Manual Técnico

Simplex Educativo

Profesora: Maria Estrada.

Curso: Proyecto de Ingeniería de Software.

13.12.2016

Jose Fernando Molina Chacón

Yordan Jiménez Hernández

TABLE OF CONTENTS

Introducción

Durante el transcurso del curso de Investigación de Operaciones de la carrera de Ingeniería en Computación impartida en el Instituto Tecnológico de Costa Rica, se abarcan temas de maximización y minimización de costos, métodos matemáticos que pueden ser computarizados para su utilización en diversos campos. En la educación, un método utilizado en el curso nombrado anteriormente es el método Simplex. Dicho método es ampliamente conocido, y existen herramientas de software como Lindo o páginas web PhpSimplex que proveen un programa para solucionar dichos problemas.

Sin embargo, estos programas poseen grandes limitaciones a la hora de utilizarlos como herramienta educativa. Algunos de ellos solamente solucionan el problema, sin dar información que puede ser relevante para el profesor o estudiantes. Por otra parte, cada software representa los problemas de programación lineal por medio de una matriz con un formato diferente, por lo que se vuelve difícil interpretar los resultados o datos provistos por estas herramientas. Es por estos motivos que para la enseñanza en el campus estas herramientas no poseen las características que faciliten el aprendizaje de los estudiantes o ayuden al transcurso de la clase impartida por el profesor encargado del curso.

El programa Simplex Educativo se centra en la implementación de una solución computacional que solucione problemas de programación lineal por medio del algoritmo simplex de una manera dinámica y amigable para el aprendizaje, donde el usuario puede indicar distintas formas de solucionar un ejemplo de maximización o minimización, con el fin que el usuario pueda resaltar características relevantes a los estudiantes acerca de comportamientos que tome el algoritmo simplex ante los diferentes escenarios posibles.

El usuario podrá ingresar textualmente un problema de programación lineal con un formato definido, y el programa podrá identificarlo y resolverlo de manera que el usuario pueda entender cada uno de los pasos que implica la resolución de un problema de este tipo. Debido a la naturalez educativa del producto, no hay restricciones en cuanto a eficiencia del programa. La solución debe ser portable a través de diferentes sistemas operativos, por lo tanto será desarrollada en el lenguaje de programación Java, debido a que este lenguaje no es dependiente de la plataforma en que se ejecute. El sistema no poseerá manejo de usuarios ni autenticaciones.

Se desarrollaron conjuntamente dos sistemas: Una aplicación para escritorio y una aplicación móvil para dispositivos Android. Ambos sistemas poseen la funcionalidad básica de interpretar un problema ingresada a manera de texto y resolverlo utilizando el método simplex. Sin embargo, la aplicación de escritorio poseerá más funcionalidades que la versión móvil. La versión móvil solamente poseerá la capacidad de mostrar los pasos intermedios a modo de tabla resumen, o

llegar de manera directa a una solución. Por su parte, la versión de escritorio podrá resolver un problema de programación lineal entero (esto es, donde su solución sean números enteros) mediante el algoritmo de Gomory o Branch and Bound, ingresar una matriz numérica textual en lugar de un problema, escoger la posición en la cual se va a pivotar en cada paso del algoritmo, cambiar entradas de la matriz una vez iniciado el algoritmo y agregar restricciones, filas o columnas a la matriz de cada paso.

Propósito

El propósito de este documento es proveer a las partes interesadas en el funcionamiento del sistema Simplex Educativo con la mayor cantidad de información posible acerca de la de la manera en que fue implementada la solución. Se tratará de describir en detalle todos los componentes de software involucrados, la arquitectura y las decisiones que se tomaron durante el desarrollo de producto, la interacción entre las diferentes partes del software y en general todo el conocimiento que tiene el equipo de desarrollo que pueda ser utilizado por personas ajenas al equipo para modificar o extender la solución. Se espera que este documento pueda ser una referencia para desarrolladores que quieran comprender a fondo la implementación del software Simplex Educativo, además se espera que las personas que hayan leído el documento sean capaces de comprender más sencillamente el código fuente del programa, y de esta manera poder realizar futuros cambios, extensiones o cualquier tipo de modificación al producto de una manera sencilla, disminuyendo así el tiempo de aprendizaje acerca de la implementación de un nuevo producto de software.

Alcance

El proyecto incluye el desarrollo de una aplicación para escritorio multiplataforma y una aplicación móvil para el sistema operativo Android que ayude al profesor PhD. José Elías Helo a desarrollar su clase de Investigación de Operaciones de una manera dinámica, que propicie el entendimiento de la materia por parte del estudiante. Una vez desarrollada, esta aplicación será utilizada por el profesor solamente y será propiedad total y exclusiva de él, incluido el código fuente del programa.

El proyecto incluye documentación técnica en forma de comentarios en el código fuente, manual de usuario y documentación externa. También incluye una serie de artefactos de aseguramiento de la calidad como pruebas unitarias y de integración, así como pruebas de sistema y pruebas de aceptación que serán realizadas por el cliente.

El proyecto no incluye el mantenimiento de la aplicación una vez entregado el producto. Cuando se haga la entrega final del proyecto, el cliente se hará

responsable del producto de ese momento en adelante y el equipo de desarrollo ya no será responsable del mismo.

La construcción del proyecto está estimada en un plazo no mayor a 35 días. La aplicación de escritorio podrá ser ejecutada en distintas plataformas ya sea Windows, Mac o Linux. Para ello se desarrollará en el lenguaje de programación java, el cual es multiplataforma y portable.

La aplicación móvil podrá ser ejecutada en dispositivos con el sistema operativo Android en su versión 6.0 o superior.

Definiciones, acrónimos y abreviaturas

- **ERS:** Especificación de Requerimientos de Software.
- **MVC:** Model-View-Controller. Patrón de diseño arquitectural.
- **Método Simplex:** Algoritmo creado por George Dantzig para resolver problemas de programación lineal.
- **Programación Lineal:** Modelo matemático para buscar la optimización de una función objetivo que cumpla con N restricciones al mismo tiempo.
- **Java:** Lenguaje de programación orientado a objetos.
- **JDK**: Java Development Kit. Herramientas de desarrollo del lenguaje Java.
- **UML:** Unified Modeling Language. Lenguaje gráfico descriptivo para uso en procesos de software.
- **Problema Infactible:** Problema de programación lineal que debido a sus restricciones hace que la función objetivo no se pueda cumplir bajo ninguna circunstancia.
- **Problema No Acotado:** Problema de programación lineal que debido a sus restricciones hace que la función objetivo pueda ser maximizada o minimizada de manera infinita (no existe límite para detenerse).
- **BVS:** Basic Variables. Variables básicas, es decir, variables que poseen un valor definido en cierto momento.
- **RHS:** Right Hand Side. Lado derecho de una igualdad o desigualdad. En representación matricial simplex, la última columna de derecha a izquierda.
- **Radios:** Resultado de dividir el RHS entre una columna N de una matriz simplex. Tienen la particularidad de que si la división es entre 0 o el resultado es un número negativo, el resultado de los radios será infinito.
- **Pivotar:** Tomar una entrada diferente de cero de una matriz y realizar operaciones fila de manera que la columna de esa entrada sea básica, es decir, solamente posea 0ñs y solamente un 1 en la posición de pivote escogida.
- **Android:** Sistema operativo móvil desarrollado por Google.

- **Scanner:** Programa computacional que tiene por entrada una cadena de caracteres. Toma esta cadena de caracteres y la divide en “tokens”, estos son una unidad de texta que tiene algun valor o dignificado implícito.
- **Parser:** Programa computacional que tiene por entrada una cadena de caracteres. Trabaja en conjunto con el scanner para validar que la cadena cumpla un formato específico y extrae la información relevante de dicho formato.
- **LALR**: Look-Ahead Left to Right. Un tipo de parser.

Referencias

1. Documento de Visión de Simplex Educativo.
2. Especificación de Requerimientos de Software de Simplex Educativo.
3. Documento de Arquitectura de Software Simplex Educativo.
4. Plan de Pruebas de Simplex Educativo.
5. Manual de Usario de Simplex Educativo.
6. Manual de Usuario de JFlex: http://jflex.de/manual.html.
7. Manual de Usaurio de Java CUP: http://www2.cs.tum.edu/projekte/cup/manual.html

Perspectiva del sistema

Características del sistema

En el siguiente apartado se describirán las características del sistema Simplex Educativo en su versión de escritorio. Estas características fueron tomadas del documento “Especificación de Requerimientos de Software de Simplex Educativo”.

Verificar la factibilidad de un problema de programación lineal ingresado.

Descripción

Existen ciertos problemas de programación lineal que, debido a las restricciones que los definen, es matemáticamente imposible cumplir con todas las restricciones impuestas. Por ejemplo, que una restricción defina la variable x1 <= 4 y otra defina x1>=5. El sistema debe poseer la capacidad para determinar si un problema de

programación lineal posee una solución factible óptima. Para ello, podrá reconocer una cadena de texto ingresada por el usuario con un formato establecido (el formato utilizado por el profesor Jose Helo en sus apuntes de clase) y extraer de él la información necesaria para representar el problema en forma tabular y, por medio del algoritmo Simplex, determinar si existe una solución óptima factible.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato de salida del programa, fraccionario o decimal.
4. El usuario escoge el modo de solución directa.
5. El usuario selecciona la opción de solucionar un problema.
6. El sistema muestra por medio de un mensaje si el problema es infactible.
7. El usuario cierra el mensaje.

Requerimientos Funcionales

REQ-1: Infactibilidad de un problema

El sistema debe de ser capaz de interpretar el problema de programación lineal ingresado por el usuario por medio de una cadena de texto con un formato definido e indicar si las características presentes en su contenido por medio de restricciones, variables u otros, hacen que el problema no contenga una solución factible óptima. Dicho resultado es mostrado por medio de un mensaje de información junto con la lista de pasos realizados.

Obtener de manera inmediata una solución óptima de un problema de programación lineal.

Descripción

Los problemas de programación lineal buscan maximizar o minimizar una función objetivo con N variables lineales sujetas a M inecuaciones lineales que incluyen las variables de la función objetivo.

El método Simplex es un algoritmo desarrollado por George Dantzig para resolver esta clase de problemas. Ellos se utilizan en diversos campos de la matemática aplicada e ingenierías para encontrar soluciones a problemas que puedan ser modelados mediante el método de programación lineal. El sistema será capaz de solucionar un problema de programación lineal con N variables y M restricciones lineales por medio del método Simplex, brindando inmediatamente la solución al usuario.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato numérico en el cual desea la respuesta.
4. El usuario selecciona la opción de solución inmediata.
5. El usuario selecciona que desea resolver el problema por medio del método Simplex.
6. El sistema interpreta el problema ingresado y obtiene la primera solución óptima que genere el método Simplex.
7. El sistema muestra el resultado óptimo final del problema.

Requerimientos Funcionales

REQ-2 Interpretar un problema ingresado:

El sistema debe interpretar dentro del problema de programación lineal ingresado por medio de una cadena de texto por parte del usuario los valores enteros, valores decimales o fraccionarios, variables y restricciones que se encuentren representados en dicha cadena. En caso de que no contenga el formato esperado, el sistema le indicará al usuario por medio de un mensaje la naturaleza del error.

REQ-3 Solución por medio del Simplex:

El sistema debe utilizar el método Simplex común desarrollado por George Dantzig y enseñado por el profesor José Helo en su clase de investigación de operaciones para obtener la primera solución óptima de un problema de programación lineal.

REQ-4 Resultados fraccionarios o decimales:

Los valores numéricos brindados por la ejecución del sistema deben ser mostrados en formato decimal o fraccionario. El formato decimal será representado mediante 2 decimales, mientras que el formato fraccionario será representado mediante una fracción con numerador y denominador enteros. La elección del formato será por parte del usuario antes de ejecutar la solución del problema.

Listar los pasos intermedios para encontrar la solución a un problema de programación lineal.

Descripción

El algoritmo simplex representa matricialmente (esto es, por medio de una matriz numérica) el problema de programación lineal de modo que cada columna representa una variable y cada fila representa una restricción. Por medio de operaciones de pivote (esto significa escoger una entrada de la matriz, convertir en 1 dicha entrada y realizar operaciones fila para reducir las otras entradas de la columna a 0), el algoritmo se traslada entre las posibles soluciones hasta encontrar una solución óptima. En cada paso del algoritmo, hay una operación de pivote que cambia el estado de la matriz que representa el problema lineal.

El sistema debe ser capaz de solucionar un problema de programación lineal por medio del método Simplex donde se pueda generar las tablas o matrices intermedias que se produjeron para obtener la primera solución óptima del problema ingresado. Esto significa que debe ser posible observar todas las operaciones de pivote realizadas para llegar a la solución óptima encontrada.

Prioridad

Alta.

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato numérico en el cual desea la respuesta.

4. El usuario selecciona la opción de solución por pasos.
5. El usuario selecciona que desea resolver el problema por medio del método Simplex.
6. Sistema muestra una a una cada iteración (operación de pivote) necesaria para obtener la solución óptima. El usuario avanza manualmente mediante un botón entre las iteraciones.
7. El sistema muestra que se ha obtenido la solución final óptima.

Requerimientos Funcionales

REQ-5 Iteraciones intermedias de la solución de un problema de programación lineal

El sistema será capaz de interpretar las iteraciones intermedias que se generaron a la hora de obtener una solución óptima de un problema de programación lineal por medio del algoritmo Simplex común y las mostrará al usuario en pantalla en forma de matriz numérica. En el transcurso de dicho proceso se indicarán las características de la iteración, esto significa mostrar las variables de holgura o artificiales agregadas y las variables originales, mostrando en pantalla la representación matricial y mensajes que ayuden al usuario a comprender el estado actual del algoritmo.

Verificar si un problema de programación lineal ingresado se encuentra acotado.

Descripción

Existen ciertos problemas de programación lineal que, debido a las restricciones que los definen, es matemáticamente imposible llegar a un máximo o un mínimo absoluto. Por ejemplo, si se buscara maximizar la funcion f(x1,x2) = x1 + x2, sujeto solamente a la restricción x1 <= 4, entonces estamos ante un problema no acotado. Esto significa que no hay un límite para maximizar la función f(x1,x2), pues la variable x2 no posee un límite (acote) bajo el cual se deba mantener, por lo que la función puede ser maximizada infinitamente.

El sistema debe poseer la capacidad para determinar si un problema de programación lineal posee una solución acotada, esto es, posee un límite superior o inferior. El algoritmo simplex explica cuándo un problema no está acotado, esto sucede cuando los radios (división entre el lado derecho de la matriz y la columna sobre la cual se busca pivotar) que se utilizan para elegir la fila pivote son todos infinito,

significando que la función objetivo puede crecer de manera infinita sobre todas las restricciones.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato numérico en el cual desea la respuesta.
4. El usuario selecciona que desea resolver un problema de programación lineal.
5. El usuario selecciona la opción de solución inmediata.
6. El sistema muestra por medio de un mensaje la solución del problema, o un mensaje de error si el problema no se encuentra acotado.

Requerimientos Funcionales

REQ-6 Problema No Acotado:

El sistema debe de ser capaz de interpretar el problema de programación lineal ingresado por el usuario e indicar si las características presentes en su contenido por medio de restricciones, variables u otros, hacen que el problema no contenga una solución acotada. Dicho resultado es mostrado al usuario por medio de un mensaje de advertencia en caso que el problema no se encuentre acotado.

Ingresar una matriz para realizar operaciones

Descripción

Debido a que en esencia el algoritmo simplex trabaja con operaciones de matrices, el cliente desea poseer la capacidad de ingresar manualmente una matriz de N x M números en formato fraccionario o decimal y escoger manualmente la entrada sobre la cual se debe pivotar.

También deben mostrarse los radios entre el lado derecho de la matriz (esto es, la última columna) y la columna que está seleccionada en cierto momento. El usuario será el encargado de escoger en cada paso la entrada sobre la cual desea pivotar, y el sistema no hará

ninguna asunción o sugerencia acerca de la próxima entrada que debería ser escogida en cada paso.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa la cadena de texto que representa la matriz en el apartado correspondiente.
3. El usuario selecciona que desea realizar operaciones a una matriz numérica.
4. El sistema muestra la representación de la matriz.
5. El usuario escoge la casilla donde desea hacer el pivoteo.
6. El usuario realiza el pivoteo.

Requerimientos Funcionales

REQ-7 Interpretar matriz ingresada:

El sistema debe de ser capaz de interpretar la matriz numérica ingresada por el usuario e indicar si las características presentes en su contenido hacen que la matriz tenga alguna información o dato inválido. En caso de que exista alguna inconsistencia o el formato sea inválido, el sistema responderá con un mensaje de información que clarifique el error.

