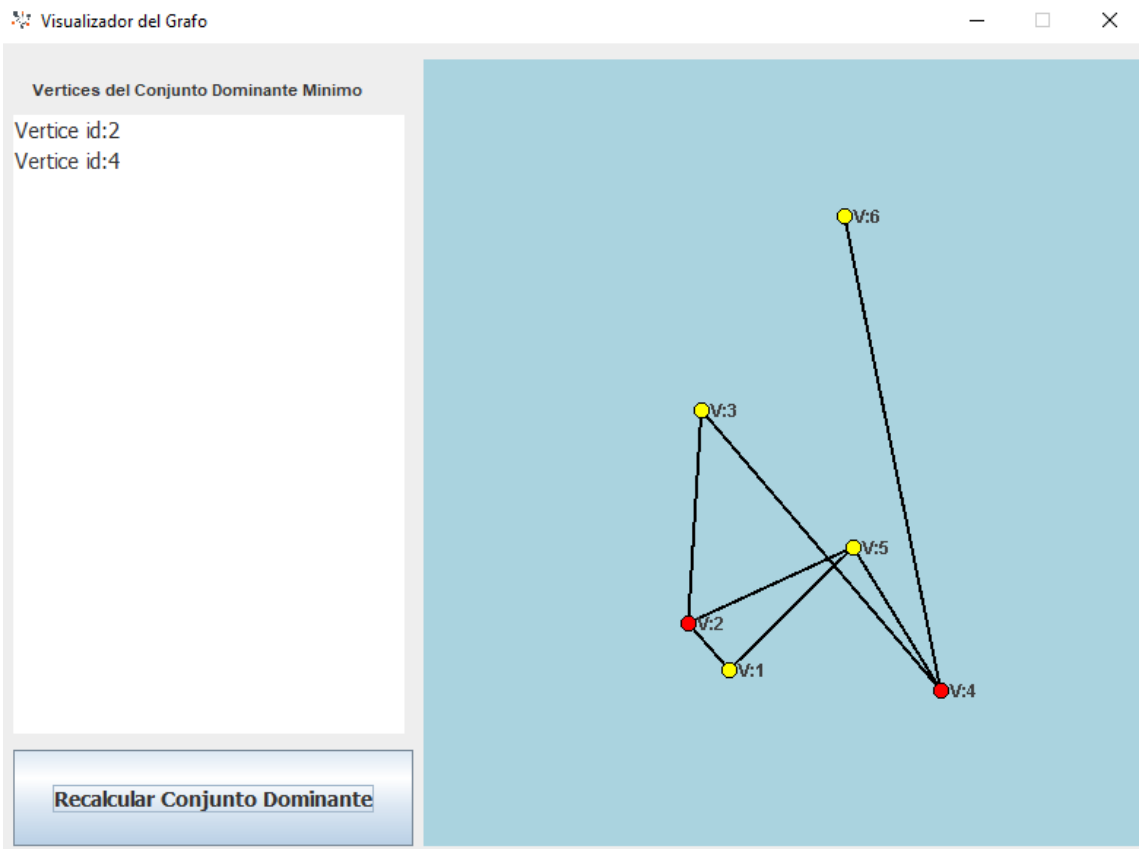
Donde cada línea, que representa la arista, llevara dos números, los cuales son los vértices de origen y de destino. Después, el lector de archivos leerá línea por línea el archivo y devolverá un grafo con las aristas y vértices seteados.



