ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Curso Académico 2022/2023

Trabajo Fin de Grado

GENERADORES DE DATOS PARA LA EXPERIMENTACIÓN CON ALGORITMOS

Autor: Pablo Cerdán López

Director: Jesús Ángel Velázquez Iturbide

RESUMEN

AlgorEx es una herramienta con fines educativos construida en Java y es utilizada en ciertas asignaturas en la Universidad Rey Juan Carlos para permitir que los alumnos puedan evaluar y comparar los algoritmos que desarrollan.

El propósito principal de este Trabajo de Fin de Grado ha sido el de completar esta aplicación, proporcionando nuevos generadores de datos que permitan probar algoritmos relativos a problemas más específicos.

Además de ello, se ha trabajado en otras cuestiones relacionadas con la interfaz de usuario con el fin de dotar de mayor coherencia a AlgorEx respecto a los nuevos cambios, así como en mejorar ciertas partes de la herramienta y resolver algunos errores, descritos en la presente memoria.

Palabras Clave:

- AlgorEx
- Java
- Generadores de datos
- Algoritmos

ÍNDICE

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGOS

1 Introducción

La algoritmia es un campo fundamental en la informática, que se centra en el diseño y análisis de algoritmos eficientes para resolver problemas computacionales. Con el crecimiento exponencial de los datos y las demandas de procesamiento en diversas ramas, desde la inteligencia artificial hasta la optimización logística, la importancia de desarrollar y comparar algoritmos cada vez es mayor. En este contexto, AlgorEx resulta una buena herramienta para evaluar la optimalidad y eficiencia de diferentes algoritmos.

AlgorEx, inicialmente llamada *Optimex* [1], es una herramienta de comparación de algoritmos implementada en Java. Este programa ofrece una amplia variedad de generadores de datos que permiten simular escenarios complejos y evaluar el rendimiento de los algoritmos bajo diferentes condiciones.

En esta memoria, se presenta el trabajo realizado en el marco de este Trabajo de Fin de Grado, que se ha centrado en el desarrollo de nuevos generadores de datos para AlgorEx, así como en la mejora y resolución de otras cuestiones o errores presentes en la herramienta. A través de un enfoque riguroso y meticuloso, se han abordado los desafíos asociados con la generación de datos realistas y representativos en AlgorEx, que permiten al usuario obtener resultados específicos a ciertos tipos de problemas no contemplados en la herramienta.

1.1 Motivación

Debido al uso de AlgorEx en asignaturas como Algoritmos Avanzados, en la que es necesario probar tipos de problemas resueltos mediante diferentes técnicas de algoritmia, surge la elección del tema de este TFG.

La herramienta ya presentaba una amplia variedad de generadores que permiten evaluar los algoritmos desarrollados por los alumnos, pero se ha considerado que sería aún más útil si pudiera realizarse dicha evaluación mediante la generación de juegos de datos específicos al tipo de problema que quiera resolverse. De esta manera, podrán compararse los resultados con un enfoque más realista, ofreciendo mayor facilidad a la hora de generar estos datos o evitando que tengan que despreciarse algunos por no cumplir con las restricciones que el usuario necesite.

1.2 Objetivos

Los objetivos de este trabajo se centran en la revisión, implementación y mejora de AlgorEx.

El objetivo principal pasa por desarrollar varios generadores de datos aleatorios, que deben ser decididos después de una fase de análisis junto al tutor del TFG. Estos generadores deben cumplir con las restricciones correspondientes y estar completamente integrados en la aplicación, incluyendo la interfaz de usuario.

Además del objetivo principal, se plantean objetivos secundarios que involucran corregir problemas existentes en la generación de juegos de datos repetidos, mejorar la interfaz de usuario para una mejor visualización y solucionar errores detectados en la generación de matrices, así como actualizar la importación y exportación de la información de los generadores.

1.3 Estructura de la memoria

Tras la introducción del presente documento, la estructura resultante de la memoria sigue de la siguiente manera:

Primero, se ha redactado una posible sesión de usuario con varias capturas de pantalla, para contextualizar a la persona que esté leyendo el documento, en caso de que nunca haya utilizado AlgorEx.

En segundo lugar, se procede a explicar cuál ha sido la metodología de trabajo de este TFG. Se expone de manera más detallada la especificación de objetivos, junto al proceso de desarrollo que ha seguido el alumno y las herramientas que ha utilizado para llevarlo a cabo.

Tras ello, siguen tres secciones más técnicas, centradas en el diseño, la implementación y las pruebas realizadas para la consecución de los objetivos descritos en la especificación.

La última sección incluye una valoración personal del trabajo por parte del alumno, así como una estimación del esfuerzo realizado y unas propuestas de posibles trabajos futuros.

La memoria es completada por las referencias y los dos anexos incluidos, en los que se adjuntan varias clases Java para probar AlgorEx y un manual de usuario que indica cómo importar el proyecto en el entorno IntelliJ IDEA.

2 Sesión de Usuario

Para contextualizar a quien esté leyendo este documento, se ha decidido elaborar un ejemplo de sesión de usuario utilizando AlgorEx, la herramienta de comparación de eficiencia algorítmica. De esta manera, podrá tener una idea más clara acerca del funcionamiento de la aplicación.

Las capturas de pantalla y los ejemplos que se presentan a continuación pertenecen a la versión de AlgorEx previa al desarrollo del TFG, a excepción de la Ilustración 3, que incluye cambios realizados por el alumno.

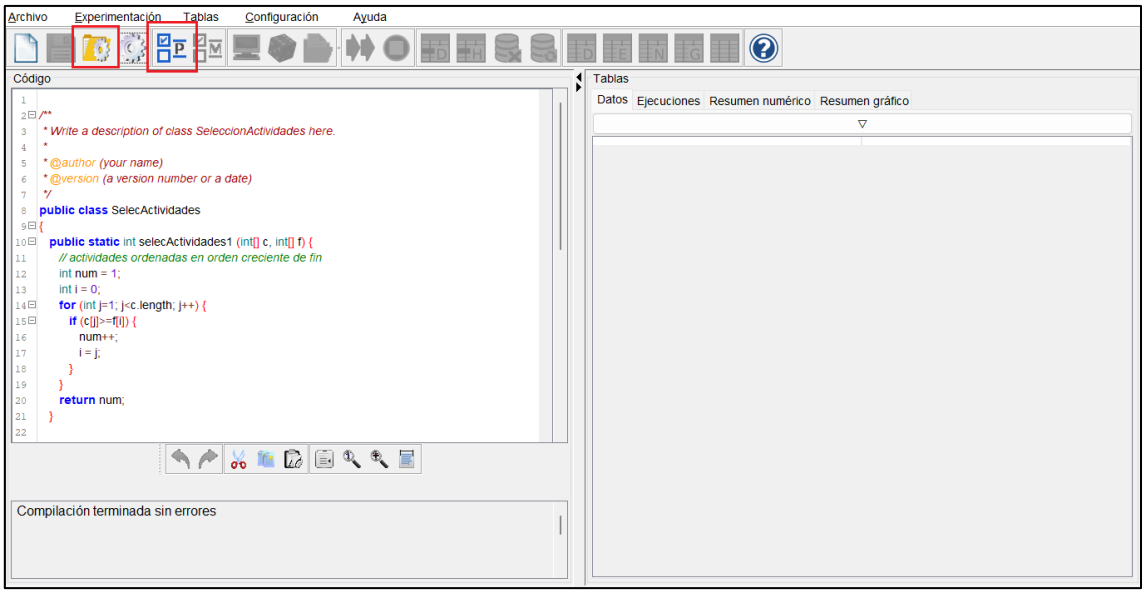


Ilustración 1. Pantalla principal AlgorEx

Cuando abrimos la aplicación, nos encontramos con una pantalla similar a la vista en la Ilustración 1 (en este caso, ya hemos cargado una clase Java). Los elementos que destacan en esta imagen son los siguientes:

1. Botón de “Cargar y compilar clase”. Haciendo clic en él, se puede seleccionar una clase Java que contenga los algoritmos que deseamos comparar en AlgorEx.
2. Botón de “Seleccionar problema”. Antes de ejecutar, es obligatorio pulsar en esta opción. Se abrirá una ventana en la que debemos confirmar los métodos a comparar, así como el criterio de experimentación (optimalidad, eficiencia en tiempo o nulo), el objetivo del problema y el tipo de medidas que se van a utilizar.

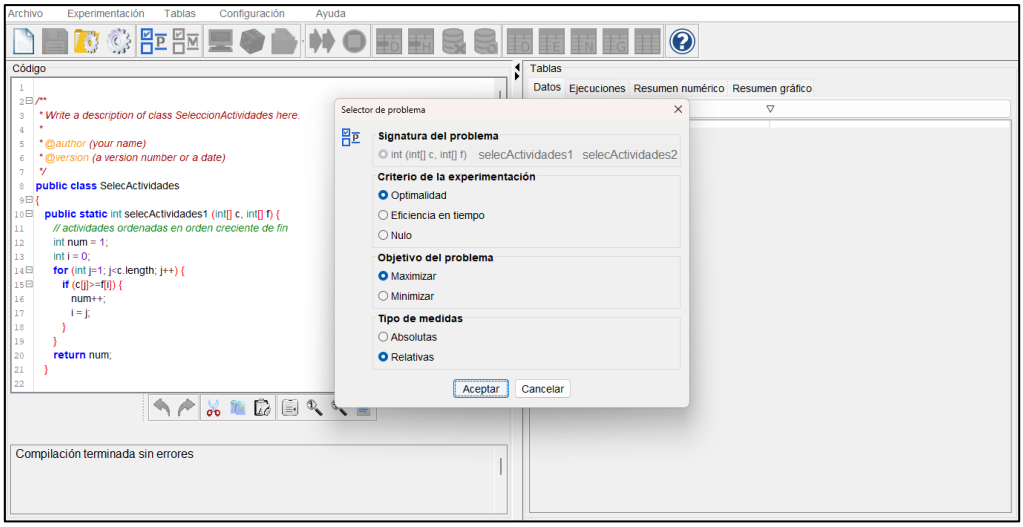


Ilustración 2. Selección de signaturas

Tras seleccionar la signatura del problema (en este caso se ha escogido el criterio de optimalidad, con medidas relativas y objetivo de maximización), veremos que se activan los botones de generación de datos. Seleccionando el icono del dado, se procederá a la generación de datos aleatorios. Se indica el número de juegos de datos que se quieren generar y se procede a la selección de restricciones de dichos datos.

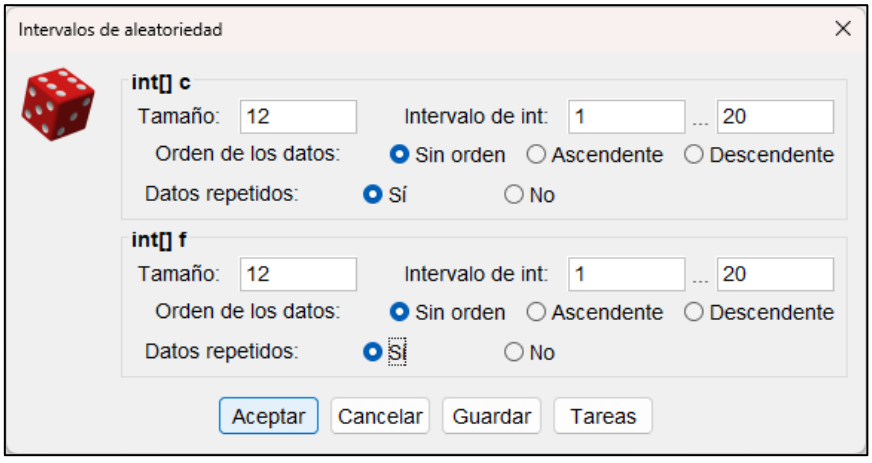


Ilustración 3. Selección de restricciones

Para este ejemplo, vamos a dejar todo seleccionado por defecto, de manera que generaremos cien conjuntos de dos vectores de enteros de tamaño 12, cuyos elementos estarán comprendidos entre 1 y 20 y en los que puede haber repetición de datos.

Tras generar estos juegos de datos, al pulsar en el icono de las flechas verdes, se realizarán las 100 ejecuciones de los métodos comprendidos en la clase y podremos ver y exportar un resumen y unas comparativas de dichas ejecuciones.