REQ-8 Realizar pivoteos en las matrices intermedias:

Ya sea por el ingreso de un problema de programación lineal o por el ingreso de una matriz, debe ser posible para el usuario elegir la casilla donde desee realizar el siguiente pivoteo. Para ello, el usuario escogerá mediante el mouse la casilla donde desea realizar el siguiente pivoteo, el sistema responderá marcando la casilla escogida e indicando las operaciones fila que realizará.

Obtener de manera inmediata la solución entera de un problema de programación lineal mediante el algoritmo de Gomory.

Descripción

El método Simplex para resolución de problemas de programación lineal encuentra soluciones factibles en el dominio de los

números reales. Esto significa que es posible encontrar respuestas que no sean enteras, osea, fracciones y decimales pueden ser el valor que tomen las variables o el resultado de la función objetivo. Sin embargo, en algunos casos es necesario encontrar la solución entera a un problema de programación lineal por su naturaleza, debido a que puede ser el caso de que las variables representan objetos de la vida real que no puedan ser subdivididos en partes fraccionarias.

Para resolver el problema de encontrar la solución entera óptima existen varios métodos, uno de ellos es el denominado algoritmo de Cortes de Gomory, en el cual se resuelve un problema de programación lineal mediante el Simplex común y, en caso de llegar a una solución óptima no entera, se agrega una restricción extra al problema (denominado “corte”) y se vuelve a resolver el problema. Este método se repite N veces hasta encontrar una respuesta con todas las variables y el resultado de la función objetivo enteras.

Debe ser posible encontrar la solución entera óptima de un problema de programación lineal por medio del algoritmo de cortes de gomory de manera inmediata, esto es, sin mostrar los pasos intermedios. Una vez encontrada la solución, el usuario podrá observar el resumen de pasos o la tabla final, siendo posible devolverse en la lista de soluciones.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato numérico en el cual desea la respuesta.
4. El usuario selecciona que desea resolver el problema por medio del método Cortes de Gomory.
5. El usuario selecciona la opción de solución directa.
6. El sistema interpreta el problema ingresado y obtiene la primera solución óptima que genere el método Cortes de Gomory.
7. El sistema muestra el resultado óptimo final del problema.

Requerimientos Funcionales

REQ-9 Obtener solución entera directa por medio del algoritmo de Cortes de Gomory:

El sistema debe de ser capaz de obtener la solución óptima entera de manera directa, sin listar los pasos intermedios, de un problema de programación lineal ingresado utilizando el algoritmo de Gomory. El sistema debe agregar restricciones en caso de no encontrar una solución entera hasta el momento que se llegue a tener el valor de todas las variables, junto con el valor de la función objetivo enteros.

Listar los pasos intermedios para encontrar la solución de un problema de programación lineal mediante el algoritmo de Gomory

Descripción

El método Simplex para resolución de problemas de programación lineal encuentra soluciones factibles en el dominio de los números reales. Esto significa que es posible encontrar respuestas que no sean enteras, osea, fracciones y decimales pueden ser el valor que tomen las variables o el resultado de la función objetivo. Sin embargo, en algunos casos es necesario encontrar la solución entera a un problema de programación lineal por su naturaleza, debido a que puede ser el caso de que las variables representan objetos de la vida real que no puedan ser subdivididos en partes fraccionarias.

Para resolver el problema de encontrar la solución entera óptima existen varios métodos, uno de ellos es el denominado algoritmo de Cortes de Gomory, en el cual se resuelve un problema de programación lineal mediante el Simplex común y, en caso de llegar a una solución óptima no entera, se agrega una restricción extra al problema (denominado “corte”) y se vuelve a resolver el problema. Este método se repite N veces hasta encontrar una respuesta con todas las variables y el resultado de la función objetivo enteras.

Debe ser posible listar los pasos intermedios para encontrar la solución entera óptima de un problema de programación lineal por medio del algoritmo de cortes de gomory. El sistema debe ir indicando paso por paso el estado actual del problema por medio de mensajes y notificar cuándo se ha llegado un óptimo no entero, junto con el corte agregado y la fila que fue escogida para realizar el corte.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato numérico en el cual desea la respuesta.
4. El usuario selecciona que desea resolver el problema por medio del método Cortes de Gomory.
5. El usuario selecciona la opción de solución por pasos.
6. El sistema interpreta el problema ingresado y muestra la primera tabla del problema.
7. El usuario indica que desea conocer el siguiente paso.
8. El sistema muestra el siguiente paso.
    a. Si existe otro paso, volver a 7.
9. El sistema muestra la solución óptima entera encontrada.

Requerimientos Funcionales

REQ-10 Obtener los pasos intermedios para resolver un problema de programación lineal por medio del algoritmo de Gomory

El sistema debe de ser capaz de obtener los pasos intermedios para encontrar la solución óptima entera de un problema de programación lineal ingresado por medio del algoritmo de Gomory. Dichos pasos serán mostrados al usuario en representación matrícula, junto con información relevante como nombres de las variables, variables básicas y radios en cada paso. El sistema irá agregando restricciones una vez encontrada una solución óptima no entera y notificará al usuario por medio de un mensaje.

Obtener de manera inmediata una solución entera de un problema de programación lineal mediante el algoritmo de Branch and Bound.

Descripción

El método Simplex para resolución de problemas de programación lineal encuentra soluciones factibles en el dominio de los números reales. Esto significa que es posible encontrar respuestas que no sean enteras, osea, fracciones y decimales pueden ser el valor que tomen las variables o el resultado de la función objetivo. Sin embargo, en algunos casos es necesario encontrar la solución entera

a un problema de programación lineal por su naturaleza, debido a que puede ser el caso de que las variables representan objetos de la vida real que no puedan ser subdivididos en partes fraccionarias.

La meta de este caso de uso es la misma del caso de uso anterior, llegar a la solución entera óptima para un problema de programación lineal, pero esta vez se debe utilizar el algoritmo de Branch and Bound. Este algoritmo es parecido a los Cortes de Gomory debido a que también agrega restricciones al final del problema, pero se diferencia en que utiliza un árbol de soluciones que va poco a poco siendo construido por medio de restricciones mayor o igual y menor o igual. Dicho árbol es generado hasta encontrar una solución entera óptima.

Debe ser posible encontrar la solución entera óptima de un problema de programación lineal por medio del algoritmo de Branch and Bound de manera inmediata. Dicho algoritmo resuelve N cantidad de subproblemas simplex agregando restricciones según sea necesario en un paso dado, llegando a acotar la solución en algún momento al valor óptimo válido.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato numérico en el cual desea la respuesta.
4. El usuario selecciona que desea resolver el problema por medio del método Branch and Bound.
5. El usuario selecciona la opción de listar los pasos intermedios.
6. El sistema interpreta el problema ingresado y obtiene la primera solución óptima que genere el método Branch and Bound.
7. El sistema muestra el resultado óptimo final del problema.

Requerimientos Funcionales

REQ-11 Obtener solución entera directa por medio del algoritmo de Branch and Bound:

El sistema debe de ser capaz de obtener la solución óptima entera de manera directa, sin listar los pasos intermedios, de un problema de programación lineal ingresado

por medio del algoritmo de Branch and Bound. Una vez solucionado, debe ser posible para el usuario observar los subproblemas generados por medio de un resumen en forma textual.

Listar los pasos intermedios para encontrar la solución de un problema de programación lineal mediante el algoritmo de Branch and Bound.

Descripción

El método Simplex para resolución de problemas de programación lineal encuentra soluciones factibles en el dominio de los números reales. Esto significa que es posible encontrar respuestas que no sean enteras, osea, fracciones y decimales pueden ser el valor que tomen las variables o el resultado de la función objetivo. Sin embargo, en algunos casos es necesario encontrar la solución entera a un problema de programación lineal por su naturaleza, debido a que puede ser el caso de que las variables representan objetos de la vida real que no puedan ser subdivididos en partes fraccionarias.

La meta de este caso de uso es la misma del caso de uso anterior, llegar a la solución entera óptima para un problema de programación lineal, pero esta vez se debe utilizar el algoritmo de Branch and Bound. Este algoritmo es parecido a los Cortes de Gomory debido a que también agrega restricciones al final del problema, pero se diferencia en que utiliza un árbol de soluciones que va poco a poco siendo construido por medio de restricciones mayor o igual y menor o igual. Dicho árbol es generado hasta encontrar una solución entera óptima.

Debe ser posible encontrar la solución entera óptima de un problema de programación lineal por medio del algoritmo de Branch and Bound de manera inmediata. Dicho algoritmo resuelve N cantidad de subproblemas simplex agregando restricciones según sea necesario en un paso dado, llegando a acotar la solución en algún momento al valor óptimo válido.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato numérico en el cual desea la respuesta.
4. El usuario selecciona que desea resolver el problema por medio del método Branch and Bound.
5. El usuario selecciona la opción de solución por pasos.
6. El sistema interpreta el problema ingresado y muestra la primera tabla del problema.
7. El usuario indica que desea conocer el siguiente paso.
8. El sistema muestra el siguiente paso.
    a. Si existe otro paso, volver a 7.
9. El sistema muestra la solución óptima entera encontrada.

Requerimientos Funcionales

REQ-12 Obtener los pasos intermedios para resolver un problema de programación lineal por medio del algoritmo de Branch and Bound.

El sistema debe de ser capaz de obtener los pasos intermedios para encontrar la solución óptima entera de un problema de programación lineal ingresado por medio del algoritmo de Branch and Bound. El sistema irá desplegando a manera de árbol de texto las soluciones a los subproblemas encontrados solamente.

Agregar restricciones a un problema de programación lineal.

Descripción

El usuario desea poder agregar manualmente restricciones una vez que se muestra el problema en su representación matricial. Esto se debe a que el usuario desea manualmente agregar cortes de gomory o restricciones de Branch and Bound manualmente como técnica educativa.

Debe ser posible, una vez iniciado el programa, agregar restricciones a la matriz que representa el programa de programación lineal. Dichas restricciones pueden ser de tipo mayor igual, menor igual o igual. En el caso de menor o igual, se agregará una nueva variable de holgura. En el caso de igual, se agregará una nueva variable artificial. En el caso de mayor o igual, se agregará una variable negativa de holgura y una artificial.

Prioridad

Alta

Secuencia Estímulo/Respuesta

1. El sistema muestra el apartado donde se ingresa el problema de programación lineal.
2. El usuario ingresa el problema de programación lineal por medio de una cadena de texto.
3. El usuario escoge el formato numérico en el cual desea la respuesta.
4. El usuario selecciona que desea resolver el problema por medio del método Simplex.
5. El usuario selecciona la opción de solución por pasos.
6. El sistema interpreta el problema ingresado y muestra la primera tabla del problema.
7. El usuario indica que desea agregar una restricción.
8. El usuario indica el tipo de restricción.
9. El sistema agrega la restricción adecuada.

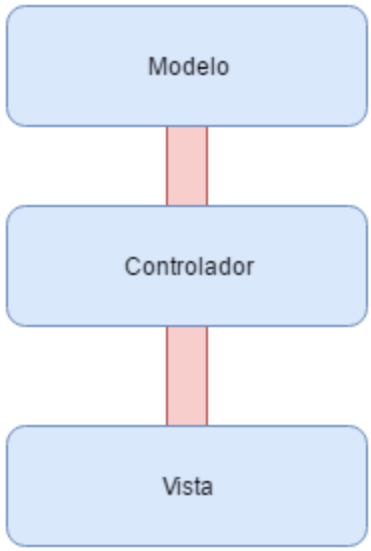
Requerimientos Funcionales

REQ-13 Agregar una restricción a un problema de programación lineal.

Una vez representado el problema en forma matricial, debe ser posible para el usuario durante el primer paso de la solución agregar restricciones mayor igual, menor igual o igual a la matriz mostrada en pantalla. Los casos igual y mayor o igual solamente serán válidos para problemas de dos fases.

Arquitectura del Sistema

La arquitectura del sistema está basado en la conocida arquitectura de N-Capas. Debido a que el programa por desarrollar no es significativamente complicado en términos de responsabilidades, cantidad de clases ni extensión se decidió realizar la división de las capas de acuerdo a la función que cumplen respecto al patrón Model-View-Controller. Esta estrategia ha sido escogida debido a que aisla las responsabilidades de cada clase dependiendo solamente de su función dentro del sistema respecto al patrón mencionado. La división lógica de las capas del sistema Simplex Educativo es la siguiente. El diagrama de clases completo puede ser encontrado en el repositorio.



- **Modelo:** Es la capa encargada de representar y manejar el comportamiento de los datos solamente. Ella responde a estímulos por parte de la capa controlador únicamente. Es en esta capa donde se encuentran todas las clases necesarias para la ejecución de los casos de uso a nivel lógico meramente, sin control por parte del usuario ni interfaz gráfica de por medio.
- **Controlador:** Capa encargada de controlar la interacción entre la vista y el modelo. Es esta capa la cual recibe los estímulos por parte del usuario y comunica al modelo la acción pedida, para luego tomar la respuesta del modelo y presentarselas a la vista para su despliegue.
- **Vista:** Capa encargada de la representación gráfica del modelo. Utiliza al controlador para hacer pedidos al modelo, y el controlador se encarga de actualizar la vista acordemente.

Diseño del sistema

Metodología del Diseño

El diseño del sistema Simplex Educativo fue realizado utilizando el paradigma de orientación a objetos. El diseño del programa fue realizado tomando en cuenta las siguientes restricciones:

Principios SOLID

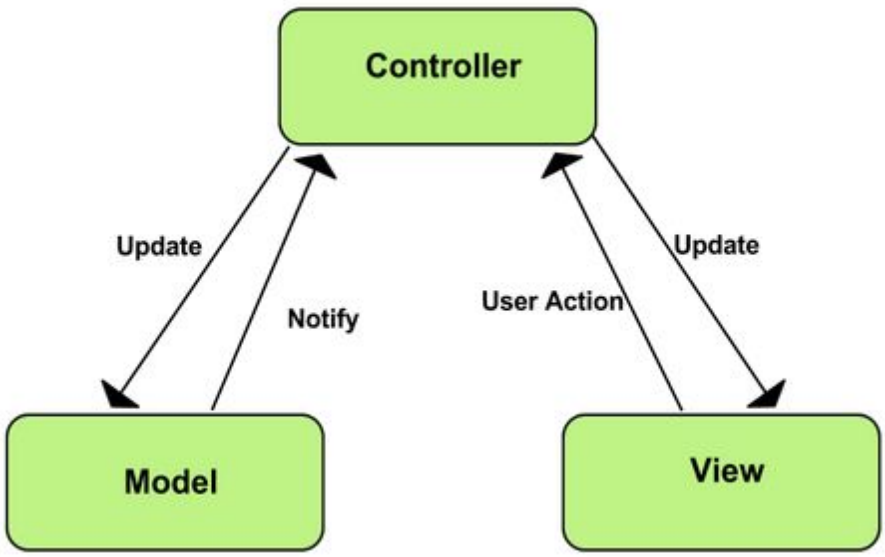
Para el diseño arquitectural se debe tomar en cuenta los principios SOLID para diseño de software. Dichos principios se describen a continuación:

- **Single Responsibility Principle:** Toda entidad de software tendrá una y solamente una responsabilidad.

- **Open-Closed principle:** Las entidades deben estar cerradas para modificación pero abiertas para extensión.
- **Liskov Substitution Principle:** Los objetos de tipo T pueden ser sustituidos por objetos de tipo S siempre que S sea un subtipo de T. Esto implica que las interfaces públicas de subclases deben ser iguales a las de su superclase.
- **Interface Segregation Principle:** Los objetos no deben depender de métodos que no utilicen.
- **Dependency Inversion Principle:** Módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Asimismo, las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones. Esto implica que las clases concretas no deben tener referencias a objetos concretos, sino a sus abstracciones.

Implementación de MVC.

Para la implementación de la solución se debe utilizar el patrón arquitectural Model-View-Controller. En este patrón existen diferentes tipos de objetos que pertenecen a alguna de las categorías anteriores: en el modelo está la lógica de negocio, en la vista las interfaces gráficas para interacción con el usuario y el controlador es el encargado de recoger las acciones del usuario en la interfaz gráfica y modificar el modelo de manera acorde. La variación del mismo será la siguiente:



Este tipo de MVC provee una mayor independencia entre el modelo y la vista, pues el modelo no debe conocer nada de la vista, de

ello se encarga el controlador, el cual la actualizará acordemente dependiendo de las acciones realizadas por el usuario.

Herramientas gratuitas.

El proyecto será realizado sin remuneración económica, por este motivo todas las herramientas que se utilicen para el desarrollo del mismo deben ser gratuitas para evitar incurrir en costos innecesarios por parte del equipo de desarrollo.

Exclusión de la eficiencia.

La eficiencia del algoritmo Simplex no es una restricción. El software será utilizado para la educación y no para la resolución de problemas complicados de programación lineal, por lo cual las restricciones de eficiencia son despreciables.

