Informe de Trabajo Práctico n°2:
“Temible operario del recontraespionaje”

Alumno:
- Avalos, Nazareno

1) Implementación de la lógica de la aplicación:

- ***Class “TemibleOperarioDelRecontraespionaje”:***

Esta clase está dedicada exclusivamente a desarrollar la lógica de la aplicación. Para esto, necesitamos un grafo ‘g’ que representa la red de espías (redEspias), un grafo ‘t’ que representa el cómo se tiene que distribuir el mensaje entre espías para que tenga el menor riesgo de interceptación (redConArbolMinimo) y la cantidad de vértices del grafo, en este caso, representando la cantidad de espías de la red.

Tenemos, entonces:
- **Espias “redBase”**: representa al grafo ‘g’.
- **Espias “redConArbolMinimo”:** representa al grafo ‘t’ de ‘g’. Es decir, el árbol mínimo de “redBase”.
- **int “cantEspias”:** variable que nos permite almacenar la cantidad de vértices del grafo (redBase) para luego crearlo.

Luego tenemos las operaciones:

Firma del método:	Funcionalidad:
prepararKruskal()	Inicializamos el grafo “redBase” con la implementación de una matriz, sabiendo previamente cuántos vértices existen: Espias_matriz(cantidad de espías)
prepararPrim()	Inicializamos el grafo “redBase” con la implementación de una lista de adyacencia, sabiendo previamente cuántos vértices existen: Espias_Lista_Ad(cantidad de espías)
agregarNombreALosEspias (int numEspia, String nombre)	Al espía “numEspia”, le brindamos el nombre que el usuario haya pensado.
agregarDistanciaEntreEspias (int Espia1, int Espia2,	Se crea una conexión (arista) entre el ‘Espia1’ y ‘Espia2’ con un riesgo de interceptación dado por ‘riesgo’.

double riesgo)	
generarArbolMinimo()	Según qué tipo de grafo se haya seleccionado usando previamente “prepararKruskal()” o “prepararPrim()”, se ejecutará uno de los dos algoritmos del mismo nombre: Kruskal o Prim, respectivamente.
esConexo()-boolean	Dado el grafo ‘redBase’, verificamos si es o no un grafo conexo, utilizando BFS.

2) Implementación de un grafo en contexto de una red de espías:

- ***Clase abstracta “Espias”:***

Esta es la clase principal, llamada “Espias”, que tiene dos subclases, “Espias_Lista_Ad” y “Espias_matriz”.

En primer lugar, se tiene como objetivo ser una clase abstracta para que sus dos subclases respeten los métodos propuestos, dado que, desarrollamos dos implementaciones para modelar una red de espías (el grafo ‘g’).
Estas dos implementaciones se desarrollan como dos subclases; de ahí las mencionadas clases “Espias_Lista_Ad” y “Espias_matriz”.

En segundo lugar, esta clase solamente tiene un array de Strings “nombresEspias” como variable principal, que será actualizada desde cualquiera de las dos subclases.
Esta variable, guardará todos los nombres de los espías sabiendo que el índice referencia a un espía de la red. *Ej: nombresEspias[0] = “pepe”.*
Cualquier espía válido tiene su representación como índice del array, además, el tamaño del array tiene que ser igual a la cantidad de espías.

- ***nombresEspias[].length*** == ***redEspias.cantEspias***

Observacion:

Sabemos (los presentes alumnos) que los nombres elegidos o el diseño de las tres clases se pueden mejorar o cambiar. Sin embargo, el tiempo que nos permitimos para pensar un diseño con nombres de variables y de métodos aclaratorios fueron de apenas 2(dos) días, ya que no quisimos perder mucho tiempo en esta parte por haber estado previamente batallando en hacer funcionar los algoritmos de Prim y de Kruskal, con y sin “UnionFind”.