Tablas

Datos | Ejecuciones | Resumen numérico | Resumen gráfico

Medidas	selecActividades1	selecActividades2
Núm. ejecuciones	100	100
Núm. ejec. válidas del método	100	100
Núm. ejec. válidas en total	100	100
% Soluciones subóptimas	0,00 %	0,00 %
% Soluciones óptimas	100,00 %	100,00 %
% Soluciones sobreóptimas	0,00 %	0,00 %
% Diferencia media subóptima	0.00 %	0.00 %
% Diferencia máxima subóptima	0,00 %	0,00 %
% Diferencia media sobreóptima	0,00 %	0,00 %
% Diferencia máxima sobreóptima	0,00 %	0,00 %

Ilustración 4. Resumen numérico del primer ejemplo

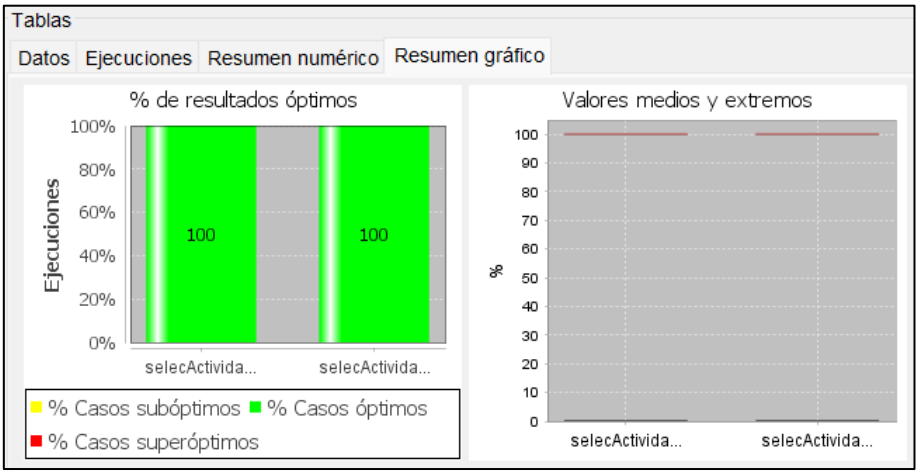


Ilustración 5. Resumen gráfico del primer ejemplo

Ambos resúmenes, junto al de los resultados de las ejecuciones, pueden ser exportados en formato tabla.

Ilustración 6. Iconos exportación de tablas

Haciendo clic en los iconos mostrados en la Ilustración 6, es posible exportar las tablas generadas que hemos visto. De izquierda a derecha: “Exportar datos”, “Exportar ejecuciones”, “Exportar resumen numérico”, “Exportar resumen gráfico” y “Exportar todas las tablas”.

Ahora, vayamos con otro caso: al volver a pulsar en el botón “Cargar y compilar clase”, seleccionamos una clase distinta para probar otros algoritmos. Presionamos en “Aceptar” para que se borren los datos de la última ejecución y actuamos de la misma manera que en el ejemplo anterior.

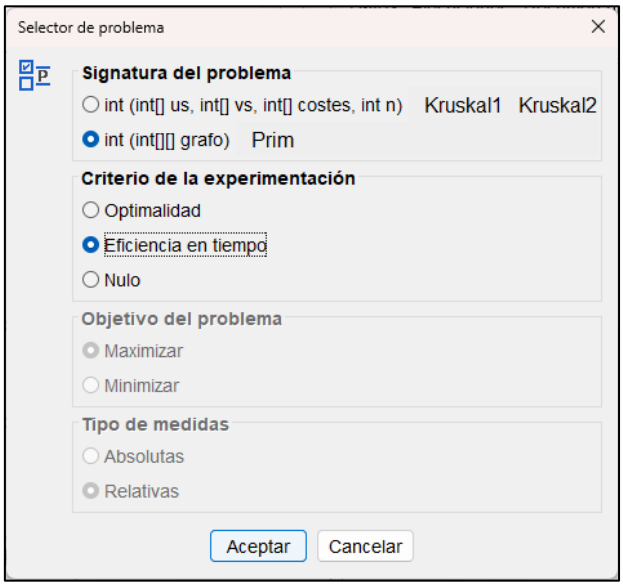


Ilustración 7. Signatura del problema. Caso 2

Para este segundo ejemplo, hemos seleccionado un método que recibe como entrada una matriz de enteros (matriz de adyacencia correspondiente a un grafo) y vamos a generar 100 juegos de datos aleatorios de la misma manera que antes.

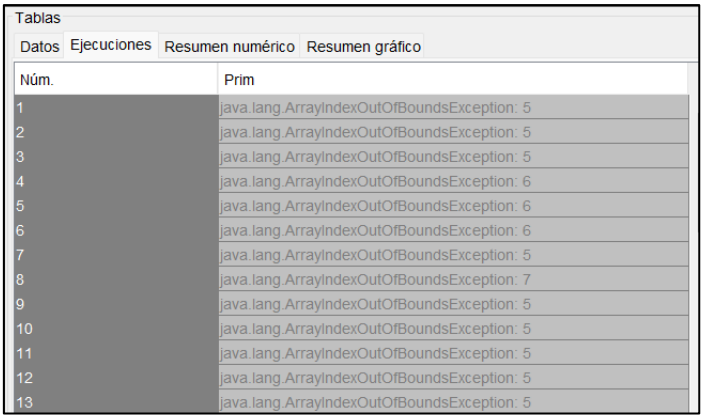


Ilustración 8. Ejecuciones segundo caso

Como vemos en la Ilustración 8, se está obteniendo una excepción en todas las ejecuciones del método, por lo que no se reciben datos para el resumen gráfico y para el resumen numérico todo sería un 0%.

Con este ejemplo, se puede ver que no siempre se pueden comparar los datos de los algoritmos, pues es necesario asegurarse de que el código es correcto y no existe ningún fallo, no solo en la compilación.

3 Metodología

En esta sección, se explicarán con detalle los objetivos mencionados en la sección 1.2 del presente documento. Tras ello, se procederá a detallar cuál ha sido el proceso que se ha seguido para el desarrollo del Trabajo Final, así como las herramientas utilizadas para ello.

3.1 Especificación

Antes de describir los objetivos del TFG, tanto el alumno como el tutor tuvieron que realizar un análisis de AlgorEx y de distintos tipos de problemas algorítmicos, con el fin de establecer un camino claro por el que continuar. Tras ello, pudieron definir un objetivo principal, acompañado de otros algo más secundarios:

- **Objetivo principal. Desarrollo de los generadores de datos:**

  Se ha tomado la decisión de implementar los siguientes generadores de datos:

  - Generador de permutaciones
  - Generador de tareas
  - Generador de grafos aleatorios por probabilidad
  - Generador de grafos aleatorios por número de arcos
  - Generador de grafos multietapa
  - Generador de árboles binarios.

  Deben desarrollarse correctamente, siguiendo las restricciones necesarias para cada uno e integrándose completamente en la aplicación (incluyendo la interfaz de usuario).

- **Objetivos secundarios:**
  1. **Actualización de la importación y exportación de las restricciones.**
     Dada la implementación de los nuevos generadores, es necesario también actualizar cómo se importan y exportan las restricciones referentes a cada uno, de manera que sean incluidas en AlgorEx.
  2. **Actualización del diálogo de los generadores de datos.**
     En las ventanas de selección de restricciones, el botón de “Guardar” aparece situado bajo la introducción de dichas restricciones, lo que en algunos casos hace que sea difícil para el usuario visualizar correctamente la ventana. Se propone cambiar dicho

botón de localización, junto a los de “Aceptar” y “Cancelar” para evitar esto. Además, el tamaño de la ventana se ha concebido de manera fija, en lugar de ajustarse automáticamente al contenido, lo que complica también la visualización de estas ventanas.

3. **Corrección en la generación de juegos de datos repetidos**.
   Con la versión actual de AlgorEx, cuando se generan juegos de datos repetidos, éstos son descartados y no se vuelven a generar unos que no lo sean, de manera que pueden generarse menos juegos de datos que los requeridos por el usuario.
   Habría que corregir esta anomalía, de modo que, aunque se mantenga la comprobación de no repetición de datos, se genere el número exacto de ellos que haya indicado el usuario.
4. **Solucionar problemas varios.**
   Principalmente, se indican dos problemas observados.
   - A la hora de generar matrices, la comprobación de dimensiones e intervalo de elementos a generar no funciona correctamente, lo que impide generarlas como debería.
   - Además, en el resumen numérico se está fallando a la hora de calcular el campo “Número de ejecuciones válidas del método” cuando se selecciona el criterio de optimalidad. Es necesario revisar el origen de este error y corregirlo.

3.2 Proceso de desarrollo

Una vez fijados los objetivos, el alumno se encargó de describir un proceso de desarrollo con la intención de realizar el TFG de manera organizada.

Se han intentado aplicar técnicas de las metodologías ágiles vistas por el alumno en sus años estudiando el Grado de Ingeniería Informática, así como en las prácticas externas, que se detallarán un poco más adelante.

El proceso consta de cuatro fases diferenciadas: documentación, desarrollo, revisión y cierre.

Fase de documentación

Como se ha mencionado en la especificación de los objetivos, el alumno tuvo que identificar las restricciones y datos de entrada de una serie de problemas de optimización, principalmente provistos por el tutor.

Mediante el estudio de una colección de problemas de optimización [2], junto a la consulta de varios libros, debieron anotarse las restricciones y datos de entrada de ellos, con el fin de crear un listado de problemas que no puedan generarse con la versión previa de AlgorEx. La bibliografía consultada, de la que se citarán el título y sus autores ha sido la siguiente (podrá consultarse toda su información en el apartado de Referencias):

- *Algorithms: Design Techniques and Analysis*. Alsuwaiyel, M. H. [3]
- *Introduction to algorithms*. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. [4]
- *Data structures and algorithms in Java*. Goodrich, M. T., Tamassia, R. [5]
- *Computer algorithms*. Horowitz, E., Shani, S., Rajasekeran, S. [6]

Tras ello, el tutor y el estudiante deben llegar a un acuerdo sobre los tipos de generadores y los cambios en la aplicación que deben desarrollarse.

Fase de desarrollo

Para esta fase, el alumno ha intentado aplicar ciertas técnicas pertenecientes a las metodologías ágiles. En este caso, la inspiración ha sido la metodología conocida como Scrum.

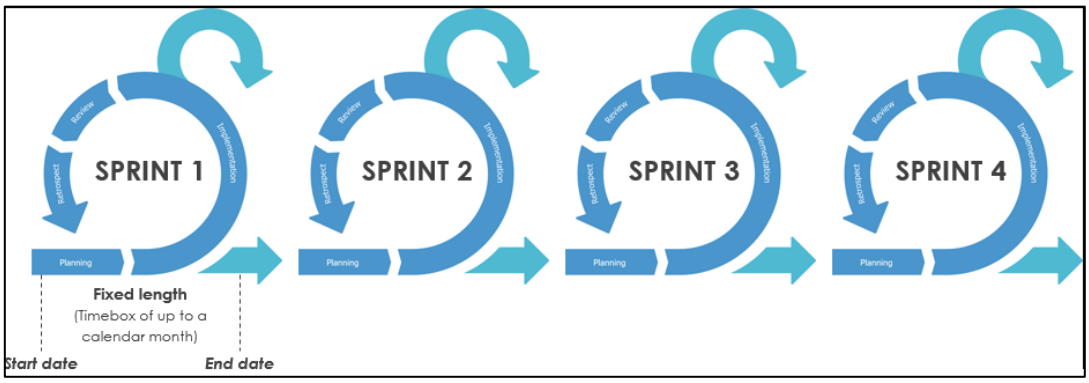


Ilustración 9. Ejemplo de estructura en Sprints en Scrum

En Scrum, las implementaciones en un proyecto de desarrollo Software se dividen en sprints: periodos de tiempo fijo y corto durante los cuales se desarrolla un incremento de trabajo.

Dado que el alumno no ha dispuesto de tiempo libre al completo para realizar el proyecto, estos sprints no han podido ser aplicados de manera totalmente fiel, por lo que se han tenido que adaptar a las características de este TFG. Siguiendo esto, tras el debido estudio inicial de la aplicación, el proceso de desarrollo ha quedado de la siguiente manera:

- Primer sprint.

  Las tareas elegidas han sido el cambio de localización del botón de "Guardar" en las ventanas de introducción de restricciones y la implementación del generador de permutaciones, así como la eliminación de la restricción a la hora de generar juegos de datos repetidos.

  Se procede a hacer diversas pruebas funcionales.

- Segundo sprint

  En este caso, la tarea elegida fue la implementación del generador de tareas.

  Sigue comprobándose el correcto funcionamiento mediante más pruebas funcionales.

  Contacto con el tutor para una revisión parcial del proyecto. Se detectan errores en la interfaz de usuario respectiva a los nuevos generadores implementados.

- Tercer sprint

  En él, se corrigen los errores comentados por el tutor para continuar con el avance del proyecto.

- Cuarto sprint

  Debido a la similitud en la interfaz de usuario entre los grafos aleatorios por probabilidad y por número de arcos, se decide incorporar la implementación de ambos a este sprint.

  Se comprueba su funcionamiento mediante más pruebas funcionales.

- Quinto sprint

  Tras estudiar la estrategia para implementar los grafos restantes, el alumno decide que en este sprint deben realizarse los generadores de grafos multietapa y de árboles binarios.

  Se prueba su funcionamiento.

- Sexto sprint

  Para este sprint se modifica la manera en que se exportan e importan las restricciones, debido a la implementación de los nuevos generadores de datos. Con la finalización de este sprint, la intención es que se habiliten dichos generadores a la hora de importar las restricciones de un fichero de texto, y se generen las ventanas y juegos de datos correspondientes.
  Se realizan pruebas funcionales a nivel de aplicación, ya que los objetivos principales han sido cumplidos por el alumno.

  Se envía la primera posible versión final de AlgorEx al tutor.

Para completar el desarrollo del código, además de utilizar las herramientas que se describirán más adelante, también ha sido necesario revisar la *documentación de Java* [7], sobre todo para entender ciertas funciones o métodos del código que ya existía previamente.

Fase de revisión

Después de que el alumno haya enviado la versión de AlgorEx que se acaba de mencionar al tutor con el objetivo de revisarla, se ponen en contacto durante varias semanas mediante conversaciones de correo electrónico donde van surgiendo ciertos errores con el programa que han ido corrigiéndose.