Portabilidad

La solución realizada debe ser portable hacia dispositivos independientes del sistema operativo que posean. Es por ello que se decide utilizar el lenguaje de programación Java para desarrollar el proyecto. Se debe tomar en cuenta que los objetos que se deban utilizar pertenezcan a las librerías clásicas del JDK o a librerías desarrolladas en Java completamente y que dicha librerías no sean dependientes del sistema operativo.

Estándar de Programación

Para el desarrollo de la solución se utiliza el estándar de desarrollo de código de Google. Dicho estándar puede ser accedido en el siguiente hipervínculo: https://google.github.io/styleguide/javaguide.html

Herramientad de Desarrollo de Software

NetBeans IDE 8.0.2

Para el desarrollo del proyecto se utilizó el entorno de desarrollo integrado NetBeans, el cual provee una gran cantidad de herramientas gráficas y funcionales para el desarrollo de proyectos en el lenguaje de programación Java. La plataforma ofrece servicios reusables comunes para las aplicaciones de escritorio, permitiendo a los desarrolladores centrarse en la lógica de sus aplicaciones. Además, la herramienta provee facilidades para el desarrollo de interfaces gráficas por medio

de la librería Java Swing.
Descarga: https://netbeans.org/downloads/8.0.2/

Draw.io

Herramienta libre para realizar diagramas para proyectos de desarrollo de software. Incluye vasta cantidad de íconos, líneas y componentes necesarios para realizar variedad de diagramas UML útiles para la descripción de un proyecto de software. Posee la funcionalidad de trabajo colaborativo en tiempo real para múltiples usuarios.

Acceso: https://www.draw.io/

JFlex

Para la realización del scanner que divida la cadena de texto en los diferentes tipos de tokens reconocibles se utilizó la herramienta JFlex. Esta herramienta permite definir una serie de expresiones regulares y generar código que deba ser ejecutado cuando el scanner encuentre dicha expresión regular en el texto que está analizando. Su función principal es generar el scanner que divida la cadena de entrada de texto en tokens que puedan ser utilizados por el parser para realizar su trabajo.
Documentación: http://jflex.de/manual.html

Java CUP

Para la realización del parser se utilizó la herramiento CUP. Esta herramienta trabaja en conjunto con JFlex para lograr un análisis léxico y sintáctico de una cadena de texto. Cup significa “Construction of Useful Parsers”, y es un generador de un parser LALR. El programador define los diferentes símbolos de la gramática, así como las diferentes producciones de un lenguaje libre de contexto, y este programa da la posibilidad de ejecutar cierto código cuando alguna producción es reducida, pudiendo controlar las acciones que va realizando el parser conforme se avanza por la gramática.
Documentación: http://www2.cs.tum.edu/projects/cup/docs.php

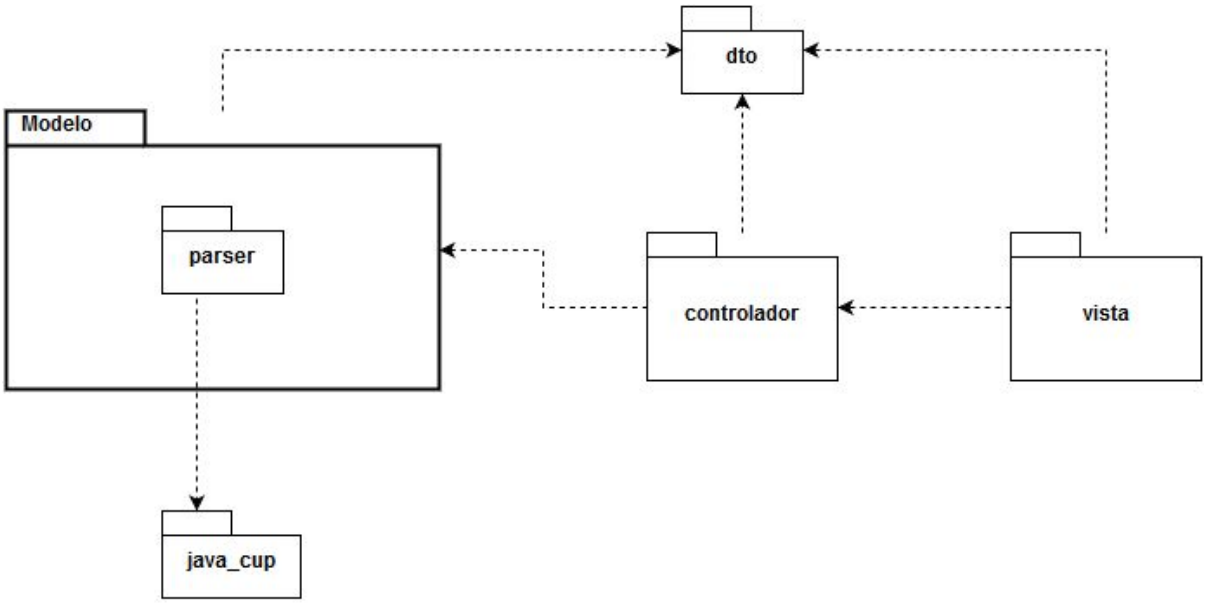
JUnit

JUnit es una biblioteca utilizadas para hacer pruebas unitarias de aplicaciones Java. JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Esta herramienta fue utilizada para realizar las pruebas unitarias y de integración de la aplicación.
Página oficial: http://junit.org/junit4/

Descomposición del Sistema

En el siguiente apartado se busca descomponer el sistema Simplex Educativo en sus subpartes para un resumen general acerca del funcionamiento y la interacción de cada componente.

Paquetes Arquitecturales



Modelo

Contiene todas las clases que representan el modelo de la solución. Es el paquete encargado del funcionamiento lógico de la aplicación, siendo él el responsable de proveer las funcionalidades necesarias para resolver los casos de uso descritos anteriormente, siendo como objeto principal solucionar problemas de programación lineal por medio del método Simplex.

El modelo es agnóstico de su controlador y la vista. Depende del paquete dto donde se almacena un Data Transfer Object, el cual es utilizado para transferir datos entre las diferentes capas arquitecturales y para el procesamiento del problema.

Modelo.parser

Almacena dentro de su contenido los componentes utilizados para comprobar y validar la estructura de la cadena de texto ingresado por parte del usuario. La importancia de la existencia de las clases

dentro del paquete rige en evitar comportamientos erróneos en la solución de un problema de programación lineal. La mayoría de las clases contenidas son autogeneradas por las herramientas Cup y JFlex que facilitan la acción objetivo del paquete, validar el formato de entrada del problema ingresado por el usuario.

Java_cup

Librería auxiliar utilizada por la clase SimplexParser dentro del paquete Modelo.parser. Java_cup brinda comportamiento de un parser LALR a la clase utilizada en la solución para validar las cadenas de texto.

Dto

Almacena objetos de transferencia de datos utilizados en los distintos paquetes de la solución creada, solo contiene una clase llamada DtoSimplex, la cual contiene toda información del problema y su representación lógica. Dicho paquete también representa al patrón de diseño Dto, con el cual se asegura que los cambios en los parámetros de los métodos de las clases causen la menor cantidad de modificaciones en cascada. Esto es de especial importancia para el mantenimiento del programa y simplifica el desarrollo, debido a que se unifica todo el estado del problema dentro de un solo objeto. Además baja los índices de rigidez y fragilidad dentro del software.

Controlador

Componente esencial del patrón de diseño MVC anteriormente descrito. Como su nombre lo indica, contiene la lógica de control y ejecución del programa. El controlador es el encargado de consumir los recursos provistos por el modelo. En él se encuentran las clases encargadas de llamar a las funciones provistas por el modelo y actualizar las clases que proveen la interfaz gráfica de la solución. Provee una interfaz que puede ser consumida por el paquete vista para cumplir con los casos de uso descritos por el cliente.

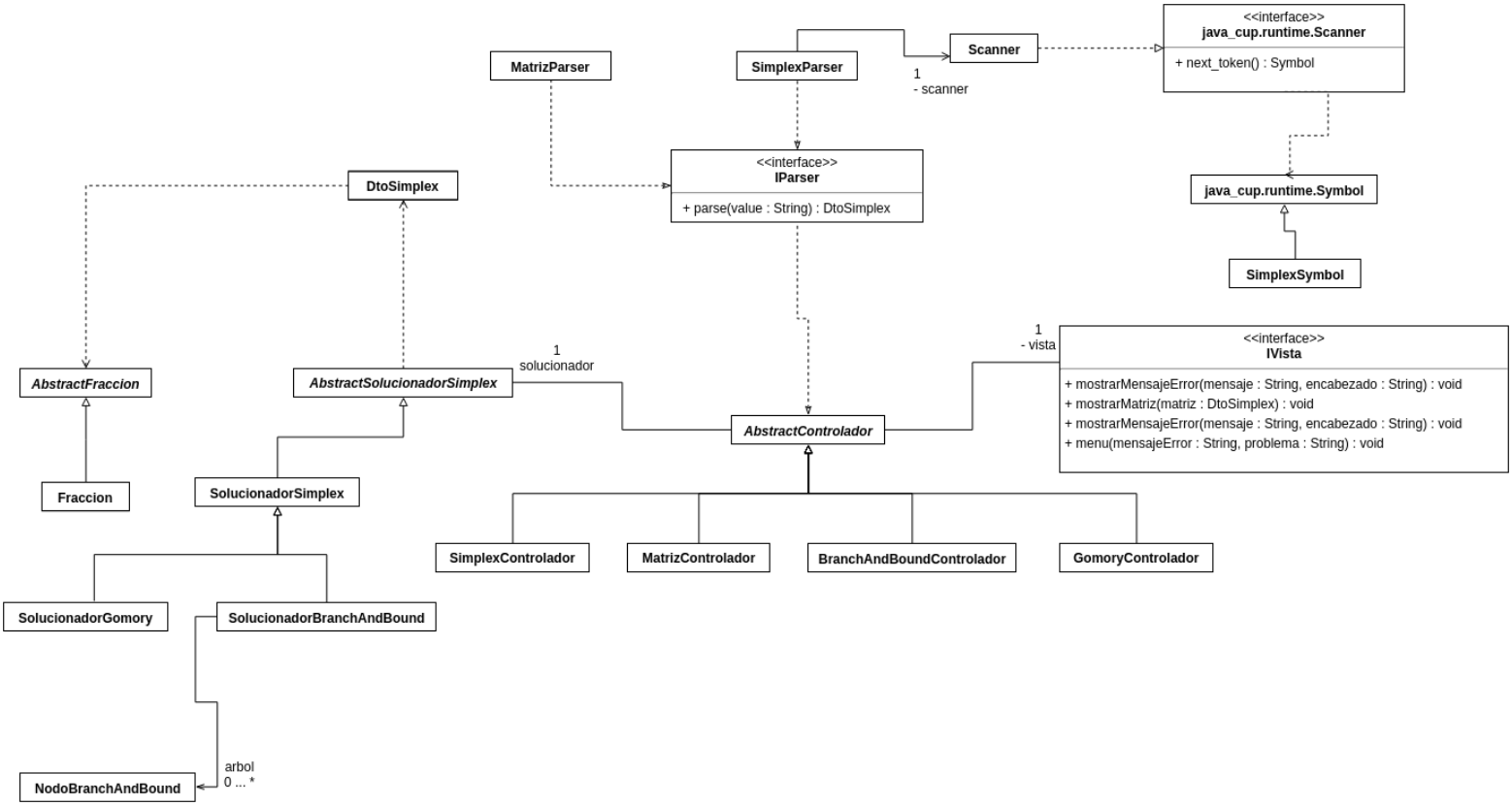
Este paquete depende del otro paquete modelo solamente.

Vista

Contiene las clases que se encargan de cumplir la funcionalidad gráfica que implementará los casos de uso solicitados por el cliente. Contiene la lógica de despliegue de la información contenida en los

objeto DTO (por ello la dependencia hacia el paquete DTO), y consume la interfaz provista por el paquete Controlador para cumplir su funcionalidad principal. En este paquete se encuentran las clases que crean la interfaz gráfica de la solución y la lógica de interacción con el usuario, peticiones, información o estado del problema ingresado.

Clases



A continuación se presenta la descripción de cada clase perteneciente al diagrama anterior, donde el lector podrá encontrar más información sobre cada clase y comprender el motivo de la inclusión de la clase dentro de la solución brindada. Se espera que el lector de la siguiente sección posea conocimientos del lenguaje de modelado unificado para poder comprender las relaciones entre clases, conceptos y estructuras presentes en el diagrama. En caso de que el lector no posea conocimientos en el mismo o desee informarse mejor puede consultar el siguiente enlace: http://www.uml.org/what-is-uml.htm

- **AbstractFraccion:**
  - **Descripción:** Definición de la clase abstracta que simboliza una fracción matemática. Posee métodos para manejar matemáticamente una representación de fracciones sin perder precisión decimal al tratar a los números reales como partes enteras, para de esta manera realizar operaciones sobre fracciones y no en punto flotante. El objetivo de agregar esta clase abstracta es la capacidad de agregar distintos comportamientos en la representación numérica del programa, permitiendo a futuros desarrolladores con nuevos

requerimientos extender fácilmente la funcionalidad solamente heredando de esta clase y programando el comportamiento que desee.
    - **Paquete:** modelo.
- **Fraccion:**
    - **Descripción:** Implementación concreta de AbstractFraccion. Dentro de esta clase se implementan los requerimientos de interpretación numérica y operaciones fraccionariass deseados por el cliente. La fracción es implementada utilizando un numerador y denominador de tipo enteros. Mediante el método toString se puede elegir el formato de representación textual que se presentará en pantalla. Este formato decimal se define en el ERS como sea fraccional (numerador / denominador) o decimal, con una cantidad de 2 dígitos en la parte punto flotante (por ejemplo, 2.50).
    - **Paquete:** modelo.
- **AbstractSolucionadorSimplex**
    - **Descripción:** Implementación abstracta de los métodos básicos necesarios para resolver un problema de programación lineal cumpliendo con los requerimientos del cliente. El motivo principal de la existencia de esta clase abstracta se basa en la existencia de distintas formas y algoritmos para resolver un problema de programación lineal. En el presente proyecto, las maneras utilizadas en total fueron solamente 3, pero en el campo de la investigación de operaciones no es un secreto la existencia de muchos más algoritmos existentes y mejores implementaciones que, a futuro, se podría agregar fácilmente sin mayor complejidad en realizar cambios en la solución gracias a esta clase.
    - **Paquete:** modelo.
- **SolucionadorSimplex:**
    - **Descripción:** Posee la implementación de los métodos de AbstractSolucionadorSimplex. Clase principal del programa, posee muchos métodos privados necesarios para la resolución mediante el método simplex, por lo cual la clase es extensa. El comportamiento de este componente se basa en la manera de resolver un problema de programación lineal utilizando el método Simplex de dos fases que el cliente, Profesor José Helo Guzmán, maneja en las clases de investigación de operaciones impartidas dentro del campus del Instituto Tecnológico de Costa Rica.
    - **Paquete:** modelo.
- **SolucionadorBranchAndBound:**
    - **Descripción:** Posee la implementación de los métodos de AbstractSolucionadorSimplex y SolucionadorSimplex. Maneja el caso

de uso de solucionar un problema mediante el algoritmo de Branch and Bound. Debido a que el algoritmo utiliza un árbol de soluciones, se implementa una clase privada NodoBranchAndBound dentro del mismo archivo .java. Esta clase principalmente está conformada por un árbol, el algoritmo trabaja sobre las hojas del mismo hasta lograr encontrar la solución óptima del problema o la infactibilidad del mismo. La descripción genérica del algoritmo se puede encontrar visitando el siguiente enlace: https://en.wikipedia.org/wiki/Branch_and_bound
    - **Paquete:** modelo.
- **NodoBranchAndBound:**
    - **Descripción:** Implementación de un árbol binario para la ejecución del algoritmo Branch and Bound. Almacena las características de un problema con todas las restricciones agregadas por el algoritmo hasta llegar al estado actual. Esta clase es privada y es únicamente utilizada por el solucionadorBranchAndBound.java.
    - **Paquete:** modelo.
- **SolucionadorGomory:**
    - **Descripción:** Posee la implementación de los métodos de AbstractSolucionadorSimplex y SolucionadorSimplex. Maneja el caso de uso de solucionar un problema mediante el algoritmo de Cortes de Gomory. El algoritmo se basa en agregar nuevas restricciones dentro del problema original, es una de las formas de resolver un problema de programación lineal sin la necesidad de crear una nueva estructura o dependencia, solamente es necesario ejecutar las veces necesarias el método Simplex de dos fases. La descripción del algoritmo de cortes de Gomory se puede encontrar en el siguiente Link: http://www.ms.unimelb.edu.au/~moshe@unimelb/620-362/gomory/
    - **Paquete:** modelo.
- **IParser:**
    - **Descripción:** Interfaz que define el método parse(String), el cual se encarga de validar el correcto formato del string de entrada y retornar en un objeto DtoSimplex los valores necesarios del algoritmo para iniciar el análisis ya sea de Simplex o de pivoteos en una matriz.
    - **Paquete:** modelo.parser.
- **SimplexParser:**
    - **Descripción:** Clase autogenerada por el programa CUP. Se encarga de validar e interpretar una cadena de texto que contenga un problema de programación lineal con el formato válido aceptado por el programa por medio de la gramática almacenada en el archivo Parser.cup, para comprender el archivo es necesario el conocimiento por parte del usuario acerca de gramáticas libres de contexto. Un comportamiento