A continuación se mencionan las operaciones abstractas principales:

Firma del método	Funcionalidad esperada:
agregarNombreAlEspia (int i, String nombre)	Dado un espía ‘i’, se tiene que verificar que existe el espía en dicha posición del array, luego se le asigna el

	nombre pasado por parámetro.
agregarArista(int i, int j, double peso)	Dado dos espias ‘i’ y ‘j’, se tiene que verificar que existen ambos en sus respectivas posiciones del array, luego se crea la “conexión” (arista) entre ambos con riesgo “peso”.
existeArista(int i, int j) -boolean	Se verifica si existe una conexión (arista) entre los espías ‘i’ y ‘j’. Devuelve ‘true’ si es así.
vecinos(int i) -Set<Integer>	Dado un espía ‘i’, se devuelve un Set con todos sus espías vecinos.
tienenNombreTodos() -boolean	Se verifica que todos los espías en el array tienen un nombre asignado.

- ***Clase “Espias_matriz”:***

Esta clase tiene como objetivo la implementación de un grafo mediante una matriz de adyacencia para luego ser utilizada específicamente por el algoritmo de Kruskal sin problemas. Por ello mismo, tiene una variable grafo que es una matriz de tipo ‘double’ y un ArrayList<Contactos> llamada aristas.

- **Double[][] “grafo”**: representa el grafo como una matriz. Esta matriz tiene que ser cuadrada, por lo tanto tiene un tamaño igual a la cantidad de espías al cuadrado.
- **ArrayList<Contactos> “aristas”:** representa una lista de todos los contactos (aristas) creadas.

Observación:

No se implementaron otros métodos particulares, pues se respetan los métodos por implementar de la clase padre.

- ***Clase “Espias_Lista_Ad”:***

Esta clase tiene como objetivo la implementación de un grafo mediante una lista de adyacencia para luego ser utilizada específicamente por el algoritmo de Prim sin problemas. Por ello mismo, tiene dos variables importantes:

- **int “vertices”:** guarda la cantidad de vértices del grafo.
- **LinkedList<Contactos> [] “listaAdyacencia”:** es un array de distintas LinkedList de tipo ‘Contactos’. Cada posición del array representa un espía, y el elemento de dicha posición representa la lista de las aristas que tienen asociados a ese espía.

Observación:

No se implementaron otros métodos particulares, pues se respetan los métodos por implementar de la clase padre.

3) Interfaz de Usuario:

Implementación:

- *Class "VentanaPrincipal":*

Aquí es donde se ejecuta la lógica de la ventana principal (JFrame) y que contiene la información de los cinco paneles (Inicio, Opciones, RedEspias, MostrarRed y Extras) que representan distintas ventanas para el usuario.

Su única peculiaridad es la de tener un botón "volver". Cuando se presiona, se vuelve al inicio y se reinicia el programa mediante el método:

- ***reiniciar():*** remueve los paneles añadidos previamente para luego generar unos nuevos, dando la idea de haberlo ejecutado por 1ra vez.

Sus métodos son referencias a los métodos de la lógica del juego:
"TemibleOperarioDelRecontraespionaje".