Las cuestiones que surgieron fueron sobre todo referentes a errores en la interfaz de usuario y a los valores cargados por defecto en las ventanas de restricciones.

Una vez arregladas y revisadas dichas cuestiones, el proceso de actualización de AlgorEx se dio por finalizado.

Fase de cierre

La fase final del proyecto está principalmente enfocada en el desarrollo del presente documento, conocido como Memoria del Proyecto.

Una vez dado el visto bueno a la Memoria, el alumno se encarga de subirla a la plataforma de Trabajos de Fin de Grado de la Universidad Rey Juan Carlos, preparándose así para su Defensa y posterior calificación.

3.3 Herramientas de desarrollo

Para la consecución de este Proyecto, el proceso de desarrollo ha sido llevado a cabo mediante el uso de las siguientes herramientas:

- **IntelliJ IDEA Community Edition**. Entorno de desarrollo en el que se ha implementado el código de AlgorEx.
- **Stack Overflow.** Foro web dedicado a programadores profesionales y aficionados. Se ha utilizado para resolver dudas de programación durante la fase de desarrollo.
- **Graphonline.ru**. Sitio web que permite dibujar grafos a partir de su matriz de adyacencia. Ha resultado muy útil para poder comprobar la generación de los diferentes tipos de grafos.
- **Word**. Se ha utilizado para la redacción de la Memoria, así como para llevar un registro de los avances producidos durante el proceso de desarrollo.
- **Adobe Acrobat Reader**. Para la lectura de diferentes documentos en formato PDF, tales como las colecciones de problemas de optimización revisadas en la fase de documentación.
- **Outlook**. Herramienta de correo electrónico utilizada como método de contacto entre el alumno y el profesor.
- **One Drive**. El servicio de almacenamiento en la nube de Microsoft se ha utilizado para guardar copias del proyecto, así como para compartir las diferentes versiones de AlgorEx con el tutor.

- **Microsoft Teams.** Su uso principal ha sido el de albergar las reuniones entre el alumno y el tutor. Además, se ha utilizado la herramienta “Tasks de Planner y To Do” perteneciente a esta herramienta para llevar a cabo la fase de desarrollo.
- **Paint**. Utilizada para la edición de algunas imágenes presentes en este documento.

4 Diseño

Para poder completar los objetivos propuestos, ha sido necesario llevar a cabo una labor de diseño de las soluciones previstas. En esta sección se explicará cómo funcionan los nuevos generadores de datos, su incorporación en la interfaz de usuario y la arquitectura de la herramienta.

4.1 Generadores de datos

En este apartado se procede a explicar el diseño de cada nuevo generador de datos. Posteriormente, en la sección de Implementación, se detalla su pseudocódigo.

Si bien es cierto que no se diseñaron todos al mismo tiempo (se han ido desarrollando uno por uno), la metodología a seguir ha sido la misma en todos los casos: identificación del tipo de dato que se va a generar y qué información se necesita (restricciones) para generarlo; codificación del método en un proyecto utilizado específicamente para pruebas; testeo del método e incorporación al programa, junto a la actualización de la interfaz de usuario.

La especificación de cada generador, explicada con más detalle, aparece en la siguiente página.

Generador	Datos de entrada	Especificación
Generador de permutaciones	Un vector de números enteros	Debe poder generarse una permutación de números enteros con tamaño y valor inicial indicados por el usuario.
Generador de tareas	Dos vectores de números enteros	Deben poder generarse dichos vectores específicamente para los problemas de selección de tareas. Esto es, que el primer vector tendrá los tiempos de inicio de cada tarea y el segundo los de finalización.
Generador de grafos por probabilidad	Matriz de números enteros	Se debe generar la matriz de adyacencia de un grafo G(n, p) siendo n el número de nodos y p la probabilidad de generación de cada arco.
Generador de grafos por número de arcos	Matriz de números enteros	Generación de la matriz de adyacencia de un grafo G(n, k) siendo n el número de nodos y k el número de arcos que deben generarse en el grafo.
Generador de grafos multietapa	Matriz de números enteros	Debe generarse la matriz de adyacencia de un grafo G(n, k) siendo n el número de nodos y k el número de etapas en las que se dividen los nodos.
Generador de árboles binarios	Matriz de números enteros	Generación de la matriz de adyacencia de un grafo que represente un árbol binario.

Tabla 1. Especificación de los generadores

4.2 Interfaz de usuario

Las actualizaciones referentes a la interfaz de usuario pasan por la creación de nuevas ventanas para la introducción de las restricciones de los diferentes generadores de datos, así como la nueva localización del botón "Guardar" y el redimensionamiento de dichas ventanas para que se ajuste al contenido.

En la versión previa de AlgorEx, el diálogo de selección de restricciones cuando la entrada son dos vectores de enteros era el siguiente:

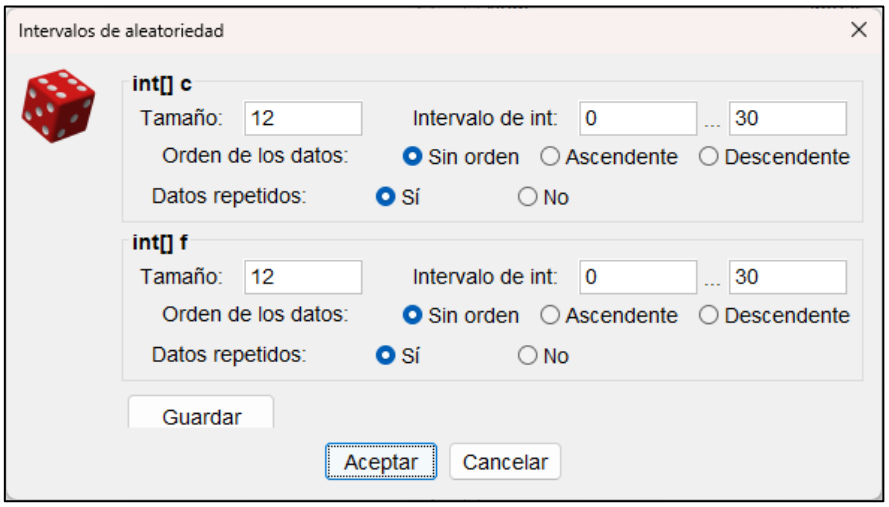


Ilustración 10. Ventana de restricciones de dos vectores en la versión previa de AlgorEx

Como se ha mencionado en el apartado 3.1, el botón de "Guardar" de la versión previa de AlgorEx no está bien localizado. Cuando hay pocas restricciones que introducir se muestra casi correctamente, aunque no llega a ser completamente visible.

Esto se debe principalmente a que la ventana fue concebida con un tamaño específico, en lugar de redimensionarse según la información contenida dentro. Además, se ha considerado que carece de sentido que el botón aparezca en ese espacio, cuando los botones de "Aceptar" y "Cancelar" tienen una localización diferente.

Ahora, se muestra el estado actual de las ventanas en AlgorEx, tras el desarrollo del alumno:

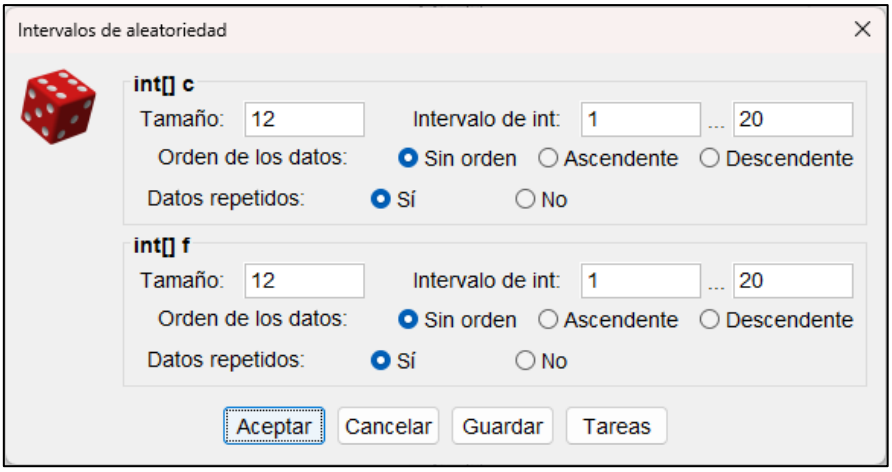


Ilustración 11. Ventana de restricciones de dos vectores en la versión final de AlgorEx

En comparación a la imagen anterior, en la

Ilustración 11 puede verse la nueva localización del botón "Guardar". Además, aparece uno nuevo botón llamado "Tareas", que hace referencia al nuevo generador de datos. Con esto, se presupone la intención del usuario de querer probar un algoritmo de selección de tareas, pues los datos de entrada del algoritmo son dos vectores. Esto no tiene por qué ser así (podría querer probar un algoritmo de ordenación y mezcla de vectores, por ejemplo), por lo que se ha optado por la opción del botón para que la decisión recaiga en el propio usuario.

Tras pulsar en el botón "Tareas", se abre la ventana de selección de restricciones referentes a su generador.

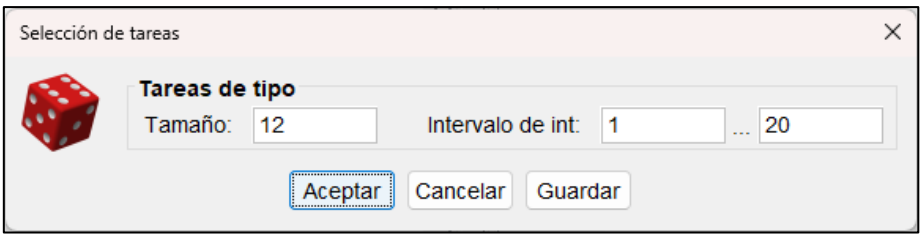


Ilustración 12. Ventana de restricciones del generador de Tareas

Los cambios que se realicen en la ventana principal se verán reflejados en la ventana de generación de tareas (y viceversa) y, al hacer clic en "Cancelar", se regresa a la ventana original, aportando una navegación sencilla y con sentido a la experiencia del usuario.

A continuación, se muestra la variación de la selección de restricciones en el caso de las matrices de enteros:

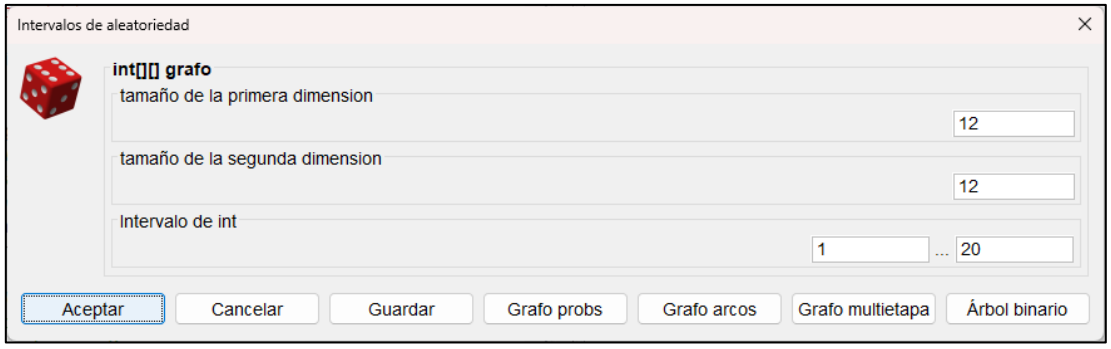


Ilustración 13. Ventana de restricciones de matrices

A diferencia de la anterior versión de AlgorEx, cuando se detecta que un dato de entrada es una matriz de enteros, se habilitan los cuatro botones visibles en la Ilustración 13, acorde a los nuevos generadores implementados.

En este caso, la información que aparece en la ventana principal (como el tamaño de las dimensiones de la matriz) no es trasladada a la específica de cada generador, pues los datos de entrada de cada grafo difieren. En cambio, la navegación funciona igual que en las ventanas comentadas anteriormente: al pulsar la opción “Cancelar” en la ventana de alguno de los grafos, se vuelve a la ventana principal, evitando que el usuario deba reiniciar la interacción desde cero.

Tras ello, se procede a mostrar un ejemplo para cada generador de gafos (a excepción del grafo aleatorio por número de arcos, pues su ventana prácticamente no difiere de la del grafo por probabilidad):

Generador de grafos por probabilidad

Este tipo de grafo se genera recibiendo el número de nodos que contiene y la probabilidad de que existan arcos entre ellos (a menor probabilidad, mayor dispersión). Además, se piden al usuario otros datos que ayudan a especificar aún más el grafo que quiere generar: su dirección, valuación (cada arco tendrá un valor comprendido dentro del rango que indique el usuario o sus valores serán 0 o 1) y si es reflexivo o no (sólo cuando no sea valuado, si se selecciona permite generar arcos desde un nodo hacia sí mismo).