que ejecuta esta clase es identificar el incorrecto ingreso del texto de entrada, cuando este caso es detectado se lanza una excepción.
    - **Paquete:** modelo.parser.
- **Scanner:**
    - **Descripción:** Clase autogenerada por JFLex. Se encarga de dividir el texto de entrada en tokens que puedan ser interpretados por el parser. Por ejemplo, la cadena de texto “3 x1 + ½ x2” identifica los tokens “3”, “x1”, “+”, “½ ” y “x2”. Por su parte, el Parser es el encargado de asegurarse que dichos tokens vengan en el orden correcto e interpretar el significado de la cadena de texto. La definición de los tokens se puede encontrar en el archivo Scanner.flex y es necesario el conocimiento acerca de expresiones regulares.
    - **Paquete:** modelo.parser.
- **SimplexSymbol:**
    - **Descripción:** Clase que representa un token de la cadena de texto del problema de programación lineal. Utilizado por Scanner y Parser para realizar sus operaciones. Subclase de java_cup.runtime.Symbol, y básicamente almacena información de la ubicación del Token dentro del texto ingresado.
    - **Paquete:** modelo.parser.
- **sym:**
    - **Descripción:** Posee valores enteros constantes que representan los posibles símbolos reconocidos por el scanner. Cada tipo de token tiene asignado un valor entero que está definido en esa clase. Por ser una estructura que únicamente almacena constantes no se presenta una representación dentro del diagrama de clase.
    - **Paquete:** modelo.parser.
- **java_cup.runtime.Scanner:**
    - **Descripción:** Interfaz creada con el único objetivo de ser utilizada por el SimplexPaser cuando necesita un nuevo Token, protege a la clase Scanner de un acceso a métodos que no ocupe por parte del parser. El único método necesario por el Parser es next_Token.
    - **Paquete:** modelo.parser.
- **AbstractControlador:**
    - **Descripción:** Definición de la clase abstracta que simboliza un controlador para cumplir con los casos de uso junto a una interfaz gráfica. Posee métodos para manejar la interacción entre el modelo y la vista, por lo que contiene referencias a objetos de tipo AbstractSolucionadorSimplex y IVista. Es el encargado de tomar las peticiones de IVista, realizar la acción en el modelo y actualizar la vista con el nuevo estado del modelo. El múltiple comportamiento que puede tomar el método Simplex de dos fases es donde yace la

necesidad de que existan varios controladores para ejecutar principalmente los casos de uso solicitados por el cliente donde el controlador se adapte según la acción que se encuentre realizando por parte del usuario.
    - **Paquete:** controlador**.**
- **SimplexControlador:**
    - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con solucionar un problema de programación lineal, es una clase concreta de la clase AbstractControlador. Principalmente trabaja conjunto a una instancia de SolucionadorSimplex para resolver directamente o por pasos un problema de programación lineal.
    - **Paquete:** controlador**.**
- **MatrizControlador:**
    - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con reconocer una matriz numérica y realizar operaciones pivote en ella. Es una clase concreta de la clase AbstractControlador. Los pivoteos sobre la matriz se realizan por medio de una instancia de SolucionadorSimplex, el cual es el encargado de realizar las operaciones matemáticas sobre la matriz así como otras funcionalidades solicitadas por el cliente.
    - **Paquete:** controlador**.**
- **GomoryControlador:**
    - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con solucionar un problema de programación lineal por medio del algoritmo de cortes de Gomory. Es una clase concreta de la clase AbstractControlador. Utiliza una instancia de SolucionadorGomory, con el cual puede generar el paso a paso de la solución de un problema entero o obtener la solución de manera directa.
    - **Paquete:** controlador**.**
- **BranchAndBoundControlador:**
    - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con solucionar un problema de programación lineal por medio del algoritmo de Branch and Bound, es una clase concreta de la clase AbstractControlador. Resuelve el problema de programación lineal utilizando una instancia de la clase SolucionadorBranchAndBound, donde se genera el paso a paso de la solución del problema observando por medio de un DtoSimplex el estado del árbol y las restricciones que se crearon para obtener la solución óptima entera.
    - **Paquete:** controlador**.**
- **IVista:**
    - **Descripción:** Interfaz que define los métodos que necesita una clase de interfaz gráfica para desplegar la información solicitada por el

cliente. La clase que implemente esta estructura debe ser capaz de informar por medio de una capa gráfica representativa e informativa al usuario lo que sucede dentro del algoritmo y qué acciones puede realizar.
    - **Paquete:** controlador**.**
- **Dto:**
    - **Descripción:** Data Transfer Object utilizado para resolver un problema simplex y pasar el resultado entre las diferentes capas arquitecturales. Almacena información de suma importancia para que el problema de programación lineal logre ser resuelto, contiene dentro su contenido datos con los cuales la instancia de AbstractSolucionadorSimplex se guía para aplicar acciones correspondientes sobre la matriz y variables.
    - **Paquete:** dto.

Descripción de Componentes

1. modelo.AbstractFraccion.java

1.1. Tipo

Archivo .java.

1.2. Propósito

La necesidad de poder representar valores numérico de distintas formas es donde encuentra el porqué de la existencia de este componente. El requerimiento REQ-4 solicitado, implica desarrollar distintos comportamientos para los números que se utilizan. Por ello la clase abstracta definida en este archivo nos permite adaptar el software a distintos formatos en los que se encuentra la fracción matemática y la representación decimal que utilizará los usuario según sea su selección.

1.3. Función

Definir qué métodos y atributos debe poseer las clases concretas de una fracción para representar y operar valores numéricos dentro de la lógica principal del sistema. Establece las acciones u operaciones matemáticas se realizan sobre su contenido dentro de las cuales se encuentran las cuatro operaciones básicas sumar, restar,

multiplicar y dividir, en los cuales se ejecutan modificaciones válidas para obtener un resultado preciso.Además define un comportamiento de gran importancia para el cliente, donde transforma los valores numéricos en cadenas de texto totalmente legibles para el usuario cuando consulte los datos en la capa gráfica.

Inclusive debido a otros requerimientos, se implementan métodos para obtener datos sobre las características del objeto instanciado donde se invoque la acción. Más adelante se podrá informar acerca de las propiedades de cada método y atributo que contiene este archivo.

1.4. Restricciones

Es necesario que la definición del atributo denominador sea distinto de cero, para evitar excepción de división entre cero.

La asignación valores de los atributos numerador y denominador deben encontrarse entre el rango brindado por los constantes definidas en la clase Integer, y se puede consultar por medio de acceder a los valores Integer.MAX_VALUE e Integer.MIN_VALUE.

1.5. Dependencias

- **java.io.Serializable:** la inclusión de esta clase es necesaria para leer datos en archivos que contienen información válida para la ejecución de las pruebas.

1.6. Interfaces

Los métodos definidos dentro de este componente son los siguientes:

sumar(operando : AbstractFraccion) : realiza la operación de suma de fracciones matemáticas, tomando como primer operador la fracción donde se llame el método y como segundo operando la fracción indicada por medio de parámetro. La operación se realiza por medio del siguiente método:

(Numerador1 * denominador2 **+** Numerador2*Denominador1) **/** (Denomindor1*Denominador2)

Retornando una nueva instancia de Fraccion.

- **restar(operando : AbstractFraccion) :** realiza la operación de resta de fracciones matemáticas, tomando como primer operador la fracción donde se llame el método y como segundo operando la fracción indicada por medio de parámetro. La operación se realiza por medio del siguiente método:

  (Numerador1 * denominador2 **-** Numerador2*Denominador1) **/** (Denomindor1*Denominador2)

  Retornando una nueva instancia de Fraccion.

- **multiplicar(operando : AbstractFraccion) :** realiza la operación de multiplicación de fracciones matemáticas, tomando como primer operador la fracción donde se llame el método y como segundo operando la fracción indicada por medio de parámetro. La operación se realiza por medio del siguiente método:
  (Numerador1 * Numerador2*) **/** (Denomindor1*Denominador2)
  Retornando una nueva instancia de Fraccion.
- **Dividir(operando : AbstractFraccion) :** realiza la operación de división de fracciones matemáticas, tomando como primer operador la fracción donde se llame el método y como segundo operando la fracción indicada por medio de parámetro. La operación se realiza por medio del siguiente método:
  (Numerador1 * Denominador2*) **/** (Denomindor1*Numerador2)
  Retornando una nueva instancia de Fraccion.
- **clonar() :** crea una nueva referencia o instancia del objeto tipo Fraccion donde se invoque este método. El numerador y denominador se mantienen igual a la instancia original.
- **obtenerInverso() :** crea una nueva referencia o instancia del objeto tipo Fraccion donde se invoque este método, creando el inverso multiplicativo. El numerador y denominador cambian el papel, en otras palabras en la nueva instancia el numerador de la fracción original ahora sería el denominador y el denominador sería el numerador.
- **obtenerParteDecimal() :** Retorna la parte no entera del número real que representa la fracción. Dicho resultado es encapsulado en una nueva Fraccion con el numerador siendo la parte decimal y con un denominador de 1.
- **obtenerParteEntera() :** Retorna la parte entera del número real que representa la fracción. Dicho resultado es encapsulado en una nueva Fraccion con el numerador siendo la parte entera y con un denominador de 1.
- **toString() :** retornPara elloPara elloa un valor String de la Fraccion, con el siguiente formato que lo representa "*numerador / denominador*".
- **toString(fraccional:boolean) :** retorna un valor String de la Fraccion,con un formato indicado, si el parámetro es **true** se creará un string con el formato de la función toString() indicada anteriormente y si es **false** retorna un String con el formato de un número real con coma o punto que representa la fracción.
- **iguales(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **igual** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que son iguales y **false** lo contrario.

- **mayorIgualQue(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **mayor igual** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que es mayor igual y **false** lo contrario.
- **mayorQue(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **mayor** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que es mayor y **false** lo contrario.
- **menorIgualQue(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **menor igual** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que es menor igual y **false** lo contrario.
- **menorQue(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **menor** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que es menor y **false** lo contrario.
- **hacerNegativa():** Convierte una instancia de Fraccion a su inverso de signo, negando los valores de numerador y denominador.
- **obtenerMayorDivisorComun(numero1 : double, numero2 : double) :** Obtiene el divisor mayor existente entre los números enteros indicados por medio de parámetro, utiliza el algoritmo de euclides.
- **validarSignos():** valida los signos del numerador y denominador dejando un signo negativo si fuese necesario en el numerador.

1.7. Recursos

El componente no utiliza algún recurso.

1.8. Procesamiento

Esta clase es únicamente para uso utilitario, no realiza ningún procesamiento sobre los datos o flujo del sistema.

1.9. Datos

- **Numerador:** representa la parte del numerador de una fracción matemática. Posee el tipo Integer.
- **Denominador:** representa el denominador de una fracción matemática, por defecto posee un valor de 1. Posee el tipo Integer.

2. modelo.Fraccion.java

2.1. Tipo

Archivo .java

2.2. Propósito

Define la solución concreta de los métodos abstractos de la clase AbstractFraccion, refleja en código el requerimiento REQ-4, se materializan en código fuente la implementación de los formatos numéricos ya sea Fraccionario o decimal.

2.3. Función

El objetivo principal de la creación de este componentes, es el desarrollo de los métodos abstractos de la clase AbstractFraccion, donde el comportamiento de las fracciones implicadas en el REQ-4 se definen a lo largo de su alcance. Se implementan las operaciones matemáticas que caracterizan una fracción y complementa las acciones para poder representar su datos en una cadena de texto en los formatos requeridos.

2.4. Restricciones

Esta clase implementa las mismas restricciones de su clase padre, consultar el punto 1.

2.5. Dependencias

- **java.io.Serializable:** la inclusión de esta clase es necesaria para leer datos en archivos que contienen información válida para la ejecución de las pruebas.
- **java.text.DecimalFormat:** el uso de esta librería externa se encuentra en desarrollar un formato decimal con 2 dígitos en su parte punto flotante. Consultar el método **toString**.

2.6. Interfaces

- **esCero(fraccion : AbstractFraccion) :** analiza si el valor numérico de la fracción indicada por parámetro representa al valor cero.

2.7. Recursos

Este componente no consume recursos.

2.8. Procesamiento

Esta clase es únicamente para uso utilitario, no realiza ningún procesamiento sobre los datos o flujo del sistema.

2.9. Datos

Este componente utiliza los mismos datos de la clase padre AbstractFraccion.

3. modelo.AbstractSolucionadorSimplex.java

3.1. Tipo

Archivo .java.

3.2. Propósito

Proveer los métodos necesarias para la mayoría de requerimientos, todos aquellos que impliquen resolver un problema mediante el algoritmo simplex o trabajar con representaciones numéricas matriciales. Ellos son REQ-1, REQ-3, REQ-5, REQ-6, REQ-8, REQ-9, REQ-10, REQ-11, REQ-12, REQ-13.

3.3. Función

El motivo principal de esta clase abstracta se basa en la existencia de distintas formas y algoritmos para resolver un problema de programación lineal. En el presente proyecto, las maneras utilizadas en total fueron solamente 3, pero en el campo de la investigación de operaciones no es un secreto la existencia de muchos más algoritmos y mejores implementaciones que, a futuro, se podría agregar fácilmente sin mayor complejidad en realizar cambios en la solución gracias a esta clase.

3.4. Dependencias

- **DtoSimplex:** El intercambio de información entre las diferentes capas del sistema se realiza mediante un objeto de transferencia de datos (dto). El solucionador recibe por entrada a sus métodos un dto que representa un paso de la solución del problema. Esto significa que el solucionador no poseerá atributos propios, solamente utilizará la información contenida dentro del dto para ejecutar la funcionalidad requerida.

3.5. Restricciones

- El dto recibido por parámetro en cada uno de los métodos debe estar bien formado:
    - Todos los atributos del dto deben estar inicializados.
    - Todos los valores de la matriz numérica de fracciones deben estar inicializados.

    - El dto debe indicar mediante sus atributos correctamente la información sobre los datos contenidos en ese dto. Esto significa que las variables de control dentro del dto deben reflejar verazmente el formato de los datos allí contenidos.
    - La lista con los nombres de las filas y columnas representan repectivamente las restricciones del problema y las variables asociadas a cada fila y columna de la matriz numérica. Por lo tanto, estas listas deberán tener las dimensiones apropiadas para describir la matriz numérica en su totalidad.

3.6. Interfaces

Los métodos de la clase son los siguientes:

- **solucionar(dto:DtoSimplex):** resuelve un problema de programación lineal, este método es implementado por sus clases concretas y puede actuar de distintas maneras, al final devuelve una lista de DtoSimplex, donde cada objeto representa un paso de la solución.
- **siguientePaso(dto:DtoSimplex):** genera la siguiente iteración de un problema de programación lineal, este método es implementado por sus clases concretas y puede actuar de distintas maneras, devuelve un DtoSimplex con los datos del problema en la siguiente iteración.
- **calcularRadios(**dto:DtoSimplex, columna : int**):** genera los radios según la columna indicada por medio del paramétro. Este método es implementado por sus clases concretas y puede actuar de distintas maneras, si no se define devuelve el valor generado por el método definido por su superclase. Devuelve un **String[ ]** de los radios según el formato indicado dentro del DtoSimplex.
- **completarProblema(dto:DtoSimplex):** Agrega las variables artificiales y de holgura a la matriz de coeficientes (AbstractFraccion) contenida dentro del DtoSimplex, además cambia el signo del "lado derecho" y los coeficientes de una fila si el valor del "lado derecho" es negativo. Este método debe ser implementado por sus clases concretas. Devuelve un DtoSimplex con la matriz y variables completadas.
- **siguientesOperaciones(dto : DtoSimplex):** genera un string almacenado dentro del objeto de transferencia de datos, que

indica cuales son las siguientes operaciones filas a realizar. Este método debe ser implementado por sus clases concretas. El String generado se almacena dentro del DtoSimplex para retornar.

- **agregarRestriccion(dto:DtoSimplex, tipo:int):** Agrega una restricción al problema de programación lineal ingresado por parámetro. Esta restricción puede ser mayorIgual(>=), menorIgual(<=) o igual(=), este tipo de restricción se indicará por medio del número entero y el valor de cada restricción se encuentra almacenado por defecto dentro del archivo sym.java. Este método debe ser implementado por sus clases concretas.