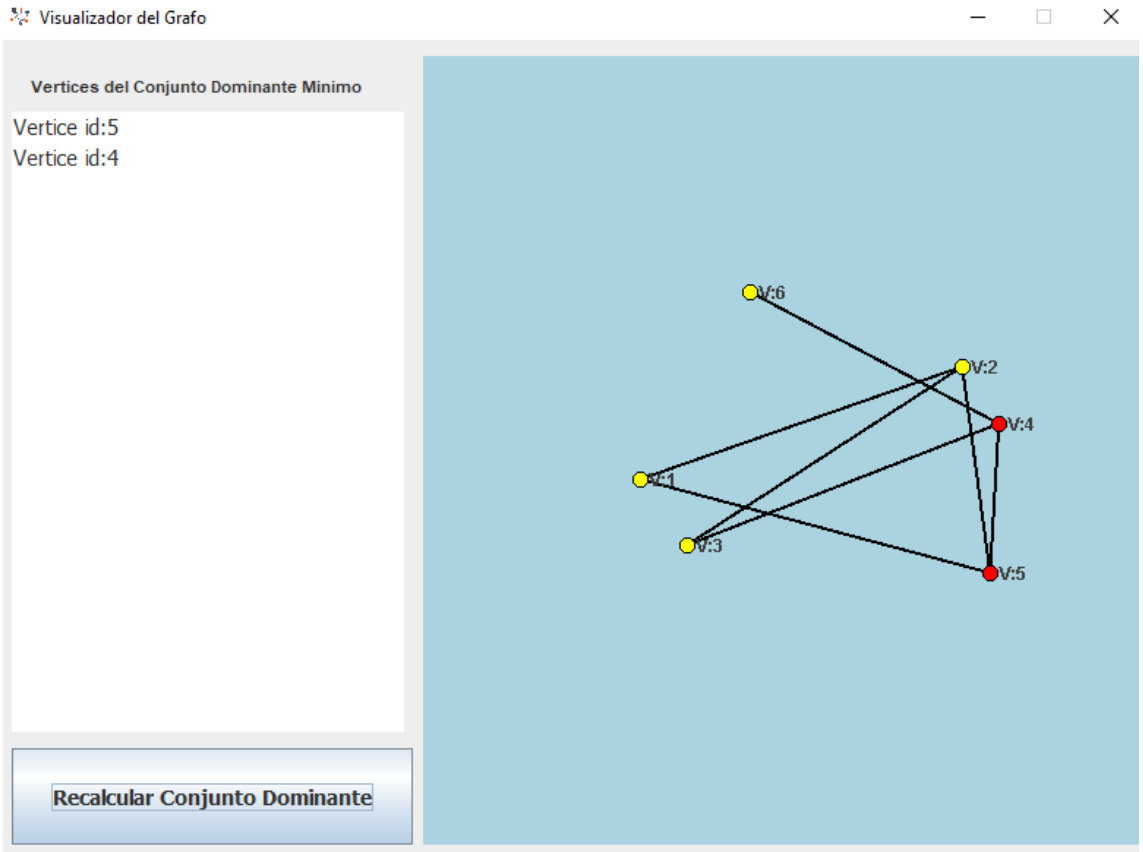
Luego de corroborar que los vértices estén correctamente, el usuario presiona en “Ver Grafo” para pasar a la pestaña de visualización, donde vera el grafo como una pestaña de JMap, y al lado una lista con todos los vértices pertenecientes al Conjunto Mínimo Dominante en una lista al lado del mapa.

(Aclaración: para este caso, se usa el grafo mostrado como ejemplo en la consigna del Trabajo Practico)

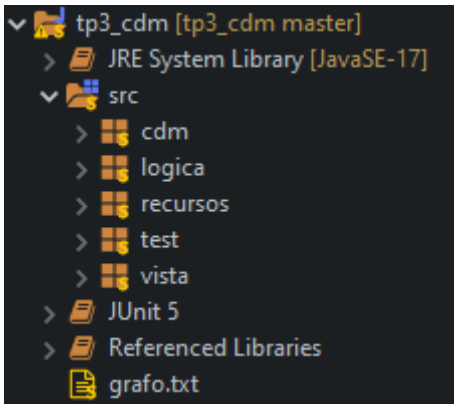




Si el usuario presiona nuevamente el botón, el Conjunto se recalculara y mostrara uno distinto al anterior (si es posible).



Implementación:



Implementando las arquitecturas usadas en los trabajos anteriores, se dividió en varios paquetes

La aplicación se ejecuta desde la clase **Main** desde el paquete **cdm**, para facilitar su identificación y que no colisiones con las demás clases del proyecto.

PACKAGE VISTA

Están presentes las clases de las ventanas: **VentanaMenu** y **VentanaGrafo**. Estas son las ventanas donde interactúa el usuario para la visualización de su grafo y su Conjunto Mínimo Dominante.