- *Class "Inicio" extends "JPanel":*

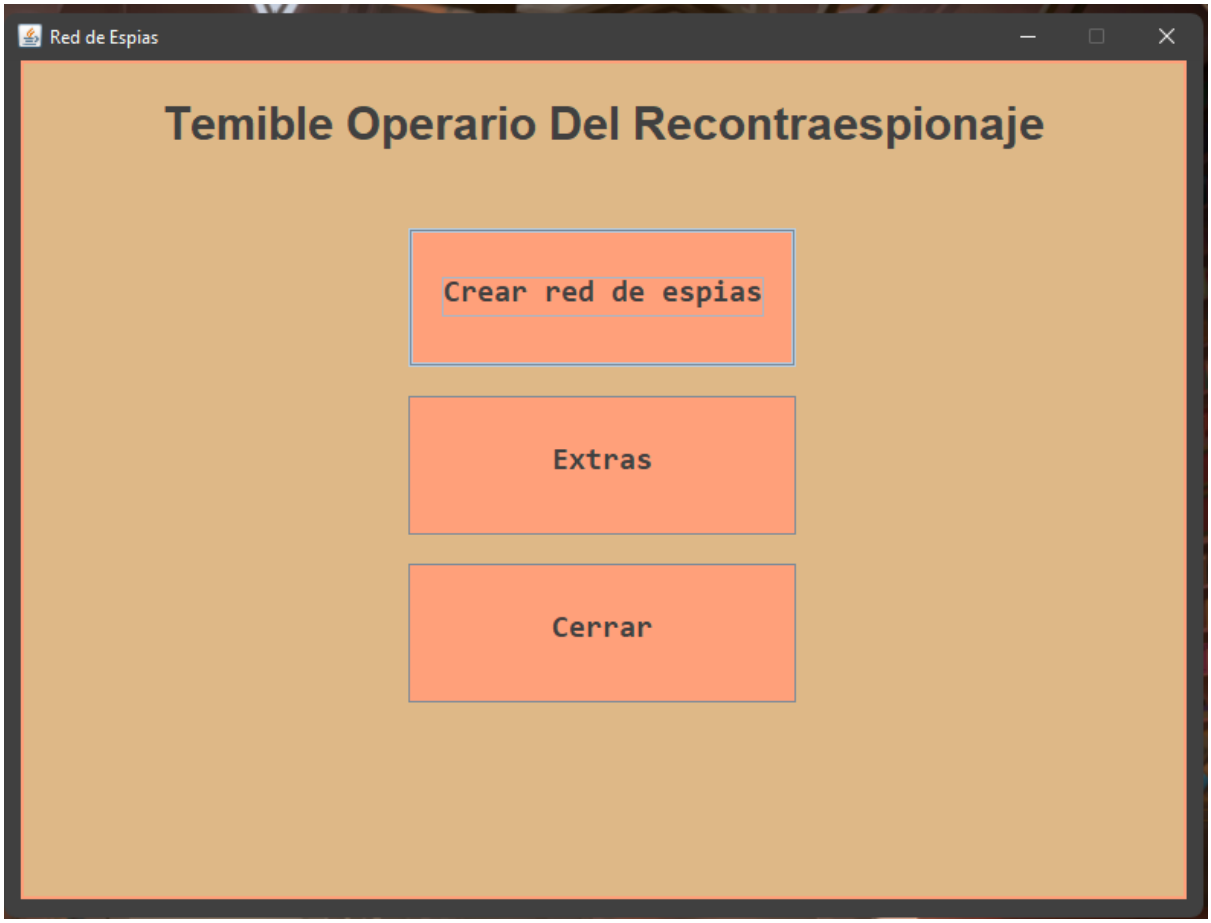
Aquí es donde se implementa la primera ventana que el usuario presencia.

Tiene tres botones principales:

- **JButton "CrearRedEspias":** una vez presionado se redirige al panel "Opciones".

- **JButton "Extras":** una vez presionado se redirige al panel "Extras".

- **JButton "Cerrar":** una vez presionado se cierra el programa.



- ***Class "Opciones" extends "JPanel":***

Aquí es donde se implementa la ventana donde el usuario brinda la cantidad de espías para crear la red (grafo 'g') y en donde se elige qué algoritmo quiere utilizar.