3.7. Procesamiento

Esta clase no posee implementación de ningún método, por lo tanto no tiene procesamiento.

3.8. Datos

La clase no tiene atributos.

4. modelo.SolucionadorSimplex.java

4.1. Tipo

Archivo .java.

4.2. Propósito

A lo largo de las características de la solución es necesario la implementación del método Simplex de dos fases. Dicho algoritmo es utilizado por parte del cliente en la enseñanza y forma parte de los requerimientos REQ-1, REQ-2, REQ-3, REQ-4, REQ-5, REQ-6, REQ-8, REQ-9, REQ-10, REQ-11, REQ-12, REQ-13 puesto a que este archivo se definen las heurísticas principales para cumplir el objetivo principal del proyecto

4.3. Función

Implementa los métodos abstractos de la clase AbstractSolucionadorSimplex, donde además se definen comportamientos para trabajar con el DtoSimplex y lograr la solución de un problema de programación lineal. La función de este componente se basa el objetivo principal de implementación del método Simplex y del cual otros componentes utilizan para solucionar problemas.

4.4. Restricciones

- El dto recibido por parámetro en cada uno de los métodos debe estar bien formado:
    - Todos los atributos del dto deben estar inicializados.
    - Todos los valores de la matriz numérica de fracciones deben estar inicializados.
    - El dto debe indicar mediante sus atributos correctamente la información sobre los datos contenidos en ese dto. Esto significa que las variables de control dentro del dto deben reflejar verazmente el formato de los datos allí contenidos.
    - La lista con los nombres de las filas y columnas representan repectivamente las restricciones del problema y las variables asociadas a cada fila y columna de la matriz numérica. Por lo tanto, estas listas deberán tener las dimensiones apropiadas para describir la matriz numérica en su totalidad.

4.5. Dependencias

- **DtoSimplex:** El intercambio de información entre las diferentes capas del sistema se realiza mediante un objeto de transferencia de datos (dto). El solucionador recibe por entrada a sus métodos un dto que representa un paso de la solución del problema. Esto significa que el solucionador no poseerá atributos propios, solamente utilizará la información contenida dentro del dto para ejecutar la funcionalidad requerida.
- **java.awt.Point:** las operaciones realizadas sobre la matriz que representa el problema de programación lineal, necesita de una estructura que integra la ubicación de las operaciones fila del algoritmo en una iteración. Dicha dependencia contiene un valor entero “x” que indica la columna de la matriz y un valor entero “y” que indica la fila.
- **modelo.parser.sym:** los métodos completarProblema y agregarRestriccion utilizan la representación numérica que se le asigna a las restricciones en el Parser. Dichos números se alojan en un archivo denominado sym.java, donde se encuentran las constantes definidas para representar restricciones.

4.6. Interfaces

- **solucionar(dto:DtoSimplex):** soluciona un problema de programación lineal y obtiene de inmediato un arreglo de DtoSimplex con el estado del problema en cada iteración. Dicho resultado se obtiene por medio de ejecutar el método.
- **_SiguientePaso** y obtener automáticamente la iteración las iteraciones. Previamente antes de ejecutar este método el DtoSimplex

que representa al problema enviado por parámetro debe de estar completado que se obtiene ejecutando el método **completaProblema**.

- **siguientePaso(dto : DtoSimplex):** genera la siguiente iteración del problema de programación lineal, invoca al método auxiliar **_SiguientePaso**.
- **_SiguientePaso(dto : DtoSimplex):** método auxiliar creado para eliminar dependencias cíclicas de llamado de métodos que se encuentra en las clases hijas que heredan de SolucionadorSimplex. Este método se encarga de validar si el problema se encuentra en la primera fase o en la segunda del método Simplex, para ejecutar el método correspondiente. Retorna un DtoSimplex con los datos del problema en la siguiente iteración.
- **calcularRadios(dto : DtoSimplex, columna : int):** genera un arreglo de tipo String, donde se calcula el radio de la columna número indicada por parámetro, por el RHS que es la última columna de la matriz que representa al problema almacenada dentro del DtoSimplex. Dichos Strings se obtienen por ejecutar el método **toString(fraccional:boolean),** de los elementos tipo AbstractFraccion de la matriz. El dato tipo fraccional siempre va a ser **false por petición del cliente.**
- **completarProblema(dto : DtoSimplex):** se encarga de agregar las columnas necesarias según el tipo de restricción indicado por parámetro dentro del DtoSimplex. Además agrega el nombre que le corresponde a una variable ya sea en la columna o si es necesario en las filas. Agrega la característica al DtoSimplex de ser o no resuelto por medio del método de dos fases y agrega la fila que representa la función -w. Inicializa la coordenada para comenzar a iterar sobre el problema. Devuelve un DtoSimplex con los todos los datos generados agregados listos para comenzar a solucionar el problema.
- **siguientesOperaciones(dto:DtoSimplex):** crea un arreglo de elementos tipo String, que representan a las operaciones fila que se realizarán según la casilla indicada por la coordenada que se encuentra como atributo dentro del DtoSimplex. Dicho arreglo se almacena dentro del DtoSimplex y es retornado.
- **generarOperacion(coeficiente : AbstractFraccion, fila1 : int, fila2 : int, fraccional : boolean, dosFases : boolean):** genera un String que representa una operación fila cuando se involucran más de una fila. El coeficiente poseerá el formato indicado por parámetro y si el problema se encuentra en la etapa de dos fases la enumeración(num) de las filas comenzará desde la 0’ continuando luego con el conteo normal. El formato del String resultante sigue el patrón siguiente:

  Coeficiente * F num(fila1) + F num(fila2) -> F num(fila2)

- **generarOperacion(coeficiente : AbstractFraccion, fila1 : int, fraccional : boolean):** genera un String que representa una operación fila cuando se involucran unicamente una fila. El coeficiente poseerá el formato indicado por parámetro, la numeración es normal no es necesario validar si se encuentra en el método de dos fases. El formato del String resultante sigue el patrón siguiente:

  Coeficiente * F num(fila1) -> F num(fila1)
- **obtenerIndiceDelValorMenor(valores: AbstractFraccion[ ], indiceInicio : int, acotoFinal : int):** Recorre un arreglo de elementos tipo AbstractFraccion comenzando desde el índice de inicio y culminando en el índice generado por cantidad de elementos menos el acotoFinal. El fin de recorrer la lista se basa en encontrar el elemento menor de la lista y retornar el índice de donde se encuentra.
- **obtenerIndiceDeSiguienteVariableSaliente(valores, AbstractFraccion[], indiceInicio : int, acotoFinal : int, dtoProblema : DtoSimplex):** Obtiene el índice del valor menor que se encuentre dentro de la lista de fracciones que representa al lado derecho, si el algoritmo se encuentra en el método de dos fases se le da prioridad a las variables artificiales.
- **obtenerRadios (fracciones : AbstractFraccion[ ] [ ], columna : int):** realiza la división entre una columna indica por parámetro y el "*lado derecho*" que se encuentra en la última posición de la matriz indicada por parámetro. Dicha división se realiza dividiendo las posiciones correspondientes entre columnas.
- **generarRadio (ladoDerecho : AbstractFraccion, columna : AbstractFraccion):** divide el elemento del *"lado derecho"* entre el de la columna. Dicha división si es menor a 0 o el denominador es 0 retorna infinito. En nuestro caso el infinito es representado por medio de una Fraccion con denominador igual a Double.maxValue.
- **pivotear(fracciones : AbstractFraccion[ ] [ ], fila1 : int, fila2 : int, valorOperador : AbstractFraccion) :** realiza la acción de pivotear del método simplex. Esta operación se ejecuta entre dos filas de la matriz y ejecutando la siguiente operación entre las posiciones correspondientes:

  (valorOperador ***** elemetofila1) **+** elementofila2
- **verificarFactibilidad(valores : AbstractFraccion[ ]) :** Verifica que si los valores de la fila indicada son todos mayores o igual que 0, el elemento que se encuentra en la última posición debe ser igual a 0. En caso de no cumplirse estas características retorna **false** indicando que no es factible y del contrario **true**.
- **esAcotado(ladoDerecho : AbstractFraccion[ ], esDosFases:boolean) :** Verifica si los elementos generados como

radios de el *“lado derecho”,* brindan la característica de que el problema está acotado o no. Este valor se obtiene comparando los elementos del radio que pertenezcan a las restricciones, por ellos se le indica si el problema se encuentra en la etapa de dos fases para omitir la primera o las dos primeras filas. Para obtener si se encuentra acotado se compara cada radio si es igual a nuestro **infinito** si existe al menos uno diferente retorna **true** y si no **false** que representa si está acotado.

- **generarUno(fila : AbstractFraccion[ ], indiceElemento : int):** crea un un uno en la posición indicada dentro del arreglo de fracciones. Para generar dicho uno se divide toda la fila de fracciones por el elemento indicado en la posición.
- **realizarOperaciones(matriz : AbstraccionFraccion[ ] [ ], indiceFilaEntrante : int, indiceColumnaEntrante : int) :** realiza todas las operaciones fila necesarias para que la variable que se encuentra en la fila y columna indicada sea variable básica. Este resultado se obtiene por medio de pivotear todas las filas por la fila entrante utilizando como coeficiente el inverso de signo del elemento en las otras filas en la columna indicada.
- **validarSimplexTerminado(funcionObjetivo : AbstraccionFraccion[ ]) :** válida si los coeficiente de la función objetivo de un problema de programación lineal se encuentra en su estado óptimo. Dicho estado se encuentra comparando los elementos del arreglo de fracciones que todos sean mayor que 0.
- **negarCoeficientes(coeficientes : AbstraccionFraccion[ ]) :** invierte el signo de las fracciones del arreglo indicado por parámetro.
- **agregarUnoMatriz(matriz : AbstraccionFraccion[ ] [ ], int fila, int columna, boolean positivo):** agrega una Fraccion con el valor de 1,en la fila y columna indicada. Dicho uno puede ser negativo o positivo dependiendo del parámetro indicado. Retorna la matriz con el uno indicado.
- **agregarColumnas(matriz : AbstraccionFraccion[ ] [ ], int cantidad) :** agrega la cantidad de columnas indicadas a la matriz. Dicha operación mantiene la última columna en su posición debido a que el algoritmo la utiliza como *“lado derecho”* y es necesario que se mantenga para realizar distintas operaciones del método de dos fases. Retorna la matriz con las columnas agregadas.
- **agregarColumnas(matriz : AbstraccionFraccion[ ] [ ], cantidad : int, posicion : int) :** agrega la cantidad de columnas indicadas a la matriz comienza la adicción en la posición indicada. Dicha operación mantiene la última columna en su posición debido a que el algoritmo la utiliza como *“lado derecho”* y es necesario que se mantenga para

realizar distintas operaciones del método de dos fases. Retorna la matriz con las columnas agregadas.

- **agregarUnos(matriz : AbstraccionFraccion[ ] [ ], posiciones : ArrayList<Integer>, positivo : boolean, inicio : int) :** agrega unos utilizando la función agregarUno, en las filas que se encuentran en el arreglo de posiciones y columnas secuencialmente iniciando desde el parámetro inicio, dichos unos se indicará el signo por medio del parámetro booleano. Retorna una matriz con los unos agregados.
- **convertirDosFases(matriz : AbstraccionFraccion[ ] [ ], int inicioArtificiales):** convierte la estructura de la matriz del problema de programación lineal en un problema de dos fases. Agrega la fila que representa a la función w conjunto a sus variables artificiales. Retorna la matriz convertida a dos fases.
- **crearFuncionW(tamano : int, inicioArtificiales : int) :** crea un arreglo de elementos tipo Fraccion de un tamaño indicado que representa a la función w. Dicho arreglo se le agregan los unos de las variables artificiales iniciando desde el índice indicado.
- **generarCoeficiente(matriz:AbstractFraccion[][], indiceFila : int, indiceColumna : int):** crea un coeficiente tomando el elemento que se encuentra en la fila y columna indicado. Dicho elemento se le invierte el signo y es retornado como coeficiente.
- **siguientePivoteo(dto : DtoSimplex):** crea el siguiente punto donde se realizará un pivoteo, busca la columna que contenga en la función objetivo el valor menor y el radio de la restricción con valor menor. Con los índices encontrados genera un Punto que representará las siguientes operaciones y variable entrante.
- **agregarNombreVariables(nombres : String[], cantidad : int, nuevoNombre : String):** agrega una cantidad de nombres a un arreglo de elementos tipo String. Los nuevos nombres son generados por medio del siguiente patrón: **nuevoNombre+indiceColumna**. Retorna un nuevo arreglo con los nombres agregados.
- **agregarNombresFila(nombres : String[], Indicesfila : ArrayList<Integer>, nombreFila : String, indiceInicio : int) :** crea los nombres de las filas que representan a las variables seleccionadas. Dichos nombres se crean por medio del sufijo indicado por parámetro y el índice de incio. A cada fila se le agrega el nombre correspondientemente se encuentre en el arreglo de índices.
- **agregarNombreW(nombres : String []):** agrega al inicio del nombre de las filas, el String “-w” que representa a la función objetivo de un problema de dos fases.
- **crearNombreFila(tamano : int, nombreCabeza :String):** Inicializa el arreglo de elementos tipo String que representan a los nombres de los

nombres de fila, se le asigna el nombre “z” y “-z” dependiendo del tipo de problema.

- **siguientePasoSimplex(dto : DtoSimplex) :** realiza las operaciones de la iteración actual del problema de programación lineal dentro del DtoSimplex. Para ejecutar un siguientePaso es necesario que el problema se encuentre completado. Realiza las siguientes operaciones según el punto dentro del DtoSimplex. Valida si el problema fue terminado y se encuentra en estado óptimo. Antes de iterar se valida si el problema esta acotado. El método cerciora si el problema es de dos fases para indicar cuando termina una etapa. Devuelve un DtoSimplex con los datos de la iteración.
- **siguientePasoDosFases(dto : DtoSimplex) :** se encarga de ejecutar las acciones correspondientes cuando el problema se encuentra en la sección de dos fases. Si se encuentra eliminando las variables artificiales del método de dos fases ejecuta una acción distinta a cuando se continúa con el flujo normal de terminar la primera fase. También valida si el problema es factible que es la sección donde sucede. Devuelve un DtoSimplex con los datos de la iteración.
- **eliminarArtificiales(dto : DtoSimplex):** método invocado al inicio de la primera fase del simplex de fases, se encarga de eliminar las variables artificiales que poseen un uno en la función objetivo. Genera un punto en la fila donde se encuentra seleccionada la variable artificial de la siguiente iteración.Válida cuando se eliminaron todos los unos que representan a las variables artificiales para continuar con el método normal del Simplex. Devuelve un DtoSimplex con los datos después de la iteración.
- **buscarIndice(nombres : String[], nombre : String):** busca un String dentro de un arreglo de elementos del mismo tipo y retorna el índice donde se encuentra. Si no lo encuentra retorna -1.
- **eliminarFilaW(dto : DtoSimplex):** Elimina la primera fila de la matriz que se encuentra dentro del DtoSimplex. Esta fila representa la función w del método de dos fases y se utiliza cuando termina la primera etapa. Además elimina del nombre de las filas el -w.
- **eliminarColumnasArtificiales(dto : DtoSimplex):** elimina de la matriz que se encuentra dentro del DtoSimplex, las columnas que representan a las variables artificiales. Este método se utiliza cuando termina la primera etapa del Simplex de dos fases y retorna la matriz almacena dentro del DtoSimplex.
- **eliminarNombreW(nombresFila : String[]):** Toma el arreglo de elementos tipo String y elimina el primer elemento que representa a la variable -w. Este método se utiliza cuando termina la primera etapa del

Simplex de dos fases y retorna un nuevo arreglo sin el primer elemento.

