Programación III

# TRABAJO PRÁCTICO 2:
# “TEMIBLE OPERARIO DEL RECONTRA ESPIONAJE”

Informe


**INTEGRANTES:**

- ALEGRE, Maximiliano Ariel.
- BAEZ, Tomás Luciano.
- BERINI, Bruno Nicolas.
- IBARRA, Milagros Abril.

**COMISIÓN:** 02.

**PROFESORES:**

- BERTACCINI, Daniel.
- CARRILLO, Gabriel.


## Introducción

Es necesario enviar un mensaje de manera segura a toda una red de espías que se encuentran en territorio enemigo. Para ello, en este trabajo práctico, se modeló un grafo, donde los espías son vértices y los posibles encuentros entre ellos son aristas con una probabilidad de intercepción. El objetivo es encontrar un árbol generador que minimice el riesgo de intercepción en el encuentro más peligroso, es decir, un árbol generador con mínimo cuello de botella. Para lograr este objetivo, se utilizará el algoritmo de Prim o Kruskal.

Para la implementación de este código, se utilizó la arquitectura *MVC* (Model, View, Controller). Las clases de negocio, es decir, la lógica detrás del programa junto con sus tests se encuentra en el paquete *model*, mientras que la interfaz visual se encuentra separada en *view*. El paquete *controller* es el puente entre ambos paquetes. Es el encargado de hacer las llamadas al código de negocios que la GUI necesita para funcionar correctamente.

## Instrucciones

Para iniciar la aplicación, se debe ejecutar la clase *InterfazMenu*. Se le presenta al usuario con una pantalla, donde encontrará una breve explicación del funcionamiento de la aplicación. El usuario podrá elegir entre ingresar los datos de sus espías manualmente o por archivo. Si elige manualmente, deberá ingresar la cantidad de espías y sus nombres. Una vez confirmados, podrá ingresar las conexiones entre espías. De lo contrario, se abrirá un archivo de extensión .json, donde deberá ingresar los datos necesarios para crear las conexiones. En este caso, espía1, espía2 y probabilidad, donde los espías representan un vértice y su vecino, y la probabilidad el peso.

Si los datos fueron ingresados correctamente, se abrirá un mensaje que le permite elegir el algoritmo por el cuál se recorrerá el árbol: Prim o Kruskal. Independientemente de su elección, al final se mostrarán dos tablas, una con todas las conexiones y otra con las conexiones necesarias para que el mensaje llegue a todos los espías en orden, junto con el tiempo de ejecución de la resolución.

## Implementación

Para la implementación de este proyecto, se utiliza la arquitectura MVC, ya que permite modularizar más el código en general y poder lograr mayor legibilidad, evitando acoplamiento entre clases. Además, se evita la sobrecarga de métodos en las interfaces visuales GUI.

En línea con el enfoque MVC, se estructuró el proyecto en cuatro paquetes: el paquete de negocio (model); paquete de interfaz (view); paquete de controlador (controller), y adicionalmente un paquete model.testing para pruebas unitarias en el paquete model.

En el paquete correspondiente a la lógica del programa (model) se incluyen clases relacionadas con grafos, algoritmos para encontrar el Árbol Generador Mínimo (usando Prim o Kruskal), hallar el orden por Breadth-First Search (BFS) y clases que manejan la lógica de los archivos JSON (ya que como se mencionó anteriormente, se puede utilizar tanto la interfaz por carga de datos manuales, como la de carga de datos por archivo .Json).

En cuanto a los puntos clave para implementar la lógica, cada vértice representa un espía distinto, mientras que cada arista simboliza la conexión entre los espías. Cada arista tiene un peso, que se corresponde a la probabilidad de intercepción entre las conexiones.

Siguiendo en orden de MVC respectivamente, en el paquete View solo se encuentran las interfaces para cargar las ventanas y botones para guardar nombres de espías, sus conexiones, probabilidades de intercepción y generar el Árbol Generador Mínimo (AGM). La vista interactúa exclusivamente con el controlador, lo cual permite delegar responsabilidades finalmente a la clase de negocio.

También se implementaron observadores en las GUI que requieren ser notificadas ante cambios en el grafo correspondiente al modelo.