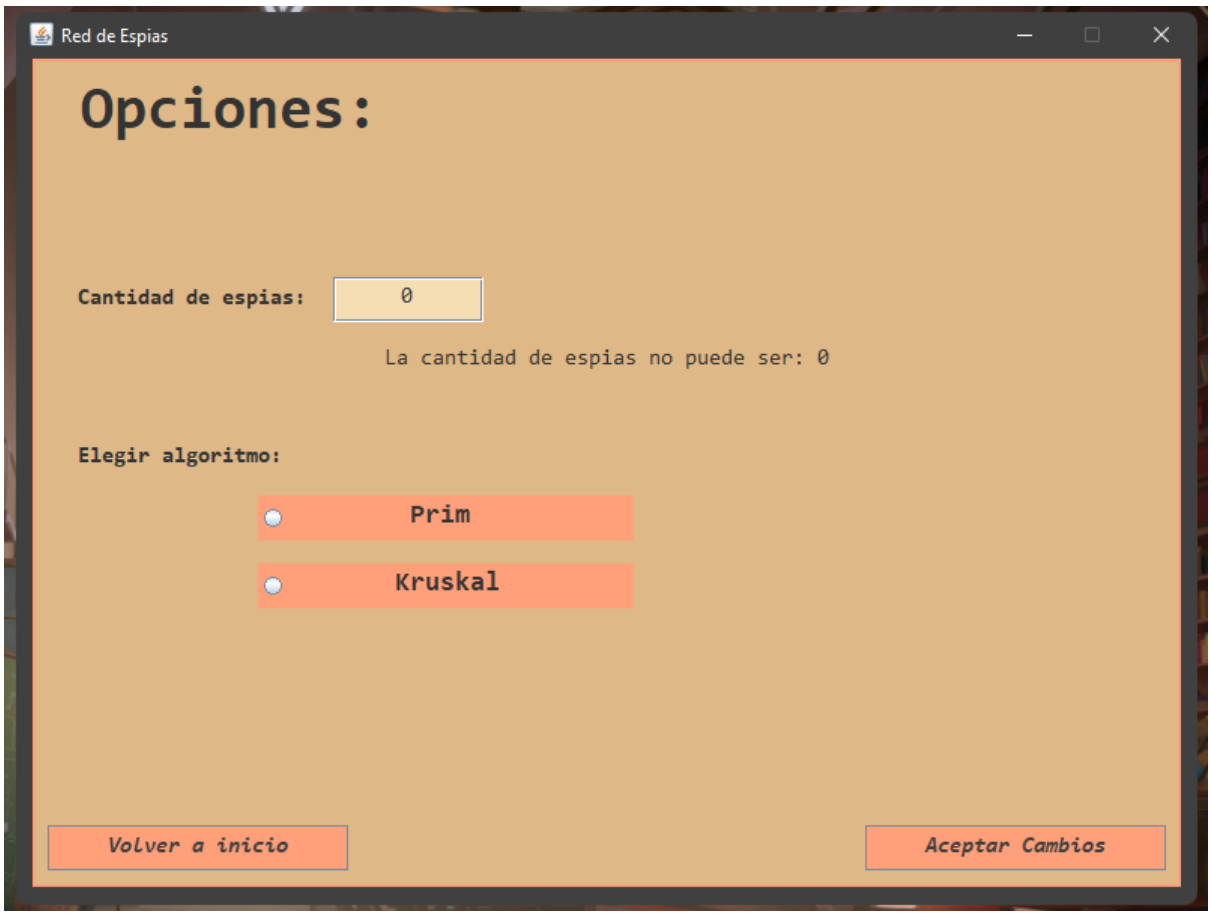
Tiene cuatro elementos principales:

- **JTextField "tfCantEspias":** toma el valor del usuario para crear la red (grafo).

- **Dos botones 'radiales' -> "rdbtnKruskal" y "rdbtnPrim":** según qué botón se haya presionado se inicializa la red, sabiendo que hay dos implementaciones desarrolladas (Espias_matriz, Espias_Lista_Ad).

- **JButton “btnAceptarCambios”:** una vez validado la cantidad de espías y que se haya seleccionado un algoritmo, se crea la red.

- **JLabel “excepciones”:** como su nombre lo indica, se le avisa al usuario de cualquier error o excepción que haya surgido.