- **eliminarNombresColumnas(nombresColumnas : String[], indiceInicioArtificiales : int):** elimina del arreglo de elementos tipo String que representa al nombre de las columnas, los elementos que representan las columnas de las variables artificiales que inician desde el índice indicado. Este método se utiliza cuando termina la primera etapa del Simplex de dos fases y retorna un nuevo arreglo sin el nombre de las artificiales.
- **siguientesOperacionesInicioDosfases(indice :int):** genera un String que representa una operación de fila que se realiza al inicio del método de dos fases, cuando se eliminan las variables artificiales. El String retornado posee el siguiente formato: "**-** F inidice + F0' -> F0' "
- **obtenerSolucion(dto : DtoSimplex):** Retorna un dato tipo String con el valor representativo de la solución del problema de programación lineal. Solo incluye las variables de la función objetivo original.
- **agregarColumna(dto:DtoSimplex, posicion: int):** agrega una columana de elementos tipo AbstractFraccion a la matriz indicada por parámetro. Dicha fila solo contiene valores de 0 y se ubica en la posición indicada..
- **agregarFila(matriz : AbstractFraccion[][]):** agrega una fila de elementos tipo AbstractFraccion a la matriz indicada por parámetro. Dicha fila solo contiene valores de 0.
- **agregarRestriccion(dto : DtoSimplex, tipo : int):** agrega una nueva restricción a la matriz de elementos tipo AbstractFraccion. El tipo int dependerá de los definidos dentro de las constantes en el archivo sym.java.
- **agregarMenorIgual(dto : dtoSimplex):** agrega una restricción tipo menor o igual (<=) a la matriz de elementos tipo AbstractFraccion. Actualiza la cantidad de columnas y las variables ligadas al número de variables de holgura y artificiales dentro del DtoSimplex y lo retorna.
- **agregarMayorIgual(dto : dtoSimplex):** agrega una restricción tipo menor o igual (>=) a la matriz de elementos tipo AbstractFraccion. Actualiza la cantidad de columnas y las variables ligadas al número de variables de holgura y artificiales dentro del DtoSimplex y lo retorna.
- **agregarIgual(dto : dtoSimplex):** agrega una restricción tipo menor o igual (=) a la matriz de elementos tipo AbstractFraccion. Actualiza la cantidad de columnas y las variables ligadas al número de variables de holgura y artificiales dentro del DtoSimplex y lo retorna.

4.7. Procesamiento

Se describen el flujo de ejecución de los métodos públicos:

- **completarProblema(dto:DtoSimplex):**
    1. Se inicializan los arreglos de elementos tipo Integer donde se almacenarán índices de las filas de la matriz, que se asignarán a las variables de holgura, artificiales y holgura negativas dentro del problema.
    2. Se compara si el problema es de maximización.
        a. Si es de maximización se niegan los coeficientes por medio del método **negarCoeficientes** de la primera fila de la matriz y se agrega como nombre de fila “z” al primer elemento de la lista nombreFilas que se encuentra dentro del DtoSimplex utilizando la función **crearNombreFila**.
        b. Si no es de maximización se agrega como nombre de fila “-z” al primer elemento de la lista nombreFilas que se encuentra dentro del DtoSimplex utilizando la función **crearNombreFila**.
    3. Se recorre el arreglo de desigualdades denominado como listaDesigualdades ubicado en el DtoSimplex, donde cada fila n de la matriz que representa una restricción del problema posee un dato en el arreglo ubicado en la posición n-1. La primera fila se omite por no ser una restricción. Antes de comenzar cada iteración del for se obtiene el elemento del lado derecho que se encuentra al final de cada fila.
        a. Si la restricción de la fila n, es igual a la constante sym.MAYORIGUAL.
            i. Si el elemento del lado derecho es menor que cero, se niegan los coeficientes de la fila n con la función **negarCoeficientes** y se agrega el índice de la fila en el arreglo de holguras positivas.
            ii. En el caso contrario, se agrega el índice al arreglo de holguras negativas y al de artificiales.
        b. Si la restricción de la fila n, es igual a la constante sym.MENORIGUAL.
            i. Si el elemento del lado derecho es menor que cero, se niegan los coeficientes de la fila n con la función **negarCoeficientes** y se agrega el índice al arreglo de holguras negativas y al de artificiales.

        ii. De lo contrario, se agrega el índice al arreglo de holguras positivas.
    c. Si la restricción de la fila n, es igual a la constante sym.IGUAL
        i. Si el elemento del lado derecho es menor que cero, se niegan los coeficientes de la fila n con la función **negarCoeficientes.**
        ii. Se agrega el índice al arreglo de variables artificiales.
4. Se contabiliza cuantas nuevas columnas se necesitan agregar a la matriz. Luego se agregan las columnas por medio del método **agregarColumnas**.
5. Se agregan los unos positivos en la matriz, en los índices de las filas que se encuentra en el arreglo de holguras positivas, utilizando el método **agregarUnos**.
6. Se agregan los unos negativos en las matriz, en los índices de las filas que se encuentran en el arreglo de holguras negativos, utilizando el método **agregarUnos**.
7. Se agrega al DtoSimplex, la cantidad de variables de holgura que se encuentra en el atributo **variablesHolgura**.
8. Se agregan los unos positivos en la matriz, en los índices de las filas que se encuentra en el arreglo de variables artificiales, utilizando el método **agregarUnos**.
9. Se agregan las variables de holgura y artificiales al arreglo de nombreColumnas ubicado en el DtoSimplex, por medio del método **agregarNombreVariables**.
10. Se agregan las variables de holgura y artificiales al arreglo de nombreFilas ubicado en el DtoSimplex, por medio del método **agregarNombresFila**.
11. Se compara la cantidad de variables artificiales que se necesitaron para completar el problema.
    a. Si es mayor que cero.
        i. Se agrega una fila a la matriz por medio del método **convertirDosFases**, donde se agrega los unos de las variables artificiales en la nueva función objetivo del problema.
        ii. Se establece la bandera de algoritmo solucionado por medio de dos fases en true.
        iii. Se establece el bloqueDosFases en true en el DtoSimplex.
        iv. Se establece la variable artificial actual que s e va eliminar ,en el DtoSimplex.

v. Se agrega el nombre “-w”, a arreglo de nombres fila del DtoSimplex por medio de la función **agregarNombreW**.
vi. Se establece la siguiente coordenada de pivote según el algoritmo de dos fases por medio del método **siguientesOperacionesInicioDosfases**.
vii. Se agrega el mensaje representativo a los cambios sucedidos.

b. De lo contrario.
i. Se establece la siguiente coordenada de pivote según el algoritmo genérico Simplex por medio de la función **siguientePivoteo**.
ii. Se agrega el mensaje representativo a los cambios sucedidos.

12. Se retorna un DtoSimplex con los cambios sucedidos sobre sus atributos y principalmente a la matriz contenida.

- **calcularRadios(dto:DtoSimplex, columna: int)**
  1. Obtiene los radios utilizando la columna con el número indicado y la matriz contenida dentro del DtoSimplex. Para dicho proceso se utiliza el método **obtenerRadios**.
  2. Recorre el arreglo de radios tipo AbstractFraccion generados obteniendo su valor representativo tipo String con la bandera de formato en false, creando un nuevo arreglo tipo String.
  3. Retorna el arreglo creado con elementos tipo String.
- **siguientesOperaciones(dto:DtoSimplex):**
  1. Obtiene de los datos del DtoSimplex la matriz de coeficientes, la coordenada para realizar operaciones y la casilla seleccionada para realizar las operaciones fila según la coordenada.
  2. Válida si se encuentra resolviendo un problema de dos fases.
     a. Si se encuentra resolviendo un problema de dos fases
        i. Reduce el índice de la fila con el cual se genera el String de la operación fila.
        ii. Genera una operación fila donde representa un afectamiento sobre la misma fila utilizando la función **generarOperacion(coeficiente : AbstractFraccion, fila1 : int, fraccional : boolean)**.
     b. Sino.
        i. Genera una operación fila donde representa un afectamiento sobre la misma fila utilizando la función **generarOperacion(coeficiente :**

AbstractFraccion, fila1 : int, fraccional : boolean) donde no se reduce el índice de la fila.

3. Recorre con un contador que no puede ser máximo a la cantidad de filas de la matriz, creando el String representativo a la operación fila correspondiente. Donde se valida que el índice (**indiceOperacion**) a almacenar el nuevo String no sobreescriba el primer dato almacenado al inicio de la ejecución del método
    a. Genera el String de cada operación fila para todas las filas de la matriz utilizando el método **generarOperacion(coeficiente : AbstractFraccion, fila1 : int, fila2 : int, fraccional : boolean).**
4. Agrega al DtoSimplex el arreglo de String generado y lo retorna.

- **siguientePaso(dto : DtoSimplex):**
    1. Ejecuta el método auxiliar para evitar dependencias cíclicas entre llamadas de clases hijas llamado **_SiguientePaso.**
    2. Retorna el DtoSimplex generado por el paso anterior.
- **solucionar(dto:DtoSimplex):**
    1. Valida si el problema contenido dentro del DtoSimplex es factible, se encuentra acotado para poder ejecutar el ciclo.
    2. Se agrega al arreglo de iteraciones de elementos tipo DtoSimplex el dto actual.
    3. Mientras que el problema este acotado, sea factible y no haya finalizado:
        a. Se clona el DtoSimplex por medio del método **clonarProfundo**.
        b. Se obtiene la siguiente iteración del problema ejecutando el método **_SiguientePaso** con el DtoSimplex clonado.
        c. Agrega el resultado del paso anterior al arreglo de iteraciones.
    4. Retorna el arreglo con los DtoSimplex que representan a las iteraciones necesitadas para resolver el problema.
- **agregarColumna(dto:DtoSimplex, posicion: int):**
    1. Actualiza el arreglo de nombre columnas que se encuentra dentro del DtoSimplex. Agrega como nuevo nombre “-”.
    2. Agrega la nueva columna a la matriz contenida dentro del DtoSimplex por medio del método **agregarColumnas**.
    3. Agrega la matriz y los nombre de las columnas al DtoSimplex.
    4. Retorna el DtoSimplex.
- **agregarFila(matriz : AbstractFraccion[][]):**
    1. Actualiza el arreglo de nombreFilas que se encuentra dentro del DtoSimplex. Agrega como nuevo nombre “-”.

2. Agrega la nueva fila a la matriz contenida dentro del DtoSimplex por medio del método **agregarFila**.
3. Agrega la matriz y los nombre de las filas al DtoSimplex.
4. Retorna el DtoSimplex.

- **agregarRestriccion(dto : DtoSimplex, tipo : int):**
  1. Clona el DtoSimplex actual por medio del método **clonarProfundo**.
  2. Ejecuta un método distinto apoyándose del tipo de restricción indicada por parámetro y las constantes dentro del archivo sym.java donde se le asigna a cada restricción una constante.
     a. Si el tipo indicado es igual a la constante sym.MAYORIGUAL, ejecuta el método **agregarMayorIgual** con el DtoSimplex clonado.
     b. Si el tipo indicado es igual a la constante sym.MENORIGUAL, ejecuta el método **agregarMenorIgual** con el DtoSimplex clonado.
     c. Si el tipo indicado es igual a la constante sym.IGUAL, ejecuta el método **agregarIgual** con el DtoSimplex clonado.
  3. Retorna el resultado obtenido tras ejecutar alguno de los tres casos anteriores.

4.8. Datos

Esta clase no contiene atributos.

5. modelo.SolucionadorBranchAndBound.java

5.1. Tipo

Archivo .java.

5.2. Propósito

Una manera de resolver un problema de programación lineal y que los datos que brinde como resultado sean enteros, es por medio del algoritmo de Branch and Bound, los requerimientos REQ-11 y REQ-12 abarcan el desarrollo de este algoritmo donde se pueda generar todas las iteraciones necesarias que se realizan sobre el árbol que se produce como resultado al usuario.

5.3. Función

La función de esta clase es implementar la lógica necesaria para resolver un problema de programación lineal mediante el algoritmo de Branch and Bound Extiende de la clase SimplexControlador y sobreescribe los métodos solucionar y siguientePaso para comportarse de acuerdo al algoritmo de Branch and Bound.

5.4. Restricciones

- El dto recibido por parámetro en cada uno de los métodos debe estar bien formado:
    - Todos los atributos del dto deben estar inicializados.
    - Todos los valores de la matriz numérica de fracciones deben estar inicializados.
    - El dto debe indicar mediante sus atributos correctamente la información sobre los datos contenidos en ese dto. Esto significa que las variables de control dentro del dto deben reflejar verazmente el formato de los datos allí contenidos.
    - La lista con los nombres de las columnas que representan repectivamente las variables asociadas a cada fila y columna de la matriz numérica. Por lo tanto, estas listas deberán tener las dimensiones apropiadas para describir la matriz numérica en su totalidad.
    - El problema que representa el DtoSimplex, no debe de llegar completado por medio del método completarProblema, debido a que el algoritmo agrega restricciones al árbol por su naturaleza y modifica al problema original, por ello el encargado de completar el problema es dentro del método siguientePaso de este componente.

5.5. Dependencias

- **DtoSimplex:** El intercambio de información entre las diferentes capas del sistema se realiza mediante un objeto de transferencia de datos (dto). El solucionador recibe por entrada a sus métodos un dto que representa un paso de la solución del problema. Esto significa que el solucionador no poseerá atributos propios, solamente utilizará la información contenida dentro del dto para ejecutar la funcionalidad requerida.
- **modelo.parser.sym:** el método crearHijos utiliza la representación numérica que se le asigna a las restricciones en el Parser. Dichos números se alojan en un archivo denominado sym.java, donde se encuentran las constantes definidas para representar restricciones.
- **modelo.NodoBranchAndBound:** dicha clase es privada e

implementa el árbol de soluciones necesario para desarrollar el algoritmo de Branch and Bound.

5.6. Interfaces

- **solucionar(dto:DtoSimplex):** itera las veces posible sobre el problema de programación lineal ingresado hasta llegar a una solución o encontrar que el problema no es factible. El arreglo de elementos tipo DtoSimplex contiene el estado del árbol como solución y un mensaje con lo sucedido en el árbol.
- **siguientePaso(dto:DtoSimplex):** itera un nodo hoja para generar dos nuevas soluciones. También se encarga de verificar si el árbol se encuentra en la primera iteración, en este caso solamente genera un nodo hoja. Verifica si el problema se soluciono con valores enteros y si existe una solución factible. Cada iteración llama al método **solucionar** de su clase padre **SolucionadorSimplex,** con la cual se crea una nueva iteración del problema.
- **acotarNodos(tipoProblema : boolean):** ejecuta una búsqueda de nodos tipo NodoBranchAndBound dentro de las hojas del árbol, donde no se encuentren acotados su solución y sean no enteras. Esto con el fin de comparar con las soluciones enteras existentes y verificar si se puede acotar un nodo. Un nodo se acota dependiendo del tipo de problema en que se encuentre ya sea de minimización o maximización, si es de **maximización** se acota si existe una solución entera con un valor z mayor o igual y si fuese **minimización** sería con un valor z menor o igual.
- **buscarNodos(esEntera : boolean) :** realiza una búsqueda dentro del arreglo de hojas del árbol del problema que se ha generado. El fin de esta búsqueda es encontrar nodos con solución factible y acotada, y además si su solución es entera o no dependiendo del valor del parámetro esEntera, si es **true** la solución es entera y del contrario es **false**.
- **generarNuevoNodo(padre : NodoBranchAndBound, problema : DtoSimplex, solucion : DtoSimplex, indiceProblema : String) :** genera una nueva hoja tipo NodoBranchAndBound de un problema anterior, al nodo generado se le asigna el valor z, el valor de las variables objetivo del problema, el índice del problema y además se verifica si el nuevo nodo posee una solución factible.
- **obtenerValorZ(matriz : AbstractFraccion[][]):** retorna el valor solución del problema actual, recibe la matriz de fracciones y retorna el valor de la función objetivo, el cual se encuentra en la última posición de la primera fila.

- **obtenerValorVariables(problema : DtoSimplex):** obtiene la cantidad de variables objetivo del problema y busca acorde la fila donde se encuentre seleccionada y su valor derecho. Dichos valor son devueltos como el valor de la variable y según como se encuentre ordenadas las filas se le asigna un valor en el arreglo de Fraccion que se va a retornar, si una variable no se encuentra seleccionada su valor es dado por 0.
- **haySolucionFactible() :** recorre el arreglo de hojas del árbol verificando si existe una solución factible. Si hay al menos una hoja con solución factible retorna **true** y si no **false**.
- **validarBranchAndBoundTerminado():** recorre el arreglo de hojas del árbol verificando si existe aún un nodo con una solución no entera factible y que no se encuentre acotada.Si encuentra un nodo con las características anteriores retorna **false** y de los contrario **true**.
- **buscarNodo(tipoProblema : boolean, esSolucionEntera : boolean):** Busca dentro del arreglo de hojas, el índice del siguiente nodo con solución de z mejor entera o no. Dicho nodo dependerá del tipo de problema que se indique, se retorna el nodo con mayor valor z si el problema es de maximización y el menor si el problema es de minimización.
- **buscarNodoMenorZ(esSolucionEntera : boolean):** busca dentro del arreglo de hojas del árbol generado, el índice de la hoja con el menor valor z. Dicho valor no debe de estar acotado y ser factible. Además el valor z podría ser entera o no dependiendo del parámetro esEntera donde **true** indica que sea entera de lo contrario **false**.
- **buscarNodoMayorZ(esSolucionEntera : boolean):** busca dentro del arreglo de hojas del árbol generado, el índice de la hoja con el mayor valor z. Dicho valor no debe de estar acotado y ser factible. Además el valor z podría ser entera o no dependiendo del parámetro esEntera donde **true** indica que sea entera de lo contrario **false**.
- **generarNuevosProblemas(indiceSiguienteNodo : int):** se encarga de generar los dos nuevos nodo hijos de un nodo seleccionado, se indica el nodo por medio del índice pasado por parámetro. Además se encarga de obtener la variable con la cual se realizarán las nuevas restricciones de las ramas del árbol.
- **obtenerParteDecimal(valorVariables : AbstractFraccion[]) :** separa la parte decimal de los valores numéricos reales que representan las fracciones que se encuentran en el arreglo indicado. Devuelve un arreglo donde se encuentran únicamente la parte decimal de cada valor respectivamente.
- **obtenerIndiceDelValorMayor(valores : AbstractFraccion[], indiceInicio : acotoFinal : int):** Recorre un arreglo de elementos tipo

AbstractFraccion comenzando desde el índice de inicio y culminando en el índice generado por cantidad de elementos menos el acotoFinal. El fin de recorrer la lista se basa en encontrar el elemento mayor de la lista y retornar el índice de donde se encuentra.