Finalmente, el paquete Controller implementa métodos necesarios y utilizados por las interfaces para delegar el trabajo “pesado” al modelo. El controller simplemente realiza llamadas a funciones de la clase Grafo, Kruskal, Prim, entre otras clases de negocio, y se deriva la responsabilidad de Controller a actuar de intermediario.

A continuación, se profundizará en cada paquete.

## Lógica de Negocio

En el paquete Model se encuentran las clases que permiten abstraer el problema planteado en un problema de grafos, vértices y aristas con peso.

En principio, se necesitaba saber puntualmente qué vértices son vecinos de otros. Por lo tanto, se implementó el grafo como lista de vecinos, lo cual permite acceder en tiempo O(1) a los vértices *b* que se relacionan con un vértice *a* cualquiera.

Luego, para almacenar las probabilidades, es decir, el peso de las aristas, se utilizó HashSet de tipo Arista. Esta estructura garantiza que las aristas no se repitan (igualmente, se verifica que esto no pase en la interfaz). Cada elemento del HashSet tipo Arista contiene los dos extremos y la probabilidad.

Este proyecto contiene los dos algoritmos vistos en la teoría, que permiten obtener el árbol generador mínimo: Prim y Kruskal.

Ambos algoritmos tienen como características en común la verificación del grafo pasado por parámetro. Es necesario que el mismo sea conexo para que el algoritmo funcione, por lo que no es posible hacerlo con un grafo que no lo sea. Al iniciar el grafo AGM, se indica la cantidad de vértices, para poder comprobar que todos se recorran y ninguno quede fuera del subgrafo.

El algoritmo de Prim inicia el AGM agregando a la lista de vértices marcados el primer vértice (de índice 0). Luego, lo construye mediante un bucle que termina una vez que todos los vértices se hayan marcado. En cada iteración, se busca la arista con peso mínimo, revisando los vecinos de los vértices marcados para encontrar el camino mínimo, y así generar el AGM propiamente dicho.

Por otra parte, el algoritmo de Kruskal se inicializa con una lista de aristas ordenadas según su peso de menor a mayor (*obtenerAristasOrdenadas()*). Luego, se utiliza Union Find para gestionar los conjuntos de vértices y su unión. Selecciona las aristas de menos peso y verifica que los extremos estén en diferentes componentes conexas para agregar la arista al AGM mediante la función *find()*. Si esto se cumple, se llama a *union()* para agregar esta arista a la componente conexa.

Por último, las clases *Conexion* y *ConexionJSON* manejan la representación de datos en el formato ya mencionado, implementando métodos para serializar y decodificar los archivos, para guardar y recuperar la información de una manera eficiente.

## Controlador

El paquete Controlador consta de una sola clase que se encarga de realizar los llamados al código de negocio. Esta clase contiene al objeto Grafo y a su AGM. A través de ella, la interfaz puede; asignar u obtener datos del Grafo (como crear aristas, asignar nombres a los vértices, consultar la existencia de estos, consultar por todas las aristas del grafo, etc), obtener el árbol generador mínimo con el algoritmo deseado (Prim o Kruskal) y procesar la lectura de un archivo de texto serializado con el formato JSON para posteriormente realizar los llamados al código de negocio que permitan crear el grafo con los datos obtenidos.

## Interfaz Visual (GUI)

La interfaz visual se divide en varias clases, siendo cada una su propia ventana. La aplicación se inicia con la clase *InterfazMenu*. Esta se encarga de mostrar el menú principal, indicando algunas instrucciones para el uso correcto del ejecutable, junto con dos botones que

llevan a la carga de datos. Dependiendo de la elección del usuario, se lo llevará a dos ventanas distintas.

*InterfazCargarDatosManualmente* e *InterfazVecinos* se encargan de la carga de datos manuales. La primera toma los datos de los espías. Se implementa sobre un ArrayList de JTextField, los cuales se generan dependiendo la cantidad de vértices ingresados, y utiliza funciones de la clase *Controlador* para asignar los nombres de cada uno. Una vez confirmados, la interfaz de Vecinos toma los datos de las conexiones. Es decir, recauda la información para agregar las aristas al grafo. Para ello, utiliza dos JComboBox, para indicar un vértice y su vecino, junto con un JTextField para la probabilidad. Además, implementa un JTable para mostrar las conexiones a medida que son agregadas por el usuario.