- ***Class “RedEspias” extends “JPanel”:***

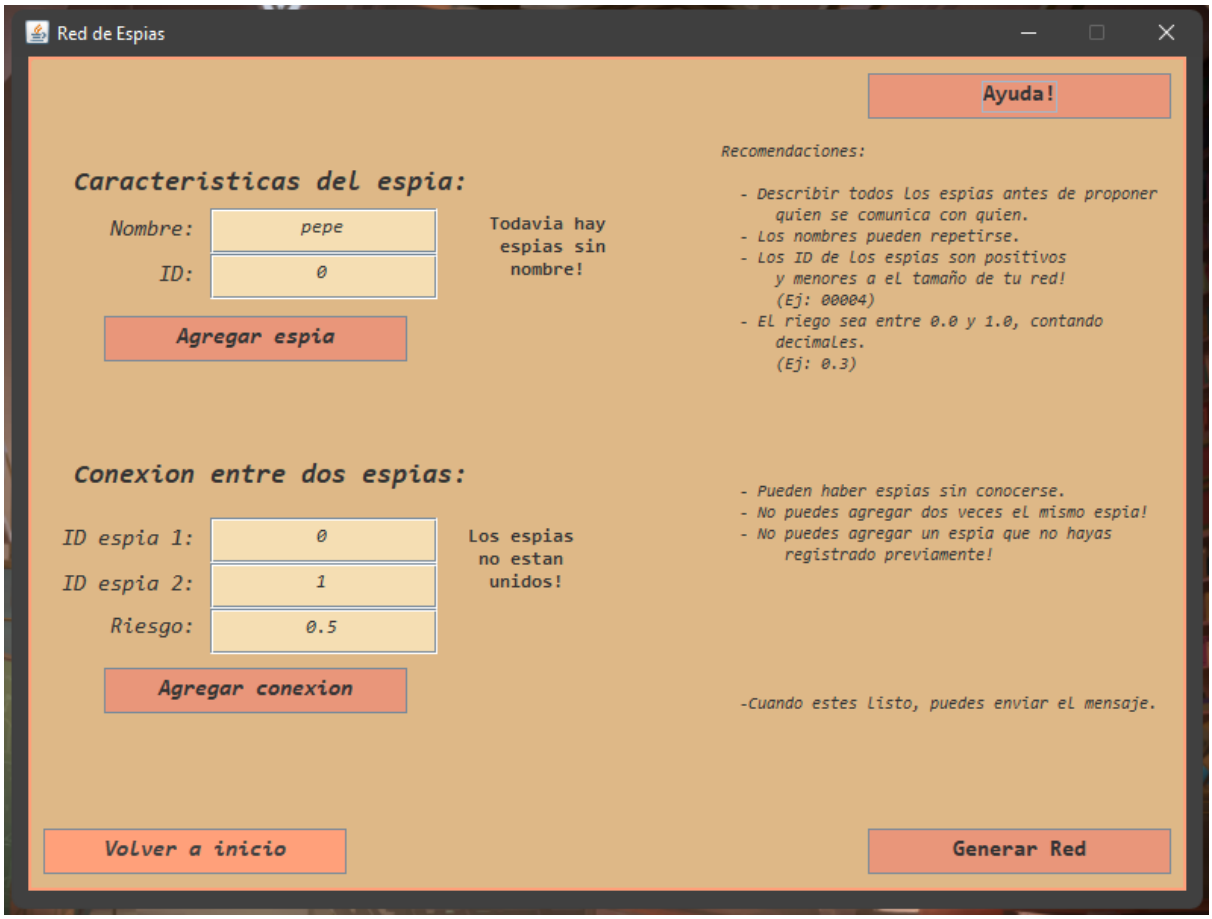
Aquí es donde se implementa la ventana para que el usuario pueda brindar más información acerca de los espías que existen en la red (nombre e id) y además qué conexiones existen en dicha red.

Esta clase contiene muchos elementos para que el usuario sepa qué es lo que tiene que hacer

Tiene cinco elementos principales:

- **JButton “btnAgregarEspia”:** una vez asignados un nombre y un id en sus respectivas casillas, se añade el espía con esos dos atributos.
- **JButton “btnAgregarConexion”:** una vez asignados ambos espías y el riesgo entre ellos, se añade la conexión a la red.
- **JButton “btnGenerarRed”:** una vez añadida toda la información pertinente (espías y conexiones), se muestra la red al usuario mediante otro panel -> “MostrarRed”.

- **<u>JButton</u> “ayuda”:** este botón le brinda un texto de ayuda al usuario cuando es presionado.
- **<u>JLabel</u> “excepciones”:** se le avisa al usuario de cualquier error o excepción que haya surgido.