- **crearHijos(siguienteNodo : NodoBranchAndBound, indiceVariable : int, valorSeleccionado : AbstractFraccion):** agrega las nueva restricciones que implica la iteración en el problema generado hasta el nodo actual. Con ello se crea dos nodos nuevos donde se resuelve el problema creado y se agrega el valor del z, el valor de las variables y verifica si el problema es factible o no. Elimina el nodo padre y agrega los dos nuevos nodos generados al arreglo de hojas del árbol.
- **agregarRestriccionFinal(problema : DtoSimplex, indiceVariable : int, valorLadoDerecho : AbstractFraccion, tipoRestriccion : int):** agrega una nueva fila a la matriz contenida dentro del DtoSimplex. Donde se crea una nueva restricción dependiendo del tipo indicado por parámetro, dichos tipos se encuentran en el archivo sym.java. Agrega un uno en la posición de la variable seleccionada y actualiza la lista de desigualdades dentro del DtoSimplex.
- **agregaDesigualdad(listaDesigualdades : int[], tipoRestriccion : int):** actualiza la lista de desigualdades que representan a las restricciones de un problema de programación lineal. Agrega al final de la lista el valor representativo entero de la nueva restricción.

5.7. Recursos

Esta clase no utiliza ningún recurso.

5.8. Procesamiento

Se describen el flujo de ejecución de los métodos públicos:

- **solucionar(dto:DtoSimplex):**
    1. Se crea un arreglo vacío para agregar las iteraciones.
    2. Mientras el problema sea factible y no este finalizado:
        a. Se ejecuta el método de esta clase para obtener la siguiente iteración, **siguientePaso**.
        b. Se agrega la iteración al arreglo inicializado al inicio.
    3. Se retorna el arreglo con las iteraciones generadas para resolver el problema.
- **siguientePaso(dto:DtoSimplex):**
    1. Se definen las banderas que indicarán que el problema es factible o finalizado.
    2. Se compara si el atributo árbol es nulo.
        a. Si es nulo.

i. Se completa el primer problema ingresado, utilizando el método **completarProblema**, el DtoSimplex utilizado es clonado por medio del método **clonarSinCompletarProfundo**, donde se copian las referencias que se encuentran inicializadas antes de agregar los datos respectivos en el método de completar.
ii. Se genera la solución del primer problema invocando al método de **solucionar** de su clase padre, dicha solución se encuentra en la última posición del arreglo de iteraciones resultantes, y se almacena en la variable problema.
iii. Se crea el nodo raíz del árbol donde se inicializa con la referencia al padre en nulo, la solución obtenida en el paso anterior y con el índice de 1.
iv. Se almacena como árbol el nodo creado y se agrega a los nodos hojas.

b. De lo contrario.
i. Se obtiene el índice del nodo hoja siguiente a iterar, donde por medio del método **buscarNodo,** se obtiene el nodo correspondiente al tipo de problema ya sea de maximización o minimización y que su no sea entera, indicado por medio del parámetro false.
ii. Se generan los dos nuevos problemas, utilizando el índice del nodo siguiente y el método **generarNuevosProblemas**.
iii. Se acotan los nodos hojas que no se pueden desarrollar más debido a que existe una solución mejor existente dependiendo del tipo de problema. Dicha operación se realiza por medio del método **acotarNodos**.

3. Se obtiene si el problema se encuentra finalizado por medio del método **validarBranchAndBoundTerminado**.
4. Se verifica que exista una solución factible por medio del método **haySolucionFactible**.
5. Se compara si el problema esta finalizado y que exista solución factible.
a. Si se cumple.
i. Se busca el nodo hoja con la solución óptima entera para el tipo de problema, utilizando el método **buscarNodo**.

ii. Se completa el mensaje a mostrar por medio de los métodos de la clases NodoBranchAndBound obtenerTodasRestricciones y valorVariables.
iii. Se agrega al nodo óptimo la bandera de óptimo en true.

b. De lo contrario.
    i. Se compara si el problema es factible
        1. Si no es factible.
            a. Se crea un mensaje para indicar que el problema no posee solución óptima entera factible.
        2. Si es factible.
            a. Se obtiene el índice del nodo hoja siguiente a iterar, donde por medio del método **buscarNodo,** se obtiene el nodo correspondiente al tipo de problema ya sea de maximización o minimización y que su no sea entera, indicado por medio del parámetro false.
            b. Se busca la variable con la parte punto flotante mayor contenido dentro del nodo anteriormente buscado. Para realizar dicho proceso se utiliza el método **getValorVariables** del NodoBranchAndBound, **obtenerParteDecimal** y **obtenerIndiceDelValorMayor.**
            c. Se obtiene el String del nombre de la variable ubicado en el arreglo de nombre columnas. Y se crea el mensaje para la siguiente iteración sobre el árbo.

c. Se crea un nuevo DtoSimplex, donde se agrega el mensaje, el texto del árbol generado hasta el estado actual ejecutando el método **toStringRepeat**, si el problema está finalizado y si es factible.
d. Se retorna el DtoSimplex generado.

5.9. Datos

- **arbol : NodoBranchAndBound:** referencia al nodo raíz del árbol generado por el algoritmo de branch and bound.
- **hojas : ArrayList<NodoBranchAndBound>:** arreglo dinámico de nodos que representan a las hojas del árbol. Se creó esta atributo para evitar búsquedas de hojas por cada iteración en el árbol.

6. modelo.NodoBranchAndBound.java

6.1. Tipo

Archivo .java.

6.2. Propósito

Una manera de resolver un problema de programación lineal y que los datos que brinde como resultado sean enteros, es por medio del algoritmo de Branch and Bound, los requerimientos REQ-11 y REQ-12 abarcan el desarrollo de este algoritmo donde se pueda generar todas las iteraciones necesarias que se realizan sobre el árbol que se produce como resultado al usuario. Esta última característica es donde rige la importancia de crear esta clase, gracias a esta clase se representa el árbol de soluciones del algoritmo.

6.3. Función

Implementar las características de un árbol binario donde se almacene el problema de programación lineal con las restricciones agregadas hasta el momento y la solución con dichas características. Sirve como estructura de apoyo para el SolucionadorBranchAndBound.

6.4. Restricciones.

- El dto recibido por parámetro en cada uno de los métodos debe estar bien formado:
    - Todos los atributos del dto deben estar inicializados.
    - Todos los valores de la matriz numérica de fracciones deben estar inicializados.
    - El dto debe indicar mediante sus atributos correctamente la información sobre los datos contenidos en ese dto. Esto significa que las variables de control dentro del dto deben reflejar verazmente el formato de los datos allí contenidos.
    - La lista con los nombres de las filas y columnas representan repectivamente las restricciones del problema y las variables asociadas a cada fila y columna de la matriz numérica. Por lo

tanto, estas listas deberán tener las dimensiones apropiadas para describir la matriz numérica en su totalidad.

6.5. Dependencias

No posee dependencias a alguna otra estructura.

6.6. Interfaces

- **esSolucionEntera() :** verifica que el valor de todas las variables que forman parte de la función objetivo poseen valores enteros. Si se cumple esta condición retorna **true** de lo contrario **false**.
- **obtenerRestriccion():** genera un String con los valores de la variable elegida para la restricción, el tipo de restricción agregada y el valor de la nueva restricción en el *"lado derecho"*. El resultado posee el siguiente formato: **variable** (**desigualdad**) **valor**.
- **valorVariables(espacio : String):** genera un String con el valor de las variables del problema contenido dentro del DtoSimplex. El string contiene el valor del resultado de la función objetivo y de las variables que la conforman, por cada salto de línea posee un espacio indicado por parámetro.
- **toString(espacio: String):** Genera un String de los datos del nodo, indica si es acotado, factible o muestra todo el contenido con un espacio de tabulación, dicho contenido incluye el valor de las variables ejecutando el método **valorVariables(espacio : String)**.
- **toStringRepeat(cantidad : int):** Genera el String del arbol que representa el nodo actual. Posee la estructura similar a la de un file system, con los nodos como elementos. Estrucutra: Problema 1 información del nodo Problema 1.1 Información del nodo, la información del nodo se genera por medio de la función **toString(espacio: String),** donde se crea la cantidad de espacio de tabulación según el parámetro llamado **cantidad**.
- **repetirCaracter(cantidad : int, caracter : char) :** repite la cantidad de veces dentro de un String el caracter que se indique por parámetro.
- **obtenerTodasRestricciones():** recorre recursivamente el arbol donde se encuentra el nodo donde se ejecute este método, tomando las restricciones agregadas y almacenandolas en un solo String. El nodo raíz no posee restricción y se le agrega el dato "No es necesario".

6.7. Recursos

- No utiliza ningún otro recurso.

6.8. Procesamiento

- **toStringRepeat:**
    1. Genera un espacio para tabular entre los niveles del árbol. Dicho espacio se obtiene repitiendo un carácter las veces indicado por el parámetro utilizando la función **repetirCaracter**.
    2. Agrega un encabezado con la cadena “Problema ” y el índice al String resultado, siempre iniciando con el espacio de tabulación.
    3. Obtiene el String de los datos del nodo actual por medio del método **toString**.
    4. Compara que los hijos del nodo actual no sean nulos y ejecuta el método **toStringRepeat** a cada nodo con una tabulación de 5 espacios más, agregandolo al resultado.
    5. Retorna el String resultado.

6.9. Datos

- **problema:DtoSimplex:** contiene todas las característicasdel problema de programación lineal con las restricciones agregadas recorriendo el árbol que genera el algoritmo de Branch And Bound.
- **padre:NodoBranchAndBound:** referencia al nodo padre del árbol donde se encuentra el nodo. Si este es nulo significa que es el nodo raíz del árbol.
- **hijoIzquierdo:NodoBranchAndBound:** referencia al hijo izquierdo del nodo actual. Si es nulo significa que es un nodo hoja y no posee hijo derecho.
- **hijoDerecho:NodoBranchAndBound:** referencia al hijo derecho del nodo actual. Si es nulo significa que es un nodo hoja y no posee hijo izquierdo.
- **acotado:boolean:** valor binario que indica si el problema o nodo actual fue acotado porque existe otra mejor solución. Si es **true** significa que está acotado, **false** de lo contrario.
- **factible:boolean:** indica si el nodo actual con la restricción agregada posee una solución óptima factible. Si es **true** significa que el problema es factible, **false** de lo contrario.
- **optimo : boolean:** indica que si el nodo actual posee la solución óptima generada por el algoritmo de Branch And Bound. Si es **true** significa que es un óptimo, **false** de lo contrario.
- **indiceVariableRestriccion : int:** contiene el índice de la variable elegida para crear una nueva restricción, dicho valor indexa el arreglo de nombre de columnas almacenada entre el DtoSimplex.

- **restriccion : int :** valor entero que indica el tipo de restricción agregada por el algoritmo de branch and bound. Dicho entero que representa a las restricciones de encuentran en el archivo sym.java.
- **valorRestriccion : AbstractFraccion:** representa al lado derecho de la restricción agregada. El valor es entero pero se encapsula en una instancia de AbstractFraccion.
- **indiceProblema : String:** valor String que indexa los problemas de programación lineal dentro del árbol representado en String.

7. modelo.SolucionadorGomory.java

7.1. Tipo

Archivo .java

7.2. Propósito

El propósito de la clase es implementar la funcionalidad para utilizar el algoritmo de Cortes de Gomory para solución entera de problemas lineales. Cumple con REQ-9 y REQ-10.

7.3. Función

La función de esta clase es implementar la lógica necesaria para resolver un problema de programación lineal mediante el algoritmo de Cortes de Gomory. Extiende de la clase SimplexControlador y sobreescribe los métodos solucionar y siguientePaso para comportarse de acuerdo al algoritmo de gomory.

7.4. Dependencias

- **DtoSimplex:** El solucionador recibe por entrada a sus métodos un dto que representa un paso de la solución del problema. Esto significa que el solucionador no poseerá atributos propios, solamente utilizará la información contenida dentro del dto para ejecutar la funcionalidad requerida.
- **SolucionadorSimplex:** Superclase de la que extiende.

7.5. Restricciones

- El dto recibido por parámetro en cada uno de los métodos debe estar bien formado:
  - Todos los atributos del dto deben estar inicializados.
  - Todos los valores de la matriz numérica de fracciones deben estar inicializados.

    - El dto debe indicar mediante sus atributos correctamente la información sobre los datos contenidos en ese dto. Esto significa que las variables de control dentro del dto deben reflejar verazmente el formato de los datos allí contenidos.
    - La lista con los nombres de las filas y columnas representan respectivamente las restricciones del problema y las variables asociadas a cada fila y columna de la matriz numérica. Por lo tanto, estas listas deberán tener las dimensiones apropiadas para describir la matriz numérica en su totalidad.

7.6. Interfaces

La clase define los siguientes métodos:

- **solucionar(dto:DtoSimplex):** Itera las veces posible sobre el problema de programación lineal ingresado hasta llegar a una solución o encontrar que el problema no es factible. El arreglo de elementos tipo DtoSimplex que retorna contiene todos los pasos tras ejecutar el algoritmo de Gomory.
- **siguientePaso(dto:DtoSimplex):** obtiene la siguiente iteración o paso del problema.
- **esSolucionEntera() :** verifica que el valor de todas las variables que forman parte de la función objetivo posean valores enteros. Si se cumple esta condición retorna **true** de lo contrario **false**.
- **obtenerIndiceRestriccionCorte(dto : DtoSimplex):** obtiene el índice de la fila seleccionada para realizar el siguiente corte.
- **realizarCorte(dto : DtoSimplex, indiceRestriccion : int):** genera un nuevo corte basado en el índice de la fila enviado por parámetro, toma la fila de la matriz y obtiene los valores decimales de los coeficientes para crear la nueva restricción o corte del problema.
- **obtenerCorte(coeficiente : AbstractFraccion):** obtiene de un coeficiente el valor de la parte decimal para generar un nuevo corte de Gomory.
- **agregarNombresColumnas(nombres : String[ ]):** agrega dos nuevos nombres de columna una artificial y otra de holgura, debido a que la nueva restricción agregada mediante el algoritmo de cortes de gomory siempre es mayor igual.
- **agregarNombresFilas(matriz : AbstractFraccion[][], nombreFilas : String[], nombreColumnas : String[]):** Busca

cuáles son las variables básicas de cada fila y retorna el arreglo de string respectivo para retornar con la nueva restricción agregada.

7.7. Procesamiento

Se describen el flujo de ejecución de los métodos públicos:

- **solucionar(dto:DtoSimplex)**
    1. Se resuelve el problema de manera directa utilizando el mismo método pero de la superclase.
    2. Una vez ejecutado el paso 1, se obtiene una lista de dto con todos los pasos intermedios para llegar a una solución óptima, o bien haber encontrado una situación de error.
    3. Se extrae el último paso de la lista, esto es, el paso final que representa la última tabla del problema.
    4. Si este último problema no es factible o no es acotado (esto se sabe mediante los atributos del dto), se retorna la lista de pasos actual.
    5. Se llama al método esSolucionEntera para saber si el último problema posee una solución entera.
    6. Si el último problema posee una solución óptima entera, se retorna la lista de pasos actual.
    7. Si el último problema posee una solución óptima no entera:
        a. Mientras el último problema no posea una solución entera, llamar al método siguientePaso.
        b. Actualizar el valor del último paso y volver al punto 4.
- **siguientePaso(dto:DtoSimplex):**

    1. Se obtiene el siguiente paso en el algoritmo simplex llamando al método siguientePaso de la superclase.
    2. Si el siguiente paso no es una solución final (atributo del dto), no es necesario agregar ningún corte pues está aún en el medio del algoritmo simplex regular. Retornar el siguiente paso.
    3. Si el siguiente paso es una solución final, se debe asegurar que la solución sea entera. Para ello, se llama al método esSolucionEntera.
    4. Si el siguiente paso posee una solución entera, se ha llegado a un óptimo entero. Retornar siguiente paso.
    5. Si el siguiente paso no posee una solución entera, se debe agregar un nuevo corte de gomory al problema:
        a. Se obtiene el índice de la fila con la parte entera mayor en el RHS mediante el metodo obtenerIndiceRestriccionCorte.

b. Se obtiene el nuevo corte mediante el método realizarCorte.
c. Se agrega una nueva fila a la matriz mediante el método agregarFila. Esta fila posee el valor de 0 en todas sus entradas.
d. Se copian los valores obtenidos en el punto 5.b a la nueva fila agregada.
e. Se agregan dos nuevas columnas a la matriz (nueva variable de holgura y variable artificial de la nueva restricción).
f. Se llama al método completarDto para que termine de formar el nuevo dto con el formato adecuado para la siguiente iteración. Completar el DTO implica actualizar los atributos para que reflejen el estado actual del problema y actualizar los nombres de las filas y columnas.