Por otro lado, *InterfazCargarDatosPorArchivo*, como su nombre indica, implementa un archivo de extensión .json para la carga de las conexiones o aristas del grafo. La clase utiliza dos objetos de tipo JButton para la apertura y guardado del archivo incrustado en la ventana gráfica. Esta verifica que los datos sean correctos antes de guardarlo. Es decir, es necesario que el archivo tenga el formato correcto y que el grafo sea conexo para poder pasar a la siguiente y última fase.

*InterfazResultado* es la última ventana. Esta activa un JOptionPane, presentando al usuario con las dos opciones de algoritmos de resolución posible (Prim o Kruskal), para luego abrir la ventana de resultados. La misma utiliza dos JTable, una para mostrar todas las conexiones ingresadas y otra para el camino mínimo final, mostrando las aristas en orden de cómo deben encontrarse los espías. De esta forma, el usuario puede ver los resultados del grafo que ingresó, junto con el tiempo que tardó el proceso en realizarse.

Por último, la clase *Auxiliares* implementa cinco funciones funciones: *mostrarError*, tomando por parámetro la pantalla en la que debe aparecer y el error encontrado; *mostrarMensaje*, que toma los mismos parámetros, pero solo se utiliza para avisar al usuario cuando algo se hizo correctamente o los datos se están procesando; *pasarALaInterfazResultado*, que cambia de la ventana actual a la de *InterfazResultado*; *pasarALaInterfazVecinos,* que cambia de la ventana actual a la de *InterfazVecinos*; y *cerrarVentanaActual*, que cierra la ventana pasada por parámetro.

## Prim vs. Kruskal

Gracias a la implementación de ambos algoritmos, es posible comparar los tiempos de ejecución de ambos. A partir de los siguientes grafos, se mostrará la diferencia entre ambos.

Es necesario aclarar que el tiempo de ejecución puede variar dependiendo de la potencia de la computadora donde se ejecuta o de la cantidad de eventos de la que se esté encargando. Por lo tanto, esta comparación es un estimado, teniendo en cuenta también la complejidad vista en clase. Se trata de una forma de visualizar las diferencias entre uno y otro, no de una comparación definitiva.

**→ Con grafo de 10 vértices:**

- *Prim:* 0.003s.
- *Kruskal:* 0.004s.

**→ Con grafo de 50 vértices:**

- *Prim:* 0.009s.
- *Kruskal:* 0.005s.

**→ Con grafo de 100 vértices:**

- *Prim:* 0.027s.
- *Kruskal:* 0.005s.

Es posible ver, a partir de estos ejemplos, que no parece haber gran diferencia con grafos con menor cantidad de vértices. Al contrario, Prim parece ser un poco más eficiente en estos casos que Kruskal. Sin embargo, a más cantidad, parece haber una diferencia significativa entre el tiempo de ejecución de Prim y de Kruskal. Basándose en estos datos, se podría decir que Kruskal será más eficiente que Prim, pero se vuelve más evidente con ejecuciones en grafos con gran cantidad de vértices. Lo mismo puede verse en la teoría, donde la complejidad de Kruskal con la implementación de Union-Find es mejor que la de Prim.

## Extras

Como parte de los objetivos complementarios, se implementaron algunos puntos extras. Por empezar, el usuario tiene la posibilidad de ingresar los datos en un formato JSON. El mismo se procesa y se guarda en un archivo “Datos.json”, que se formatea cada vez que se cierra y se vuelve a ejecutar la aplicación. Se toman los datos de todas las conexiones, verificando que el formato se respete y que el grafo sea conexo antes de guardarlo.

Además, el usuario puede elegir entre el algoritmo de Kruskal y de Prim, y comparar así los tiempos de ejecución entre ambos. No es posible compararlos de forma paralela. Sin embargo, es posible ingresar el mismo grafo al volver a ejecutar el programa y elegir otro algoritmo.

Por último, al resultado final del AGM se recorre con BFS para mostrar el camino en orden de todos los encuentros de espías. Esto se muestra a través de la tabla del resultado final, donde se muestra como debería pasar el mensaje para tener la menor probabilidad de ser interceptado.