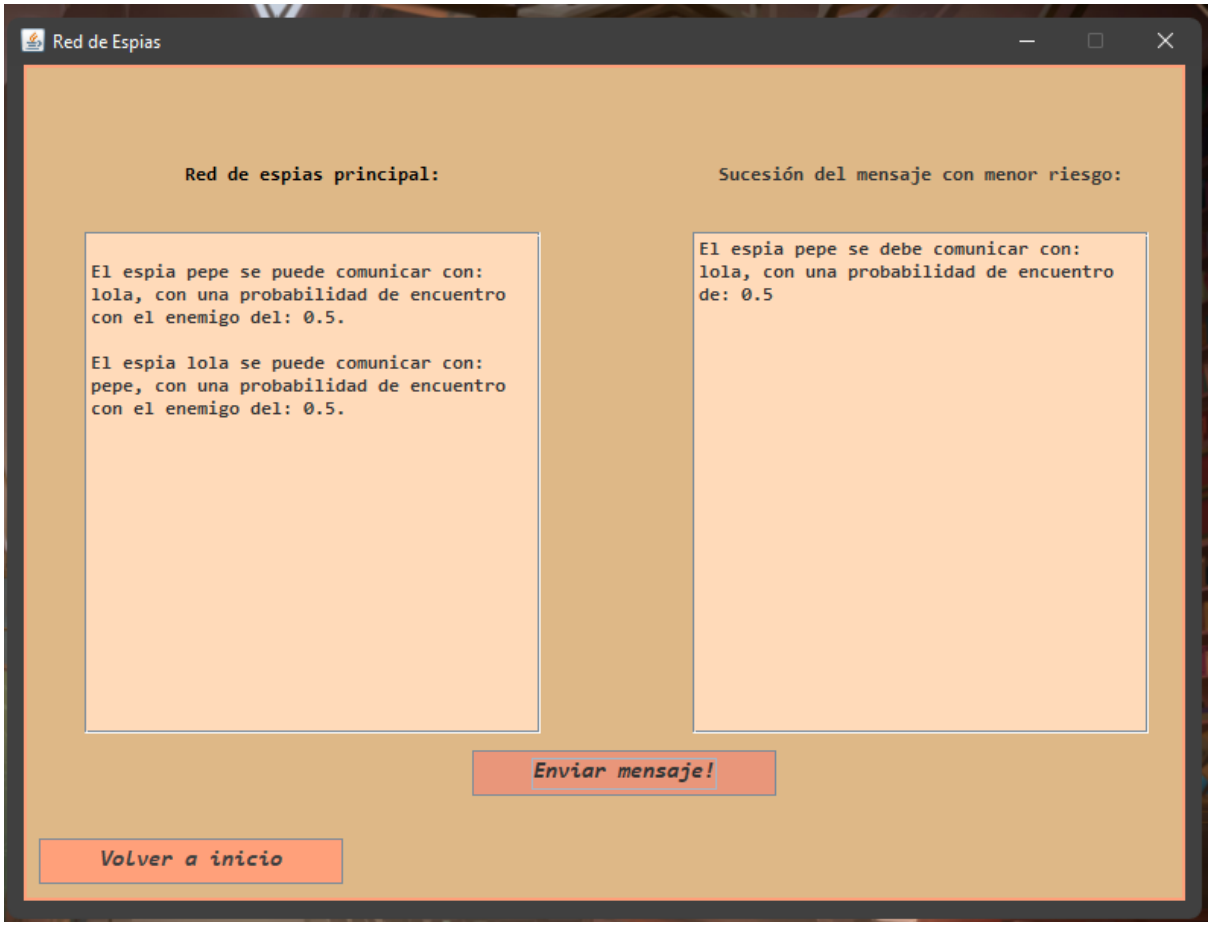


- ***Class “MostrarRed” extends “JPanel”:***

Aquí es donde se implementa la ventana para que el usuario pueda ver mediante texto, cómo es su red de espías. Además, se le permite al usuario conocer, también mediante texto, la sucesión de espías para que el mensaje tenga el menor riesgo de interceptación.

Tiene tres elementos principales:

- ***dos mini paneles*** para mostrar mediante texto:
  - el grafo principal
  - el grafo del árbol mínimo
- **<u>JButton</u> “enviarMensaje”:** una vez presionado se muestra en pantalla ambos grafos, cada uno en su respectivo mini panel.



- ***Class "Extra" extends "JPanel":***

Aquí es donde se implementa la ventana para que el usuario pueda observar y probar la velocidad de los algoritmos Prim y Kruskal, este último con y sin "Union-Find".

Para ello, puede brindar un valor cualquiera y jugar un rato.

Tiene cuatro elementos principales:

- tres JTextField:
  - 1) para mostrar Prim
  - 2) para mostrar Kruskal
  - 3) para mostrar KruskalUF
- JButton "btnStressTest": una vez seleccionado la cantidad de vértices para comparar la velocidad, se ejecuta el test.

-