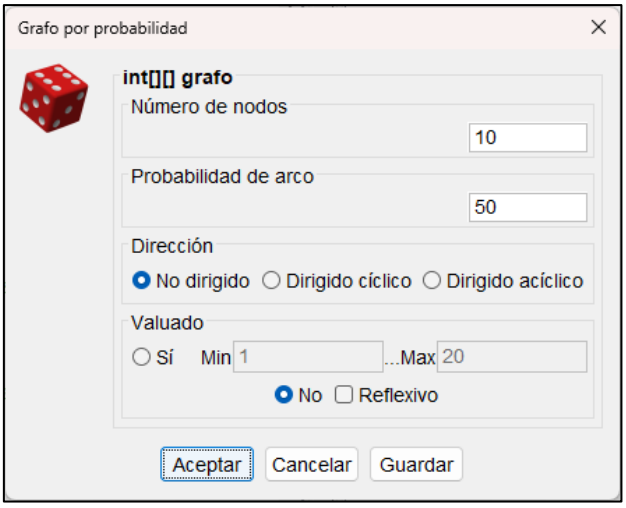


Ilustración 14. Ventana de restricciones de grafo por probabilidad

Generador de grafos multietapa

En este caso, el número de campos a introducir es menor: el número de nodos del grafo, las etapas en las que se tienen que dividir y su valuación (en caso de ser valuado, se deben indicar el valor mínimo y máximo).

Ilustración 15. Ventana de restricciones de grafo multietapa

Generador de árboles binarios

En este último generador, solamente es necesario introducir el número de nodos del árbol. En la sección de Implementación se explicará cómo funciona de manera detallada.

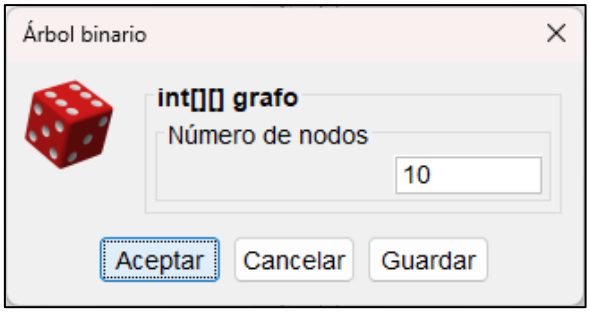


Ilustración 16. Ventana de restricciones de árbol binario

4.3 Importación y exportación de restricciones

Como se comentó en la especificación de objetivos, debido a la introducción de los nuevos generadores de datos también ha sido necesario diseñar los cambios que han tenido lugar en la importación y exportación de sus restricciones, logrando así no disminuir la coherencia de AlgorEx.

Las restricciones son los datos de entrada de cada generador. Es otras palabras, son los parámetros que se seleccionan en las ventanas que se han visto anteriormente, y que deben ser especificados por el usuario según el tipo de algoritmo al que se enfrente.

Hasta ahora, cuando el usuario escoge la opción de exportar las restricciones de un generador, se está creando (o actualizando) un fichero en formato *CNSTR* en el que se imprimen el tipo de dato que debe generarse y los parámetros que ha seleccionado.

En la nueva versión desarrollada por el alumno, se ha actualizado esta función de manera que, si se quieren exportar las restricciones desde el diálogo de uno de los nuevos generadores de datos, se indique cuál es dicho generador, de modo que, al importarse, la herramienta sepa qué ventanas mostrar y qué generador utilizar. Además, en los generadores de grafos se han modificado los parámetros que se guardan, pues no coinciden con el generador de matrices que existía anteriormente.

Veamos unos ejemplos de esta funcionalidad:

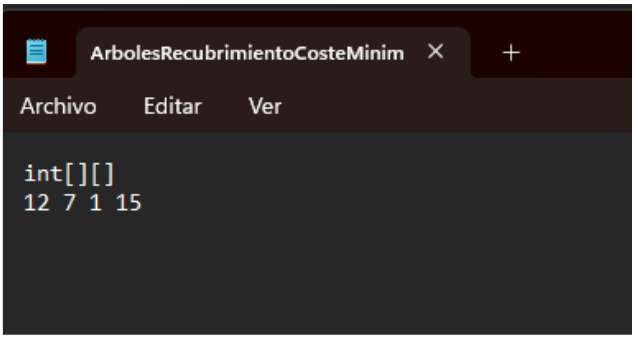


Ilustración 17. Restricciones de una matriz en la versión previa de AlgorEx

Lo que se ve en la Ilustración 17 es precisamente lo que se ha comentado sobre la importación de restricciones que ya existía en AlgorEx. En este ejemplo, se han guardado las restricciones de una matriz de números enteros. Los números de la segunda fila hacen referencia a sus dimensiones y rangos de valores, como puede verse al seleccionar su importación:

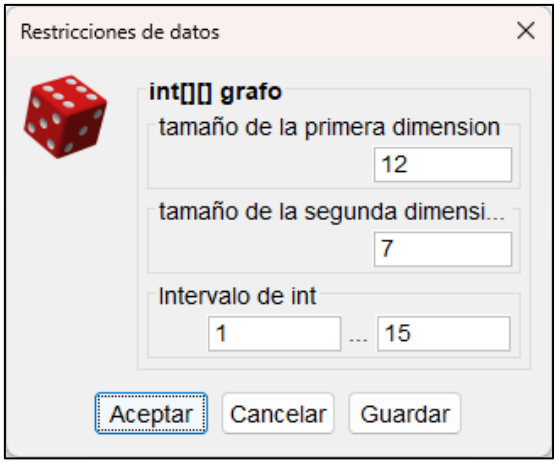


Ilustración 18. Restricciones importadas de una matriz en la versión previa de AlgorEx

Ahora, veamos los cambios producidos en el fichero cuando se han guardado las restricciones de un generador de grafos por probabilidad de arco:

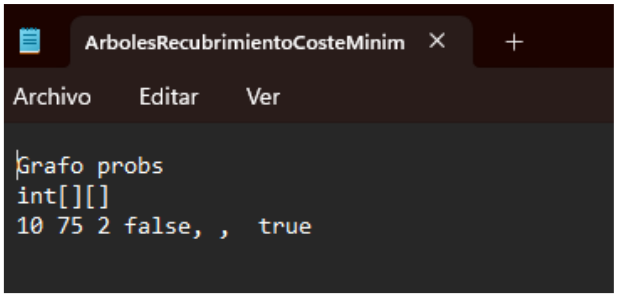


Ilustración 19. Restricciones de un grafo aleatorio

Tal y como se ha mencionado anteriormente, en la primera fila se describe cuál es el generador al que hace referencia el fichero. En cuanto a los datos, es clara la diferencia respecto a los vistos en la Ilustración 17. De esta manera, cuando se importen estas restricciones, se abrirá la ventana de este generador:

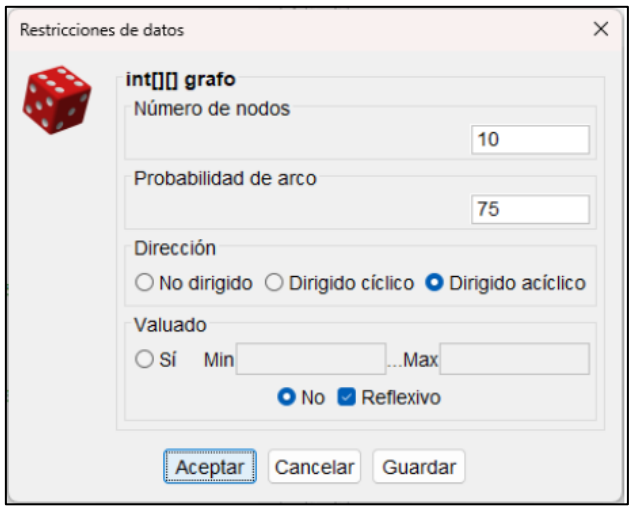


Ilustración 20. Restricciones importadas de un grafo aleatorio

4.4 Arquitectura

El proyecto está constituido por 45 clases Java, así como por varios archivos HTML y las imágenes utilizadas en él. Para otorgar una visión general de la arquitectura del proyecto, se muestra un esquema simplificado de las clases Java de AlgorEx:

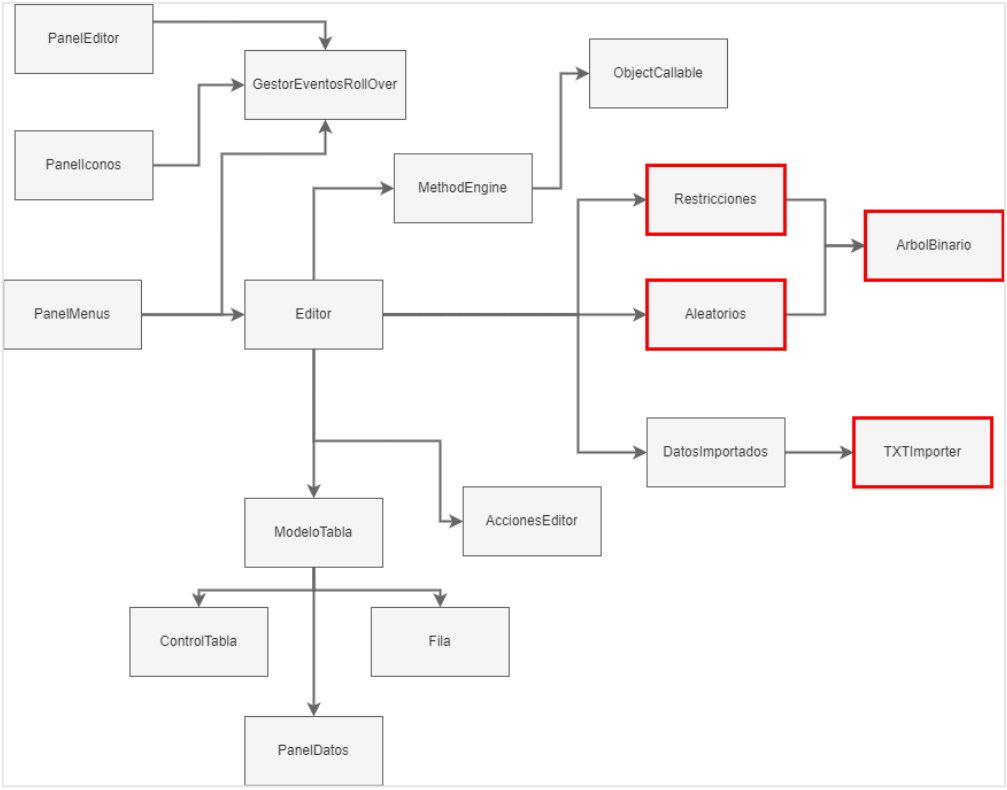


Ilustración 21. Esquema resumido de la arquitectura del proyecto

Resaltadas en rojo se encuentran las clases con las que ha trabajado el alumno. A continuación, se muestra una breve explicación del código modificado/añadido en ellas:

Clase	Resumen
Aleatorios	Es la clase que más cambios ha sufrido. En ella se han introducido los generadores de datos y se ha actualizado la interfaz de usuario en la manera vista anteriormente, además de haber realizado varias refactorizaciones.
ArbolBinario	Ha sido creada exclusivamente para el generador de árboles binarios. Solamente es llamada desde las clases Aleatorios y Restricciones.
Restricciones	Desde ella se controla cómo se comporta AlgorEx cuando se importan las restricciones desde un fichero de texto. Los cambios realizados son muy similares a los de la clase Aleatorios.

TXTImporter	Es utilizada para controlar la importación de restricciones desde los ficheros de texto. Se ha tenido que modificar de acuerdo con las nuevas opciones añadidas por la implementación de los generadores.

Tabla 2. Clases modificadas por el alumno

5 Implementación

Los cambios mencionados en la Especificación de objetivos y en la sección de Diseño han tenido que realizarse a través de la programación en lenguaje Java. En esta sección se procede a explicar cómo se han desarrollado las nuevas actualizaciones.

5.1 Código de los generadores

Una vez vista la especificación de cada nuevo generador de datos, se procede a describir cómo debe funcionar cada uno y cuál es el código que se ha implementado para cumplirlo.

5.1.1 Generador de permutaciones

Supongamos que una permutación se representa mediante un vector de longitud *n* que contiene los elementos 0..*n*-1, entonces debe poder generarse un vector de enteros con la longitud y valor inicial que indique el usuario (los elementos deben estar dispuestos aleatoriamente y, por definición, sin repetición).

Código 1. Generador de permutaciones

```
Función PermutacionAleatorio(tam, low)
    perm <- array de tamaño tam
    Para i desde 0 hasta tam-1 hacer
        perm[i] <- low + i
    Fin Para
    Para i desde tam-1 hasta 1 hacer
        alea <- entero aleatorio entre 0 e i
        temp <- perm[i]
        perm[i] <- perm[alea]
        perm[alea] <- temp
    Fin Para
```

Este método genera una permutación aleatoria de números enteros. Toma dos parámetros como entrada: **'tam'** (el tamaño de la permutación) y **'low'** (el valor más bajo para los números de la permutación).

El método crea un array **'perm'** de tamaño **'tam'** para almacenar la permutación. Luego, inicializa **'perm'** con los valores consecutivos a partir de **'low'**. Después, realiza un bucle desde '*tam*-1' hasta 1. En cada iteración, genera un número aleatorio en el rango de 0 a **'i'**. Luego, intercambia el valor de **'perm[i]'** con el valor de **'perm[alea]'**. Esto se repite hasta que todas las posiciones del array hayan sido intercambiadas al menos una vez.

Finalmente, el método devuelve el array **'perm'**, que contiene la permutación aleatoria generada.

5.1.2 Generador de tareas

Se dispone de dos vectores, uno *cs* para los instantes de inicio y otro *fs* para los instantes de fin, de forma que la tarea *i* tiene como instantes asociados el inicio *cs[i]* y el final *fs[i]*. Se generan sucesivamente dos pares de números, guardando el menor en *cs[i]* y el mayor en *fs[i]*.