PACKAGE TEST

Contiene los tests de JUnit para cada una de las clases en el paquete “logica”, así poder agilizar la detección de errores y enfocarnos en resolver el eje central del trabajo

PACKAGE RECURSOS

Además del archivo multimedia usado como icono de la aplicación, está la clase **Arista**, la cual, si bien forma parte del grafo, no tiene ningún peso lógico en la aplicación de los algoritmos, mas solo es una ayuda para la visualización del grafo.

PACKAGE LÓGICA

Se encuentran varias clases indispensables para el funcionamiento de la aplicación:

-La clase **Vertice** representa cada nodo parte del grafo. Este tiene unas coordenadas predefinidas, y su propia lista de vecinos.

-La clase **Grafo**, donde se representa un grafo no dirigido (con aristas sin peso) implementado mediante listas de vecinos individuales para cada vértice. Verifica la validez de los vértices y las aristas que lo componen, y tiene funciones como “agregar Vecino” un vértice x, o “hacer Vecinos” dos vértices X e Y, vitales para el correcto funcionamiento del programa, y el Algoritmo Goloso.

-La clase **AlgoritmoGoloso** implementa, como su nombre lo indica, un algoritmo goloso sobre el grafo asignado para poder calcular su Conjunto Dominante Mínimo. El algoritmo comienza verificando la validez del grafo, y después de eso, hace una copia de los vértices originales del grafo, llamándolos “verticesNoMarcados”. Hasta que este conjunto no este vacío, el algoritmo ejecutara esta seguidilla de instrucciones: del conjunto, se extrae el vértice que tenga mas cantidad de vecinos, si este vecino tiene hojas (es decir, vértices que tienen un solo vecino), se eliminan todas sus hojas, y si no tiene, se verifican los vecinos de los vecinos de este vértice si no tienen hojas, y solo si no tienen, son eliminados. Se añade el vértice al conjunto CDM y se elimina de los no marcados, y así sucesivamente hasta que el conjunto original se vacié. Solo cuando esta vacío, el ciclo while termina y se devuelve el conjunto dominante mínimo.

-La clase **LectorArchivo** se encarga de convertir el archivo de texto ingresado por el usuario (respetando el formato) en un grafo. Para cada línea identifica los números ingresados, si el primer número detectado no existe en el grafo, lo crea, e igual con el segundo, crea y añade su arista correspondiente, y hace vecinos a los vértices identificados, usando el método propio del grafo.

Problemas Encontrados:

La lectura del archivo:

```
private static void procesarLinea(String linea, Grafo grafo) {
    String[] partes = linea.split(" ");
        if (partes.length == 2) {
        int id1 = Integer.parseInt(partes[0]);
        if (!grafo.existeVertice(id1)) {
            Vertice vertice = new Vertice(id1);
            grafo.agregarVertice(vertice);
        }
        int id2 = Integer.parseInt(partes[1]);
        if (!grafo.existeVertice(id2)) {
            Vertice vertice = new Vertice(id2);
            grafo.agregarVertice(vertice);
        }
        Vertice ver1 = grafo.buscarVertice(id1);
        Vertice ver2 = grafo.buscarVertice(id2);
        Arista ar = new Arista(ver1,ver2);
        grafo.agregarArista(ar);
        grafo.hacerVecinos(ver1, ver2);;
    }
}
```

En un principio, era necesario primero crear el vértice en el archivo de texto para después poder trabajarlo. Esto se elimino por ser poco practico para el algoritmo y para hacer el input mas dinámico para el usuario.

Problema de hojas sueltas:

```
public static Set<Vertice> vecinosHojas(Vertice ver) {
    Set<Vertice> vecinosHojas=new HashSet<>();
    for(Vertice v:ver.getVecinos()) {
        if(v.getVecinos().size()==0) {
            vecinosHojas.add(v);
        }
    }
    return vecinosHojas;
}
```

Después de haber ajustado el algoritmo goloso para el CDM, en determinados grafos se observaba el problema de que los vértices que eran hojas quedaban sin cubrir por el este mismo. Por eso se decidió hacer la doble verificación de hojas a cada vértice que tenga mas vecinos del conjunto, ya que un vértice que tiene al menos un vecino hoja va a ser parte si o si del CDM