Código 2. Generador de tareas

```
Función TareasAleatorio(tam, low, high)
    matrix <- matriz de tamaño 2 x tam
    Para i desde 0 hasta tam-1 hacer
        valorBajo <- 0
        valorAlto <- 0
        Mientras valorBajo sea igual a valorAlto hacer
            valorBajo <- entero aleatorio entre low y high
            valorAlto <- entero aleatorio entre low y high
        Fin Mientras
        Si valorBajo > valorAlto entonces
```

```
            aux <- valorBajo
            valorBajo <- valorAlto
            valorAlto <- aux
        Fin Si
        matrix[0][i] <- valorBajo
        matrix[1][i] <- valorAlto
    Fin Para
    devolver matrix
Fin Función
```

Este método genera una matriz de tareas aleatorias con dos filas. Toma tres parámetros de entrada: **'tam'** (el número de tareas), **'low'** (el valor más bajo para las tareas) y **'high'** (el valor más alto para las tareas).

La función crea una matriz llamada **'matrix'** de tamaño 2 x **'tam'** para almacenar las tareas. Luego, realiza un bucle desde 0 hasta **'tam-1'**. En cada iteración, se generan dos valores aleatorios: **'valorBajo'** y **'valorAlto'**. El bucle se repite hasta que **'valorBajo'** sea diferente de **'valorAlto'**, asegurando que los valores generados sean distintos entre sí.

Después, se verifica si **'valorBajo'** es mayor que **'valorAlto'**. En caso afirmativo, se intercambian los valores utilizando una variable auxiliar (asegurando siempre que **'valorBajo'** se almacenará en la fila de los tiempos de inicio y **'valorAlto'** en la fila de los de finalización). Luego, se asignan los valores de **'valorBajo'** y **'valorAlto'** a las posiciones correspondientes en la matriz.

Finalmente, el método devuelve la matriz que contiene las tareas generadas aleatoriamente, donde la primera fila representa los tiempos de inicio de las tareas y la segunda fila representa los tiempos de finalización de cada una.

5.1.3 Generador de grafos por probabilidad

Supongamos que un grafo se representa mediante una matriz de adyacencia *g*[][] con las siguientes características:

- Si el grafo es valuado, cada arco *g*[*i*][*j*] puede contener un número dentro del rango de valores indicado. Supondremos que nunca hay un arco *g*[*i*][*i*] o, dicho de otra forma, estos arcos siempre valen 0.
- Si el grafo es no valuado, cada arco *g*[*i*][*j*] vale 1 cuando los nodos *i* y *j* están unidos mediante un arco y vale 0 en caso contrario. Puede haber arcos *g*[*i*][*i*].

Para el grafo aleatorio G(*n*,*p*) con *n* nodos y probabilidad de arco *p*, cada arco se genera independientemente con probabilidad *p*. En concreto, con probabilidad 0 nunca se genera el arco y con probabilidad 1 se genera siempre.

Se considera sucesivamente cada arco (*i*,*j*), insertándolo en el grafo con probabilidad *p*.

Código 3. Grafo por probabilidad

```
Función GrafoProbsAleatorio(nodos, p, direccion, valuado, min, max, reflexivo)
  grafo <- matriz de tamaño nodos x nodos
  Si valuado es verdadero entonces
    Para i desde 0 hasta nodos-1 hacer
      Para j desde 0 hasta nodos-1 hacer
        grafo[i][j] <- valor máximo de un short
      Fin Para
    Fin Para
  Fin Si
  prob <- p / 100
  Si direccion es igual a 0 entonces
    Para i desde 0 hasta nodos-1 hacer
```

```
Si valuado es falso entonces
    Si reflexivo es verdadero entonces
        Para j desde i hasta nodos-1 hacer
            Si prob > número aleatorio entre 0 y 1 entonces
                grafo[i][j] <- 1
            Fin Si
            grafo[j][i] <- grafo[i][j]
        Fin Para
    Sino
        Para j desde i hasta nodos-1 hacer
            Si prob > número aleatorio entre 0 y 1 y i no es igual a j entonces
                grafo[i][j] <- 1
            Fin Si
            grafo[j][i] <- grafo[i][j]
        Fin Para
    Fin Si
Sino
    Para j desde i hasta nodos-1 hacer
        Si prob > número aleatorio entre 0 y 1 y i no es igual a j entonces
            grafo[i][j] <- número aleatorio entre min y max (incluyendo ambos)
        Fin Si
        grafo[j][i] <- grafo[i][j]
    Fin Para
```

```
        Fin Si
    Fin Para
Sino si direccion es igual a 1 entonces
    Para i desde 0 hasta nodos-1 hacer
        Para j desde 0 hasta nodos-1 hacer
            Si prob > número aleatorio entre 0 y 1 entonces
                Si valuado es falso entonces
                    Si reflexivo es verdadero entonces
                        grafo[i][j] <- 1
                    Sino si reflexivo es falso y i no es igual a j entonces
                        grafo[i][j] <- 1
                    Fin Si
                Sino si i no es igual a j entonces
                    grafo[i][j] <- número aleatorio entre min y max
                Fin Si
            Fin Si
        Fin Para
    Fin Para
Sino
    Para i desde 0 hasta nodos-1 hacer
        Para j desde i hasta nodos-1 hacer
            Si prob > número aleatorio entre 0 y 1 entonces
                Si valuado es verdadero entonces
```

```
                grafo[i][j] <- número aleatorio entre min y max
            Sino
                Si reflexivo es verdadero entonces
                    grafo[i][j] <- 1
                Sino si reflexivo es falso y i no es igual a j entonces
                    grafo[i][j] <- 1
                Fin Si
            Fin Si
        Fin Si
    Fin Para
  Fin Para
 Fin Si
 devolver grafo
Fin Función
```

La función '**GrafoProbsAleatorio'** genera un grafo aleatorio con diferentes propiedades según los parámetros proporcionados. Toma varios parámetros como entrada: '**nodos'** (el número de nodos en el grafo), '**p'** (probabilidad en porcentaje de que exista un arco entre dos nodos), '**direccion'** (indicador de la dirección del grafo: 0 para no dirigido, 1 para dirigido cíclico, y 2 para dirigido acíclico), '**valuado'** (una variable booleana que determina si el grafo es valuado o no), '**min'** y '**max'** (los valores mínimo y máximo para los arcos valuados) y '**reflexivo'** (un indicador booleano que determina si el grafo puede tener arcos reflexivos).

El método crea una matriz llamada grafo de tamaño *nodos* x *nodos* para almacenar la estructura del grafo.

Si valuado es verdadero, se inicializa cada elemento de grafo con el valor máximo de un short. Esto se hace para indicar que no hay conexión entre los nodos en el grafo valuado cuando no se generan arcos.

Luego, se calcula '**prob'** dividiendo '**p'** entre 100.

A continuación, se aplican diferentes estrategias dependiendo del valor de '**direccion'**. Si es 0, significa que el grafo es no dirigido. Se generan los arcos según la probabilidad '**prob'** y se establecen los valores en la matriz de adyacencia. Si '**valuado'** es falso, se considera la opción '**reflexivo'** para permitir arcos reflexivos. Si '**valuado'** es verdadero, se genera un valor aleatorio entre '**min'** y '**max'** para los arcos. Después de asignar los valores de los arcos, se copian los valores simétricos en la matriz de adyacencia para mantener la propiedad de grafo no dirigido.

Si '**direccion'** es igual a 1, significa que el grafo es dirigido cíclico. Se generan los arcos según la probabilidad '**prob'** y se establecen los valores en la matriz grafo. Al igual que antes, si '**valuado'** es falso, se considera la opción '**reflexivo'** para permitir arcos reflexivos. Si valuado es verdadero, se genera un valor aleatorio entre '**min'** y '**max'** para los arcos.

Si '**direccion'** no es ni 0 ni 1, significa que el grafo es dirigido acíclico. Se generan los arcos y se asignan los valores en la matriz grafo de manera similar al caso anterior.

Finalmente, se devuelve la matriz de adyacencia que representa el grafo aleatorio generado.

5.1.4 Grafo por número de arcos

Con el grafo aleatorio G(*n*,*k*) con n nodos y un número k de arcos, se permite controlar si el grafo es denso (máximo de n^2-*n* nodos) o disperso (mínimo de 0 arcos). A cada arco se le asocia un número 0..*m*-1, donde $m = (n^2-n)/2$. Se generan k elementos no repetidos de 0..*m*-1, poniendo a 1 el arco correspondiente de la matriz de adyacencia.

Código 4. Grafo por número de arcos

```
Función GrafoArcosAleatorio(numNodos, numArcos, tipoGrafo, esValuado, min, max, esReflexivo)
    matrizAdyacencia <- matriz de tamaño numNodos x numNodos
```

```
arcosGenerados <- 0
Si esValuado es verdadero entonces
    Para i desde 0 hasta tamaño de matrizAdyacencia - 1 hacer
        Para j desde 0 hasta tamaño de matrizAdyacencia[i] - 1 hacer
            matrizAdyacencia[i][j] <- valor máximo de un short
        Fin Para
    Fin Para
Fin Si
Mientras arcosGenerados sea menor que numArcos hacer
    nodoOrigen <- número aleatorio entre 0 y numNodos-1
    nodoDestino <- número aleatorio entre 0 y numNodos-1
    Si matrizAdyacencia[nodoOrigen][nodoDestino] es igual a 0 o
      matrizAdyacencia[nodoOrigen][nodoDestino] es igual al valor máximo de un short entonces
        Si esReflexivo es verdadero o (esReflexivo es falso y nodoOrigen no es igual a nodoDestino) entonces
            Si tipoGrafo es igual a 0 entonces
                Si esValuado es verdadero entonces
                    matrizAdyacencia[nodoOrigen][nodoDestino] <- número aleatorio entre min y max
                    matrizAdyacencia[nodoDestino][nodoOrigen] <- matrizAdyacencia[nodoOrigen][nodoDestino]
                Sino
                    matrizAdyacencia[nodoOrigen][nodoDestino] <- 1
                    matrizAdyacencia[nodoDestino][nodoOrigen] <- 1
                Fin Si
            Sino si tipoGrafo es igual a 1 entonces
```

```
        Si esValuado es verdadero entonces
            matrizAdyacencia[nodoOrigen][nodoDestino] <- número aleatorio entre min y max
        Sino
            matrizAdyacencia[nodoOrigen][nodoDestino] <- 1
        Fin Si
    Sino si tipoGrafo es igual a 2 entonces
        Si nodoDestino > nodoOrigen entonces
            Si esValuado es verdadero entonces
                matrizAdyacencia[nodoOrigen][nodoDestino] <- número aleatorio entre min y max
            Sino
                matrizAdyacencia[nodoOrigen][nodoDestino] <- 1
            Fin Si
        Fin Si
    Fin Si
    arcosGenerados <- arcosGenerados + 1
  Fin Si
 Fin Si
Fin Mientras
devolver matrizAdyacencia
Fin Función
```

La función '**GrafoArcosAleatorio'** genera un grafo aleatorio con un número específico de arcos según los parámetros proporcionados. Toma varios parámetros como entrada: '**numNodos'** (el número de nodos en el grafo), '**numArcos'** (el número de arcos a generar), '**tipoGrafo'** (el tipo de grafo: 0 para no dirigido, 1 para dirigido cíclico y 2 para dirigido acíclico), '**esValuado'** (un indicador booleano que determina si el grafo es valuado o no), '**min'**

y '**max**' (los valores mínimo y máximo para los arcos valuados), y '**esReflexivo'** (un indicador booleano que determina si el grafo puede tener arcos reflexivos).

La función crea una matriz de adyacencia para representar la estructura del grafo. Si '**esValuado'** es verdadero, inicializa cada elemento de la matriz con el valor máximo de un short para indicar que no hay conexión entre los nodos cuando no se generan arcos.

Luego, en un bucle '**while'**, se generan los arcos hasta que se alcance el número deseado de '**numArcos'**. En cada iteración, se eligen aleatoriamente un '**nodoOrigen'** y un '**nodoDestino'**. Si no hay un arco existente entre estos nodos en la matriz, se procede a verificar las condiciones para generar un nuevo arco.

Si es reflexivo o si no es reflexivo y los nodos origen y destino no son el mismo, se generan los arcos según el tipo de grafo especificado.

Si '**tipoGrafo'** es igual a 0, se trata de un grafo no dirigido y se generan arcos en ambas direcciones. Si '**esValuado'** es verdadero, se asigna un valor aleatorio entre '**min'** y '**max**' a los arcos. Si '**esValuado'** es falso, se establece el valor del arco en 1.

Si '**tipoGrafo'** es igual a 1, se trata de un grafo dirigido cíclico y se genera un arco desde el nodo origen hacia el nodo destino. Nuevamente, si es valuado, se asigna un valor aleatorio entre '**min'** y '**max**' al arco.

Si '**tipoGrafo'** es igual a 2, se trata de un grafo dirigido acíclico. Solo se generan arcos en la mitad superior de la matriz para evitar ciclos. Si es valuado, se asigna un valor aleatorio entre '**min'** y '**max**' al arco.

Después de generar un nuevo arco válido, se incrementa el contador.

Una vez que se generan todos los arcos requeridos, se devuelve la matriz de adyacencia que representa el grafo aleatorio generado.

5.1.5 Grafo multietapa

El grafo multietapa G(*n*,*k*) permite generar un grafo G con *n* nodos y *k* etapas. Un grafo multietapa es un grafo acíclico y valuado tal que existe un único nodo origen, un único nodo

destino y todos los nodos se distribuyen en conjuntos (etapas) tales que a cada nodo de la etapa *s*, sólo le llegan arcos desde nodos de la etapa *s*-1 y desde él sólo surgen arcos a nodos de la etapa *s*+1.

Código 5. Grafo multietapa

```
Función GrafoMultietapaAleatorio(numNodos, numEtapas, valuado, min, max)
    matriz <- matriz de tamaño numNodos x numNodos
    etapas <- Lista de Listas vacías
    nodos <- PermutacionAleatorio(numNodos-2, 2)
    Para i desde 0 hasta numEtapas-1 hacer
        etapa <- Lista vacía
        etapas.agregar(etapa)
    Fin Para
    origen <- 0
    Para i desde 0 hasta tamaño de nodos-1 hacer
        Si origen es igual a numEtapas entonces
            origen <- 0
        Fin Si
        etapas[origen].agregar(nodos[i])
        origen <- origen + 1
    Fin Para
    Para i desde 0 hasta tamaño de etapas[0]-1 hacer
        nodoDestino <- etapas[0][i] - 1
        Si valuado es verdadero entonces
            Random rand <- nuevo objeto Random
```

```
        matriz[0][nodoDestino] <- número aleatorio entre min y max
    Sino
        matriz[0][nodoDestino] <- 1
    Fin Si
Fin Para
Para i desde 0 hasta tamaño de etapas - 2 hacer
    Para j desde 0 hasta tamaño de etapas[i]-1 hacer
        nodoOrigen <- etapas[i][j] - 1
        arcosAGenerar <- número aleatorio entre 0 y tamaño de etapas
        arcos <- 0
        Mientras arcos sea menor que arcosAGenerar hacer
            nodoDestino <- etapas[i+1][número aleatorio entre 0 y tamaño de etapas]
            Si matriz[nodoOrigen][nodoDestino] es igual a 0 entonces
                Si valuado es verdadero entonces
                    Random rand <- nuevo objeto Random
                    matriz[nodoOrigen][nodoDestino] <- número aleatorio entre min y max
                Sino
                    matriz[nodoOrigen][nodoDestino] <- 1
                Fin Si
                arcos <- arcos + 1
            Fin Si
        Fin Mientras
    Fin Para
```

```
    Fin Para
    Para i desde 0 hasta tamaño de etapas[numEtapas - 1] - 1 hacer
        nodoOrigen <- etapas[numEtapas - 1][i] - 1
        Si nodoOrigen no es igual a -1 entonces
            Si valuado es verdadero entonces
                Random rand <- nuevo objeto Random
                matriz[nodoOrigen][numNodos - 1] <- número aleatorio entre min y max
            Sino
                matriz[nodoOrigen][numNodos - 1] <- 1
            Fin Si
        Fin Si
    Fin Para
    devolver matriz
Fin Función
```

El método genera un grafo multietapa aleatorio con un número específico de nodos y etapas según los parámetros proporcionados. Toma varios parámetros como entrada: '**numNodos'** (el número de nodos en el grafo), '**numEtapas'** (el número de etapas en el grafo), '**valuado'** (un indicador booleano que determina si el grafo es valuado o no), '**min'** y '**max'** (los valores mínimo y máximo para los arcos valuados).

La función crea una matriz para representar la estructura del grafo. También crea una lista de etapas que contiene listas vacías para representar las etapas del grafo.

A continuación, se genera una permutación aleatoria de **'numNodos - 2 nodos'** utilizando la función '**PermutacionAleatorio'**. Los dos nodos restantes se utilizarán como el nodo de origen y el nodo de destino del grafo.

Después de inicializar las etapas y generar la permutación aleatoria, se procede a asignar los nodos a las etapas correspondientes. Se itera sobre los nodos y se asignan secuencialmente a las etapas.

Luego, se generan los arcos entre el nodo de origen y los de la primera etapa. Se itera sobre los nodos en la primera etapa y se genera un arco desde el nodo de origen hasta cada nodo de la primera etapa. Si el grafo es valuado, se asigna un valor aleatorio entre '**min**' y '**max**' al arco. De lo contrario, se establece el valor del arco en 1.

A continuación, se generan los arcos entre las etapas restantes del grafo. Se itera sobre las etapas, comenzando desde la segunda y hasta la penúltima. Luego, se itera sobre los nodos en cada etapa y se generan arcos hacia nodos aleatorios en la siguiente. El número de arcos a generar se elige aleatoriamente entre 0 y el tamaño de la siguiente etapa. Se generan arcos hasta que se alcance la cantidad deseada. Si el arco generado no existe previamente, se asigna un valor aleatorio dentro del rango si el grafo es valuado, o se establece el valor del arco en 1 si no es valuado.

Finalmente, se generan los arcos desde la última etapa hasta el nodo final del grafo. Se itera sobre los nodos en la última etapa y se generan arcos hacia el nodo final. Se establece el valor del arco según corresponda.

Finalmente, se devuelve la matriz de adyacencia.

5.1.6 Árbol binario

El caso de este generador es especial, pues se decidió crear una clase que contenga la estructura de un árbol binario, de manera que pueda usarse en las clases que generen los juegos de datos. Esta clase solo contiene tres elementos: la raíz del árbol, su hijo izquierdo y su hijo derecho. Además del constructor y de los Getters y Setters propios de la clase, se han implementado otros dos métodos adicionales (la mínima cantidad de código necesaria para la versión actual de AlgorEx):

- Método insertar. Recibe como parámetro un nodo y, siguiendo las reglas de inserción de los árboles binarios (si es menor que la raíz se introduce en el hijo izquierdo y si no, en el hijo derecho), se añade al árbol.

- Método matrizAdyacencia. Recibe la propia estructura matricial. Mediante llamadas recursivas, se encarga de poner a 1 los arcos que unan la raíz del árbol actual con sus dos hijos.

De esta manera, el pseudocódigo del generador de árboles binarios resulta así:

Código 6. Árbol binario

```
Función ArboBinarioAleatorio(numNodos)
    permutacion <- PermutacionAleatorio(numNodos, 1)
    matriz <- matriz de tamaño numNodos x numNodos
    arbol <- nuevo objeto ArbolBinario con el primer elemento de la permutación
    Para i desde 1 hasta tamaño de permutacion - 1 hacer
        arbol.insertar(permutacion[i])
    Fin Para
    arbol.matrizAdyacencia(matriz)
    devolver matriz
Fin Función
```

La función recibe como entrada el número de nodos y utiliza el generador de permutaciones para generar una permutación aleatoria de con dichos nodos. Esta permutación se almacena en un array de enteros.

Luego, se crea una matriz que se utilizará para almacenar la matriz de adyacencia del árbol binario.

Se crea un objeto '**arbol'** de la clase '**ArbolBinario'** y se inicializa con el primer elemento de la permutación. Luego, se recorre la permutación desde el segundo elemento hasta el último e inserta cada elemento en el árbol binario utilizando el método insertar del objeto '**arbol'**.

Una vez que se han insertado todos los elementos en el árbol binario, se llama al método '**matrizAdyacencia'** del objeto '**arbol'**.

Finalmente, se devuelve la matriz de adyacencia del árbol binario generado aleatoriamente.

5.2 Restricciones

En la clase Restricciones.java se controla la importación de restricciones desde un fichero de texto. Sigue una estructura similar a la de la clase **Aleatorios.java**, por lo que algunos de los cambios que se mencionan en el siguiente apartado también se han llevado a cabo aquí.

Para la importación y exportación de las restricciones, se han tenido que realizar cambios en varias clases. En **Aleatorios.java**, en el método **GuardarRestricciones**, se tiene en cuenta si las restricciones que se están exportando pertenecen a alguno de los nuevos generadores. En ese caso, la primera línea del fichero de texto contendrá la palabra "Tareas" o "Árbol", por ejemplo, para indicar que se debe utilizar ese generador en específico. A continuación, se escribirían el tipo de dato al que hace referencia y los valores de las restricciones que el usuario haya indicado. En caso de que no se haya utilizado ninguno de ellos, se sigue llamando al método **guardarRestriccionesTxt** de la clase **TXTImporter**.

El otro cambio significativo se ha producido en la clase **Restricciones.java**. Cuando se lee el fichero de texto, se detecta si en la primera línea aparece alguna de las palabras que se han utilizado para exportar las restricciones de los nuevos generadores de datos. En ese caso, dependiendo de cuál sea, se le asigna su valor correspondiente a una variable que controla el generador adecuado, y se crea la ventana necesaria para la introducción de sus restricciones.

5.3 Otras cuestiones

En este apartado están presentes los cambios "menores" que se han realizado en relación al código de AlgorEx.

En la clase Aleatorios.java se han realizado la mayoría de las modificaciones, pues es en ella donde se llama a los generadores y se obtienen los datos que se utilizarán para las ejecuciones de los distintos algoritmos. Es necesario mencionar el método **ActualizarTopes**, a través del cual, aparte de comprobar los datos pedidos en los algoritmos, se generan la ventana principal de selección de restricciones (

Ilustración 11) y se controlan las opciones de dicha ventana. Empezando por las opciones, se ha visto que en la versión previa de AlgorEx solo existían los botones "Aceptar", "Cancelar" y "Guardar", pero se han añadido los que direccionan a la ventana de cada generador, suponiendo un gran aumento de código, pues es necesario realizar un proceso similar al de la ventana principal con cada uno de ellos. Para reducir el tamaño del código y evitar que haya excesivas repeticiones de un mismo fragmento, se ha decidido extraer parte de él en métodos auxiliares, para ser sustituidos por llamadas a ellos. Dicho esto, se han creado los métodos *GenerarPanel*, *AsignarTopes* y *GuardarRestricciones*, cuya función es descrita brevemente a continuación:

- **GenerarPanel.** Antiguamente, se trataba del código que rellenaba la ventana de restricciones con las opciones referentes a cada uno de los datos de entrada de los algoritmos de la clase cargada en AlgorEx. Ahora, se utiliza también como método adicional para las ventanas específicas de los generadores, puesto que mediante este método pueden crearse los elementos de la interfaz de usuario de los tipos de datos que sean diferentes a los de los propios generadores (un ejemplo sería al recibir como entrada una matriz de enteros y varios datos de entrada adicionales: se sigue pudiendo generar un grafo además de estos datos).
- **AsignarTopes.** Mediante este método se cargan los valores introducidos en la ventana de selección de restricciones a unas variables globales que son utilizadas para realizar las llamadas a los generadores, guardar las restricciones en un fichero de texto…
- **GuardarRestricciones.** Se ha extraído el código referente al antiguo botón de "Guardar", para no repetirlo en todas las ventanas que se creen desde *ActualizarTopes*.

Aparte de esto, para la traducción de la interfaz de usuario se han añadido varias cadenas de caracteres a nivel global tanto en la clase **Aleatorios.java** como en **Restricciones.java** que contienen los mensajes y el texto de todas las ventanas (cuando se detecta que el usuario cambia el idioma al inglés, se cargan los valores de dichos mensajes en ese idioma).

6 Evaluación

A lo largo de esta sección se procede a detallar cómo ha sido evaluada la implementación de las mejoras mencionadas.

6.1 Pruebas

Al comienzo del proceso de desarrollo, el alumno valoró el uso de pruebas unitarias, sobre todo en cuanto a la implementación de los generadores se refiere. Enseguida se detectó que la automatización de estas pruebas requería un nivel de complejidad demasiado alto:

En primer lugar, se está hablando de generadores de datos aleatorios, por lo que no es posible predecir con exactitud los valores que serían devueltos por cada generador.

Aun así, en el improbable caso de que pudieran predecir los resultados, el proceso de testeo solo sería viable para los generadores de permutaciones y tareas, pues devuelven unas estructuras más pequeñas y sencillas que los grafos.

Tras rechazar la idea de la automatización de pruebas unitarias, el alumno procedió a realizar pruebas manuales. Valiéndose de la herramienta de depuración del IDE IntelliJ, es posible revisar el código que se quiere implementar en tiempo de ejecución, lo que da al programador una visión real de lo que está sucediendo en él.

En cuanto a la evaluación de los generadores de grafos, dado que su estructura y opciones son demasiado grandes como para comprobarlas a simple vista a partir de su matriz de adyacencia, se ha hecho uso de la ya mencionada web **graphonline.ru**, que permite dibujar dichos grafos a partir de la matriz. De esta manera, la comprobación de las estructuras generadas ha sido mucho más sencilla.

A continuación, se muestra la matriz de adyacencia de un grafo multietapa generado en AlgorEx, seguida de su representación gráfica:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Ilustración 22. Matriz de adyacencia de un grafo multietapa

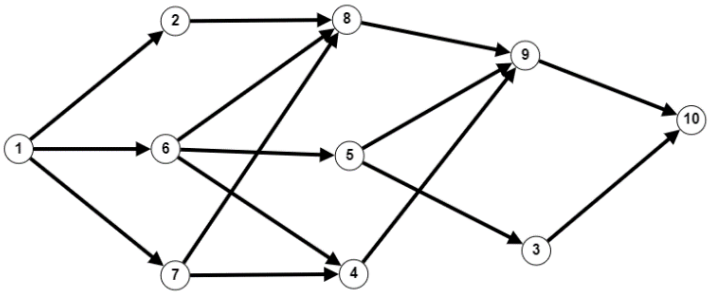


Ilustración 23. Representación gráfica de un grafo multietapa

Una vez implementados los generadores y las refactorizaciones de código se han realizado pruebas funcionales de AlgorEx: cargando diferentes clases con distintos algoritmos, probando otras opciones en cada ejecución, etc. Con la intención de simular una sesión de usuario real.

6.2 Evaluaciones de usabilidad

Como se comentó anteriormente en la sección Proceso de desarrollo, cada vez que se cumplían ciertos hitos se ha enviado una versión de AlgorEx al tutor, quien se ha encargado de comprobar que las estructuras generadas eran correctas, así como de realizar pruebas de usabilidad de la herramienta.

Estas evaluaciones de experto realizadas por el tutor han ayudado a comprobar diferentes casos que el alumno no contemplaba. Han sido realizadas desde un enfoque heurístico (algo informal), no solo funcional. También han permitido perfeccionar la navegación del usuario a través del programa sugiriendo diversos cambios en las nuevas ventanas creadas para cada generador de datos.

6.3 Versiones desarrolladas

Siguiendo el proceso de desarrollo guiado por sprints, y como se ha comentado en el apartado anterior, el alumno ha enviado diferentes versiones de AlgorEx al tutor con motivo de evaluación por parte de este último.

Contando las fases en las que se implementaban los generadores, así como las distintas revisiones en la fase final, se han realizado un total de 9 versiones diferentes de AlgorEx. Siguiendo las indicaciones que se suelen dar a la hora de realizar proyectos de desarrollo de software, serían fácilmente nombrables como AlgorEx_vMajorVersion.MinorVersion, de manera que la versión final de este proyecto quedaría nombrada definitivamente como AlgorEx_v2.9.

7 Conclusiones y trabajos futuros

Este apartado es quizá el más personal y subjetivo por parte del alumno. En él se han tratado de redactar las impresiones sobre el proceso de trabajo y el resultado final del proyecto, así como la exposición de algunas propuestas de mejora para otros posibles TFG.

7.1 Valoración personal

Respecto al resultado final del proyecto, el alumno considera que ha conseguido realizar un buen trabajo. La elección de este TFG parte de la voluntad de realizar un trabajo completo, que demuestre su formación en el Grado de Ingeniería Informática.

Completo, porque inicia desde una fase de documentación e investigación que no suele trabajarse demasiado a lo largo de la carrera, junto a un proceso de desarrollo en el que se combinan la implementación de código Java (lenguaje que más ha trabajado el alumno durante estos últimos años) y la mejora de una interfaz de usuario, una facción de la informática que también habría deseado potenciar más durante su formación.

A pesar de estar conforme con el resultado, el alumno es consciente de que siempre existe cierto margen de mejora. Esta afirmación parte del que ha considerado como el obstáculo más complejo con el que se ha topado durante el proceso de desarrollo: el código previo de AlgorEx.

Hay que tener en cuenta que se ha trabajado sobre lo programado por otras personas: el número de clases Java no resulta demasiado elevado, pero sí lo es cuando nos fijamos en la longitud de varias de ellas. Por ejemplo, la clase Aleatorios.java consta de 4000 líneas de código, y la clase Restricciones.java llega casi al mismo tamaño. Si a esto le añadimos que el código no está demasiado comentado y los nombres de las variables, a juicio del alumno, no resultan debidamente esclarecedoras, sin duda este ha sido el mayor problema al que enfrentarse.

Tal vez otro programador no lo señalaría como una gran dificultad, pero es el caso del alumno que, a raíz de aprender el concepto de Clean Code durante sus estudios, se ha declarado ferviente defensor de la filosofía de trabajo que desarrolla. Debido a ello surge la principal propuesta de un posible TFG sobre la versión actual de AlgorEx, como se comentará más adelante.

Aun así, es posible que el código elaborado no sea óptimo o que las refactorizaciones no hayan resultado las mejores, por lo que a nivel de autocrítica diría que un aspecto negativo del resultado final es la posible imperfección del código implementado, aunque sí cumpla la función que se concibió al inicio del proyecto.

7.2 Estimación del esfuerzo realizado

El alumno considera que una estimación de este tipo es difícil realizarla de manera "tangible" o puramente "analítica" sin conocer el funcionamiento de AlgorEx, aunque ha tratado de realizarlo.

4 clases han sido trabajadas en total (una de ellas, *ArbolBinario*, creada desde cero) y la cantidad de métodos que han sido modificados o creados asciende a 16. Véase la sección de Implementación para más información.

Además, el desarrollo del TFG se ha realizado durante algo más de 9 meses, divididos de la siguiente manera, según el proceso de desarrollo indicado en la sección 3.2:

- *Fase de análisis*: 2 meses
- *Fase de desarrollo*: 5 meses
- *Fase de revisión*: 1 mes
- *Fase de cierre*: 1 mes

No hay que olvidar que se ha trabajado sobre un programa ya creado, grande y como se ha mencionado antes, difícil de comprender a nivel de código. Es por ello que al principio de esta Memoria se ha incluido la sección de Sesión de Usuario, para contextualizar a quien esté leyéndola y le permita entender, en caso de no haber utilizado nunca la herramienta, el verdadero trabajo que ha habido detrás de ella.

Si bien es cierto que el alumno ha realizado un esfuerzo que le ha permitido finalizar este proyecto de manera correcta y cumpliendo con los objetivos iniciales, piensa que durante este tiempo podría haberle dedicado más horas (lo que le habría permitido perfeccionar su trabajo), pues, aunque el cómputo general sea favorable, siempre podría haberse trabajado más: la situación laboral del alumno durante la realización del TFG ha sido activa, lo que no le ha

permitido dedicarse de lleno a ello, pero aun así reconoce que le habría gustado organizarse mejor para dedicar más horas diarias.

7.3 Propuestas de trabajos futuros

Para terminar, se proponen dos ideas para otros posibles Trabajos de Fin de Grado:

1. **Optimización y refactorización del código.** Tal y como se ha mencionado anteriormente, hay varias clases del proyecto cuyo tamaño resulta demasiado grande. Por lo tanto, se propone que, mediante la aplicación de técnicas y principios propios del Clean Code, se mejore la calidad y legibilidad del código. Como este trabajo no sería visible para un usuario genérico de AlgorEx, también podría proponerse un cambio en la interfaz gráfica.
2. **Desarrollo de nuevos generadores de datos.** Siguen existiendo generadores de datos que no han sido implementados en la herramienta, como los grafos dispuestos en forma de malla, anillo o corona, algo que podría resultar útil e interesante para estudiarse.

Referencias

[1] Velázquez-Iturbide, J. Á. (2016). *Design and evaluation of OptimEx, an experimentation system for optimization algorithms*. ICT in Education: Multiple and Inclusive Perspectives, 51-68. https://doi.org/10.1007/978-3-319-22900-3_4

[2] Debdi, O., Velázquez Iturbide, J. Á., & Paredes Velasco, M. (2010). *Revisión Bibliográfica de Problemas Combinatorios Resolubles por Técnicas Básicas de Diseño de Algoritmos.* http://hdl.handle.net/10115/4326

[3] Alsuwaiyel, M. H. (2021). *Algorithms: design techniques and analysis* (Vol. 15). World Scientific. https://doi.org/10.1142/12324

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022*). Introduction to algorithms*. MIT press. https://doi.org/10.1515/9783110522013

[5] Goodrich, M. T., & Tamassia, R. (2015). *Data structures and algorithms in Java.* Hoboken: Wiley.

[6] Horowitz, E., Shani, S., & Rajasekeran, S. (2008). *Computer algorithms.* Summit: Silicon Press.

[7] *Java Documentation*. (s.f.). Obtenido de https://docs.oracle.com/en/java/

Anexo I: Códigos de prueba

Para poder probar la aplicación con los nuevos generadores de datos, se proporcionan tres clases Java: OrdenarIndices.java para el generador de permutaciones; SelecActividades.java para el generador de tareas y ArbolesRecubrimientoCosteMinimo.java para probar los distintos generadores de grafos. Estos códigos, a su vez, han sido facilitador por el tutor del proyecto, J. Ángel Velázquez Iturbide.

- OrdenarIndices.java

```
/**

 * @author (Ángel Velázquez Iturbide, Departamento de Informática y Estadística, Universidad Rey Juan Carlos)

 * Asignatura: Algoritmos Avanzados, Grado en Ingeniería Informática

 * @version (Curso indefinido)

 */

public class OrdenarIndices

{

  // coincide con el problema de almacenamento óptimo en cintas

  // o del tiempo de espera en el sistema;

  // calcula el orden óptimo pero no el valor óptimo

  // ordenando en orden creciente por inserción

  public static void ordenarEnVectorI (int[] v) {

    for (int i=1; i<v.length; i++) {

      int aux = v[i];

      int j;

      for (j=i-1; j>=0 && v[j]>aux; j--)

        v[j+1] = v[j];
```

```
      v[j+1] = aux;
    }
  }
  public static int[] ordenarEnIndicesI (int[] v) {
    int[] v2 = new int[v.length];
    v2[0] = 0;
    for (int i=1; i<v.length; i++) {
      int aux = v[i];
      int j;
      for (j=i-1; j>=0 && v[v2[j]]>aux; j--)
        v2[j+1] = v2[j];
      v2[j+1] = i;
    }
    return v2;
  }
  // ordenando en orden creciente por burbuja
  public static void ordenarEnVectorB (int[] v) {
    for (int i=1; i<v.length; i++)
      for (int j=0; j<v.length-i; j++)
        if (v[j]>v[j+1]) {
          int aux = v[j];
          v[j] = v[j+1];
          v[j+1] = aux;
```

```
        }
    }
    public static int[] ordenarEnIndicesB (int[] v) {
        int[] v2 = new int[v.length];
        for (int i=0; i<v.length; i++)
            v2[i] = i;
        for (int i=1; i<v.length; i++)
            for (int j=0; j<v.length-i; j++)
                if (v[v2[j]]>v[v2[j+1]]) {
                    int aux = v2[j];
                    v2[j] = v2[j+1];
                    v2[j+1] = aux;
                }
        return v2;
    }
}
```

- SelecActividades.java

```
/**
 * @author (Ángel Velázquez Iturbide, Departamento de Informática y Estadística, Universidad Rey Juan Carlos)
 * Asignatura: Algoritmos Avanzados, Grado en Ingeniería Informática
 * @version (Curso indefinido)
 */
```

```
public class SelecActividades
{
  public static int selecActividades1 (int[] c, int[] f) {
    // actividades ordenadas en orden creciente de fin
    int num = 1;
    int i = 0;
    for (int j=1; j<c.length; j++) {
      if (c[j]>=f[i]) {
        num++;
        i = j;
      }
    }
    return num;
  }
  public static int selecActividades2 (int[] c, int[] f) {
    // actividades ordenadas en orden creciente de fin
    // determina también las actividades seleccionadas
    boolean[] s = new boolean[c.length];
    s[0] = true;
    int num = 1;
    int i = 0;
    for (int j=1; j<c.length; j++) {
      if (c[j]>=f[i]) {
```

```
            s[j] = true;
            num++;
            i = j;
          }
          else
            s[j] = false;
        }
        imprimir (s);
        return num;
      }
      private static void imprimir (boolean[] s) {
        for (int i=0; i<s.length; i++)
          System.out.print (s[i]+"  ");
        System.out.println ();
      }
}
```

- ArbolesRecubrimientoCosteMinimo.java

```
/**
 * @author (Ángel Velázquez Iturbide, Departamento de Informática y Estadística, Universidad Rey Juan Carlos)
 * Asignatura: Algoritmos Avanzados, Grado en Ingeniería Informática
 * @version (Curso indefinido)
 */
```

```
public class ArbolesRecubrimientoCosteMinimo
{
  public static int Prim (int[][] grafo) {
    // calcula el coste mínimo y los arcos correspondientes
    // grafo como matriz de adyacencia
    // vectores que representan los arcos del MST (orígenes y destinos)
    int[] orig = new int[grafo.length-1];
    int[] dest = new int[grafo.length-1];
    int coste = 0;
    // representación implícita del subconjunto S de nodos candidatos
    // masProx[i] contiene el nodo de S más cercano al nodo i
    // distMin[i] = -1 si el nodo i pertenece a S
    //            = si no, distancia entre i y masProx[i]
    int[] masProx = new int[grafo.length];
    int[] distMin = new int[grafo.length];
    // inicialización de vectores auxiliares, suponiendo que el MST contiene al nodo cero
    for (int i=1; i<grafo.length; i++) {
      masProx[i] = 0;
      distMin[i] = grafo[i][0];
    }
    // bucle voraz
    for (int i=0; i<grafo.length-1; i++) {
      // se selecciona nodo más cercano al MST ya formado
```

```
        int min = Integer.MAX_VALUE;
        int k = 0;
        for (int j=1; j<grafo.length; j++)
          if ((0<=distMin[j]) && (distMin[j]<min)) {
            min = distMin[j];
            k = j;
          }
        // se añade el arco al nodo k al MST
        distMin[k] = -1;
        orig[i] = masProx[k];
        dest[i] = k;
        coste += grafo[orig[i]][dest[i]];
        // se actualizan las distancias mínimas al resto de nodos
        for (int j=1; j<grafo.length; j++)
          if (grafo[k][j]<distMin[j]) {
            distMin[j] = grafo[k][j];
            masProx[j] = k;
          }
      }
      imprimirArcos (orig, dest);
      return coste;
    }
    public static int Kruskal1 (int[] us, int[] vs, int[] costes, int n) {
```

```
// calcula el coste mínimo y los arcos correspondientes
// grafo como lista de adyacencia
// presupone que los arcos vienen ordenados en orden creciente de coste
// vectores que representan los arcos del MST (orígenes y destinos)
int[] orig = new int[n-1];
int[] dest = new int[n-1];
int a = 0;
int coste = 0;
// representación implícita del bosque de nodos que forman el MST
int[] conjs = new int[n];
for (int i=0; i<n; i++)
  conjs[i] = i;
// bucle voraz
for (int i=0; i<us.length && a<n; i++) {
  // se selecciona el arco más corto
  int u = us[i];
  int v = vs[i];
  // se halla el conjunto disjunto de sus nodos
  int conju = conjs[u];
  int conjv = conjs[v];
  // se comprueba si pertenecen a conjuntos disjuntos
  if (conju != conjv) {
    orig[a] = u;
```

```
            dest[a] = v;
            a++;
            coste += costes[i];
            // se fusionan
            int min = Math.min (conju, conjv);
            int max = Math.max (conju, conjv);
            for (int k=0; k<n; k++)
              if (conjs[k]==max)
                conjs[k] = min;
          }
        }
        imprimirArcos (orig, dest);
        return coste;
    }
    public static int Kruskal2 (int[] us, int[] vs, int[] costes, int n) {
        // utiliza un TAD para conjuntos disjuntos
        // calcula el coste mínimo y los arcos correspondientes
        // grafo como lista de adyacencia
        // presupone que los arcos vienen ordenados en orden creciente de coste
        // vectores que representan los arcos del MST (orígenes y destinos)
        int[] orig = new int[n-1];
        int[] dest = new int[n-1];
        int a = 0;
```

```
int coste = 0;
// representación implícita del bosque de nodos que forman el MST
ConjuntosDisjuntos.ConjsDisjuntos (n);
// bucle voraz
for (int i=0; i<us.length && a<n; i++) {
  // se selecciona el arco más corto
  int u = us[i];
  int v = vs[i];
  // se halla el conjunto disjunto de sus nodos
  int conju = ConjuntosDisjuntos.SuConj(u);
  int conjv = ConjuntosDisjuntos.SuConj(v);
  // se comprueba si pertenecen a conjuntos disjuntos
  if (conju != conjv) {
    orig[a] = u;
    dest[a] = v;
    a++;
    coste += costes[i];
    // se fusionan
    ConjuntosDisjuntos.FusionarConjs (conju, conjv);
  }
}
imprimirArcos (orig, dest);
return coste;
```

```
    }

    private static void imprimirArcos (int[] orig, int[] dest) {

      for (int i=0; i<orig.length; i++)

        System.out.println ("Arco "+i+": ("+orig[i]+","+dest[i]+")");

      System.out.println ();

    }

  }
```

Utiliza métodos de la clase ConjuntosDisjuntos.java:

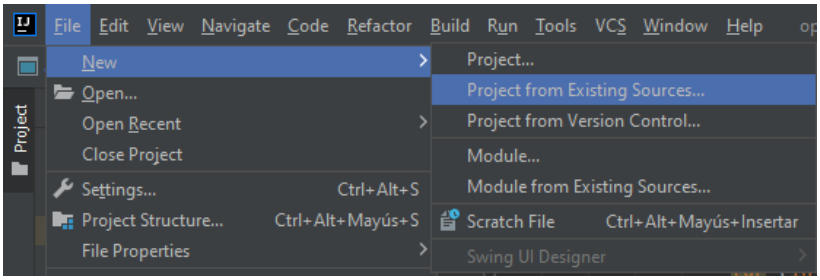
```
/**

 * @author (Ángel Velázquez Iturbide, Departamento de Informática y Estadística, Universidad Rey Juan Carlos)

 * Asignatura: Algoritmos Avanzados, Grado en Ingeniería Informática

 * @version (Curso indefinido)

 */

public class ConjuntosDisjuntos

{

   static int num;

   static int[] conjs;

   public static void ConjsDisjuntos (int n) {

     num = n;

     conjs = new int[n];

     for (int i=0; i<num; i++) {
```

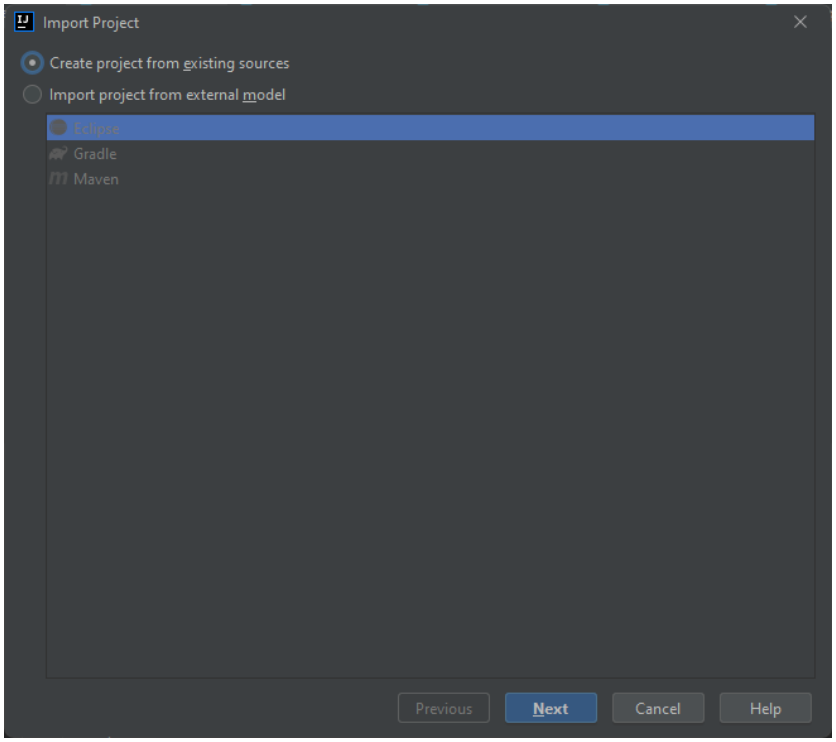
```
        conjs[i] = i;
      }
   }
   public static int SuConj (int nodo) {
     return conjs[nodo];
   }
   public static void FusionarConjs (int c1, int c2) {
      int min = Math.min (c1, c2);
      int max = Math.max (c1, c2);
      for (int k=0; k<num; k++) {
        if (conjs[k]==max)
           conjs[k] = min;
      }
   }
}
```

Anexo II: Manual del programador

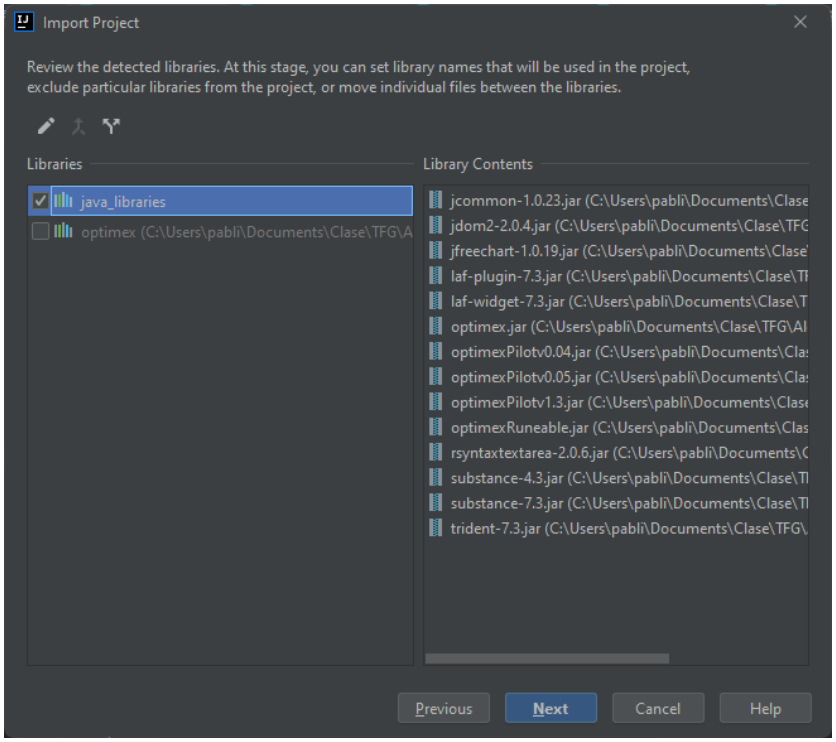
A continuación, se explica cómo construir la aplicación de forma correcta en el entorno IntelliJ IDEA, de cara a ser utilizado para futuros proyectos. Es necesario contar con la carpeta *java_libraries*, en la que se encuentran las librerías necesarias para construir el ejecutable del proyecto y la carpeta contenedora *Optimex*, donde se encuentran los archivos de código.

1. Se crea un nuevo proyecto a partir del existente.

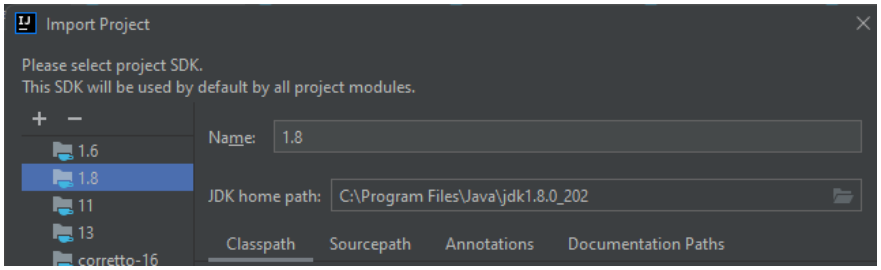




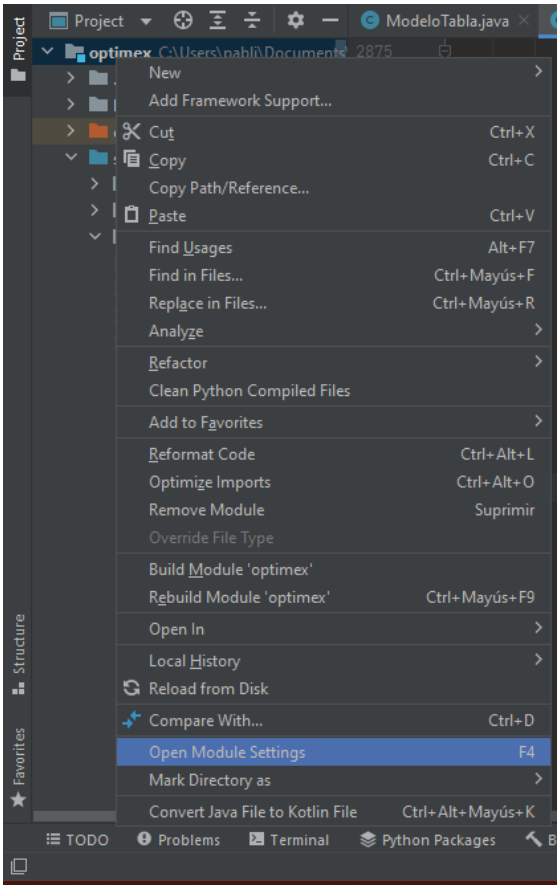
2. Hay que hacer click en “Next” hasta que aparezca la ventana de selección de librerías Java, donde hay que marcar todas las incluidas en la carpeta *java_libraries*.



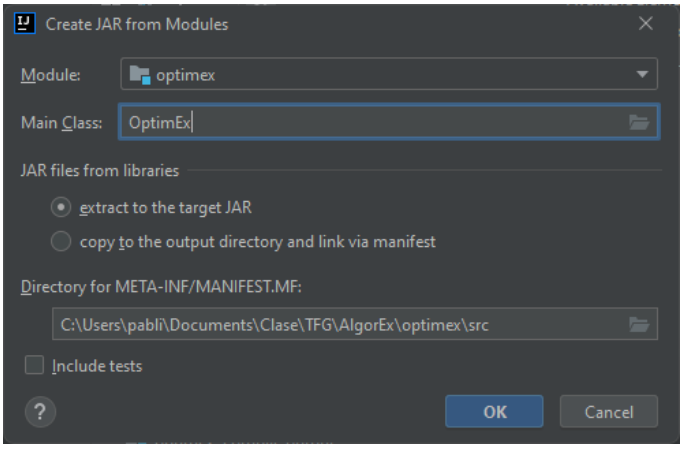
3. Se continúa de la misma manera, hasta que aparezca la opción de selección del SDK, donde hay que añadir la ruta del que se tenga instalado .



4. Cuando se haya importado el proyecto, se abre el panel de dependencias desde la raíz del proyecto.



5. Desde la sección Artifacts, hay que crear uno nuevo pulsando el botón + y seleccionando "Add -> Jar -> From modules with dependencies". Aplicando la configuración que se muestra a continuación, se permite la generación del artefacto encargado de gestionar el archivo .jar.



6. Haciendo click en el nuevo artefacto, se pueden revisar las dependencias enlazadas. Es necesario marcar la casilla “Include in the project build”, para que cada vez que se pulse “Build” en el proyecto se genere también el jar.