7.8. Datos

La clase posee los siguientes atributos:

- **listaPasos : ArrayList<DtoSimplex>:** arreglo de DtoSimplex que almacena el estado del problema de programación lineal en cada iteración del algoritmo de cortes de Gomory.

8. modelo.parser.IParser.java

8.1. Tipo

Archivo .java.

8.2. Propósito

Definir el método parse(String) para cumplir el requerimiento REQ-2.

8.3. Función

Esta interfaz la va a tener que implementar toda clase que desee proveer el servicio de parseo de un string de un problema de programación lineal para convertirlo en un objeto de tipo DtoSimplex que pueda ser solucionado por medio de una implementación de AbstractSolucionadorSimplex. Su función es aislar al controlador de la implementación concreta del componente que va a proveer el servicio de convertir un string a un objeto de tipo DtoSimplex, para disminuir el acoplamiento y cumplir con el principio de inversión de dependencias de SOLID.

8.4. Dependencias

8.5. Restricciones

- No se han identificado restricciones.

8.6. Interfaces

- **parse(value:String):** Toma el string pasado por parámetro y verifica que posea el formato correcto, retornando un objeto de tipo DtoSimplex el cual contiene la representación del problema lineal que contenía el string original.

8.7. Procesamiento

Este componente es una interfaz y no posee código ejecutable.

8.8. Datos

Este componente no tiene atributos.

9. modelo.parser.SimplexParser.java

9.1. Tipo

Archivo .java.

9.2. Propósito

Implementa la interfaz IParser para proveer el servicio de parseo de un problema de programación lineal y así complir con el requerimiento REQ-2.

9.3. Función

La función de esta clase es la implementación de un parser LALR que defina las reglas gramaticales que van a definir el lenguaje libre de contexto aceptado por el programa. Esta clase se encarga de recibir un string que represente el problema de programación lineal que se quiera resolver y extraer la información relevante de él, revisando al mismo tiempo que cumpla con las reglas de formato establecidas.

9.4. Dependencias

- **Fracción:** Utiliza esta clase para crear los obejtos de tipo fracción que poblarán la matriz numérica que representa el

problema dentro del DtoSimplex que retorna al finalizar su ejecución.

- **DtoSimplex:** La clase retorna desde su método parse(string) un DtoSimplex con la representación lógica del problema de programación lineal pasado por parámetro.

9.5. Restricciones

- El Parser posee un scanner inicializado antes de empezar la ejecución del método parse.

9.6. Interfaces

Esta clase fue autogenerada por el programa Java CUP, no se explicarán todos los métodos que posee debido a que el equipo de desarrollo no fue el encargado de escribirla y no tiene información clara acerca del funcionamiento interno de esta clase. El único método que es llamado externamente es parse(String):

- **parse(value:String):** Toma el string pasado por parámetro y verifica que posea el formato correcto, retornando un objeto de tipo DtoSimplex el cual contiene la representación del problema lineal que contenía el string original.

9.7. Recursos

- **Java CUP:** Para la realización del parser se utilizó la herramienta Java CUP. Esta herramienta es un .jar que trabaja en conjunto con JFlex para lograr un análisis léxico y sintáctico de una cadena de texto. Al igual que JFlex, la gramática se define en un archivo aparte que luego puede ser compilado utilizando un .jar que generará código Java que puede ser compilado utilizando el compilador de java. El parser posee una referencia al scanner generado por JFlex, e invoca a su método next_token() cada vez que necesita el próximo token del string. La gramática se define en el archivo /Implementacion/Java/modelo/parser/Parser.cup. Este archvo puede ser generado utilizando java-cup-11b.jar mediante el siguiente comando:

java -jar java-cup-11b.jar package "modelo.parser" -parser "SimplexParser" -Parser.cup

- **Archivo Parser.cup:** Es el archivo donde se define la gramática del formato textual de entrada, así como la lógica para armar el

DtoSimplex que será retornado como resultado. En esta sección se explicará su funcionamiento:

- La primera sección entre las etiquetas **parser code{:** ... **:}** será copiado directamente al archivo generado por CUP. En ella se definen atributos que el parser utiliza para su funcionamiento. Serán descritos en la sección 9.2.
- La siguiente sección **init with {:** ... **:}** define el codigo a ser ejecutado en la inicialización el parser. En este caso, se asigna el atributo scanner de la superclase lr_parser al atributo privado local de la subclase SimplexParser.
- La sección **scan with{:** ... **:}** define el código que se ejecutará cada vez que el parser necesite un nuevo token para realizar su funcionalidad. Pide el siguiente token al scanner y retorna este valor.
- La siguiente sección declara los valores de las producciones terminales y no terminales de la gramática.
  - **Terminales:** MAS, MENOS, COEFICIENTE, MAYORIGUAL, MENORIGUAL, IGUAL, NUMRESTRICCION, CAMBIOLINEA, VARIABLE, MAX, MIN.
  - **No terminales:** problemaSimplex, funcionObjetivo, restricciones, restriccion, declaracionVariables, declaracionVariable, declaracionVariablesAux, desigualdad, operador.
- Seguidamente se define la gramática junto con el código que se va a ejecutar una vez que se reduzca cada producción. La gramática que define el lenguaje aceptado es la siguiente:

problemaSimplex ::= funcionObjetivo **CAMBIOLINEA** restricciones
funcionObjetivo ::= **MAX VARIABLE IGUAL** declaracionVariables | **MIN VARIABLE IGUAL** declaracionVariables
restricciones ::= restriccion **CAMBIOLINEA** restricciones | restriccion
restriccion ::= declaracionVariables desigualdad **COEFICIENTE** | declaracionVariables desigualdad **MENOS COEFICIENTE**
declaracionVariables ::= declaracionVariables operador declaracionVariable | declaracionVariable
declaracionVariable ::= **COEFICIENTE VARIABLE** | **VARIABLE** | **MENOS COEFICIENTE VARIABLE** | **MENOS VARIABLE**
desigualdad ::= **MAYORIGUAL** | **MENORIGUAL** | **IGUAL**
operador ::= **MAS** | **MENOS**

9.8. Referencias

Documentación de CUP disponible en el siguiente enlace: http://www2.cs.tum.edu/projekte/cup/manual.html.

9.9. Procesamiento

El procesamiento de CUP se basa en el funcionamiento de un parser estilo LALR. Se espera que el lector de la siguiente sección tenga conocimiento sobre teoría de compiladores, lenguajes libres de contexto y gramáticas formales para definir las reglas de producción de algún lenguaje. En un apartado anterior se definió la gramática formal del lenguaje aceptado por el programa para representar un problema de programación lineal. Este estilo se toma de los apuntes de clase del profesor Jose Helo a petición de él mismo.

Java Cup tiene la funcionalidad de poder ejecutar código una vez que alguna producción ha sido reducida. Se tomó en cuenta esta funcionalidad para ir rellenando las estructuras de datos descritas en el apartado **9.7**, en el archivo Parser.cup. A continuación se describirá el funcionamiento del parser mediante un ejemplo:

```
max z = 3 x1 + x2   +   2 x3
          3 x1          +  4 x3  <= 10
                  x2    -  -4 x3 >= 1
```

1. El punto de entrada de la clase es la función parse(string). Cuando ella es llamada, se inicializa el una instancia de modelo.parser.Scanner para obtener los tokens del string pasado por parámetro.
2. Se inicializa el parser y se llama a su funcionamiento.
    a. La producción **funcionObjetivo** hace match con la primera línea. Los pares llave valor (“x1”,3) (“x2”,1) (“x3”,2) han sido agregados a *hashFracciones*. Se agrega este hash a *listaRestricciones* y se limpia la referencia con un objeto nuevo para la siguiente línea.
    b. La producción **restricción** hace match con la segunda línea. Los pares llave valor (“x1”,3) (“x3”,4) (null,10) han sido agregados a *hashFracciones*. Note como el valor null se utiliza para almacenar el coeficiente del lado derecho. También se agrega a listaDesigualdades el valor sym.MENORIGUAL. Se agrega el hash a

listaRestricciones y se limpia la referencia con un objeto nuevo para la siguiente línea.

c. La producción **restricción** hace match con la segunda línea. Los pares llave valor (“x2”,1) (“x3”,4) (null,1) han sido agregados a *hashFracciones*. El parser cambia el valor del coeficiente de acuerdo a su representacion matricial, por lo que - - 4x3 se evalúa como un par con el valor (“x3”,4) por el doble negativo. Note como el valor null se utiliza para almacenar el coeficiente del lado derecho. También se agrega a listaDesigualdades el valor sym.MAYORRIGUAL. Se agrega el hash a *listaRestricciones* y se limpia la referencia con un objeto nuevo para la siguiente línea.
d. Una vez que las estructuras están bien formadas y el problema ha sido parseado en su totalidad, se arma el DtoSimplex de retorno:
    i. Se crea una matriz con tantas filas como restricciones y con tantas columnas como variables diferentes hayan. Para esto se utiliza *conjuntoVariables*, el cual contiene todas las variables enontradas en el problema.
    ii. Se recorre la matriz para rellenarla con los valores numéricos necesarios
        1. Se extraen los hashMap de *listaRestricciones*.
        2. Se extraen los valores de cada hashMap.
        3. Se agrega el valor de los pares en las posiciones adecuadas dependiendo de la fila y columna que representen. En el ejemplo, se crea entonces la matriz.

$$\begin{matrix} 3 & 1 & 2 & 0 \\ 3 & 0 & 4 & 10 \\ 0 & 1 & 4 & 1 \end{matrix}$$

    iii. Se retorna un nuevo DtoSimplex con la matriz armada, la lista de desigualdades que representa la desigualdad presente en cada restricción, y la lista de variables del problema. Es importante notar que el orden que tenga la lista de variables debe coincidir con el orden de las columnas de la matriz, pues algoritmos posteriores utilizan los índices de la lista de variables para obtener valores numéricos en la matriz y viceversa. En el

ejemplo, el dto retornado tiene los siguientes valores:

- *listaVariables*: [“x1”. “x2”. “x3”]
- *listaDesigualdades:* [sym.MENORIGUAL, sym.MAYORIGUAL].
- *matriz:*
  3 1 2 0
  3 0 4 10
  0 1 4 1
- *esMaximizacion:* **true.**

3. Se retorna el DtoSimplex formado o se lanza una excepción de error de formato en caso de no haber sido posible parsear el string de entrada.

9.10. Datos

- **conjuntoVariables**:**TreeSet<String>** Es un conjunto ordenado provisto por Java para guardar en él el nombre de todas las variables encontradas en el texto.
- **hashFracciones**:**HashMap<String, AbstractFraccion>** un HashMap que tiene como llave un String (el nombre de una variable) y su valor es el coeficiente de dicha variable en una restricción. Su objetivo es mantener el valor del coeficiente de las variables contenidas en una restricción.
- **listaRestricciones**:**ArrayList<HashMap<String,AbstractFraccion>>** Una lista que contiene objetos del tipo anterior. Esta lista posee una referencia a un HashMap por restricción, de esta manera entonces, el elemento en la posición 0 es un mapa que contiene los coeficientes de las variables en la restricción 0. Por ejemplo:

  max z = x1 + x2
  (0) 1 x1 + 2 x2 <= 4
  (1) 3 x1 + 4 x2 <=10

  En este caso, listaRestricciones tiene dos valores:
  - listaRestricciones[0] contiene un HashMap con los pares (“x1”, 1), (“x2”,2)
  - listaRestricciones[1] contiene un HashMap con los pares (“x1”, 3), (“x2”,4)

- **listaDesigualdades**:**ArrayList<Integer>** Contiene en orden la lista de las desigualdades encontradas en el problema. Los valores que puede tomar son definidos en la clase modelo.parser.sym, en los campos sym.MAYORIGUAL, sym.MENORIGUAL y sym.IGUAL.

- **esMaximizacion**:**boolean true** si el problema ingresado es de maximización, **false** minimización.

10. modelo.parser.Scanner.java

10.1. Tipo

Archivo .java.

10.2. Propósito

Proveer un scanner que pueda dividir una cadena de texto en tokens representativos que pueda usar la clase modelo.parser.SimplexParser para su funcionamiento y cumplir el requerimiento REQ-2.

10.3. Función

La principal función de esta clase es subdividir una cadena de texto en tokens utilizando expresiones regulares para proveer a la clase modelo.simplexParser con los tokens necesarios para el correcto funcionamiento de un parser LALR. Esta es una clase autogenerada por la herramienta JFlex, de manera similar a CUP.

10.4. Dependencias

- **Fracción:** Depende de la clase fracción para instanciar objetos de esta clase cuando encuentra un coeficiente de una variable.

10.5. Restricciones

No se han identificado restricciones.

10.6. Interfaces

Esta clase fue autogenerada por el programa JFlex, no se explicarán todos los métodos que posee debido a que el equipo de desarrollo no fue el encargado de escribirla y no tiene información clara acerca del funcionamiento interno de esta clase. El único método que es llamado externamente es next_token():

- **next_token():** Retorna un objeto de tipo modelo.parser.SimplexSymbol que representa el siguiente token que hizo match con alguna de las expresiones regulares definidas en el archivo Scanner.flex.

10.7. Recursos

- **JFlex:** Para la realización del scanner que divida la cadena de texto en los diferentes tipos de tokens reconocibles se utilizó la herramienta JFlex. Esta herramienta permite definir una serie de expresiones regulares y generar código que deba ser ejecutado cuando el scanner encuentre dicha expresión regular en el texto que está analizando. Las expresiones regulares se definieron en el archivo /Implementacion/Java/modelo/parser/Scanner.flex. Dicho archivo puede ser compilado utilizando jflex-1.6.1.jar mediante el siguiente comando:

  java -jar jflex-1.6.1.jar Scanner.flex
- **Scanner.flex:** Archivo fuente utilizado por JFlex para generar esta clase. En él se define una serie de expresiones regulares que el Scanner va a tratar de encontrar en el string de entrada, y una sección de código ejecutable una vez que expresión haya hecho match con alguna de las reglas. La estructura del archivo es la siguiente:
  - Dentro de las llaves **%{** ... **}%** se declara código que será copiado sin cambios en la clase generada. En este caso se definen dos funciones para crear nuevas instancias del tipo modelo.parser.SimplexSymbol.
  - Luego, se declaran expresiones regulares que serán utilizadas para definir los tokens en los que la herramienta separará el string.
  - Dentro de la etiqueta **<YYINITIAL>** se define el comportamiento que va a tener el programa. Cada vez que la función next_token es llamada, el programa intentará hacer match con algunas de las expresiones presentes en la columna izquierda. En caso de lograr hacer match con una de ellas, se ejecutará el código que se encuentra en la columna derecha.

10.8. Referencias

Documentación de JFlex disponible en el siguiente enlace: http://www2.cs.tum.edu/projekte/cup/manual.html.

10.9. Procesamiento

El procesamiento de la clase es desconocido por el equipo de desarrollo debido a la utilización de una herramienta externa para realizar el código fuente compilable de java. A grandes razgos, la clase recibe un string y conforme va encontrando expresiones regulares dentro de ella va

retornando objetos de tipo modelo.parser.SymplexSymbol que la clase modelo.parser.SimplexParser puede consumir.

10.10. Datos

La clase no posee atributos agregados por el equipo de desarrollo.

11. modelo.parser.SimplexSymbol.java

11.1. Tipo

Archivo .java

11.2. Propósito

Proveer una clase para almacenar valores necesarios para el funcionamiento intermedio de modelo.parser.SimplexParser y cumplir con el requerimiento REQ-2.

11.3. Función

Proveer un objeto para almacenar información de cada token retornado por la función next_token de la clase modelo.parser.Scanner. Esta clase extiende de la clase java_cup.runtime.Symbol, la cual define atributos necesarios por todo token que desee trabajar en conjunto con un parser construido utilizando Java Cup.

11.4. Dependencias

- **java_cup.runtime.Symbol:** Superclase.

11.5. Restricciones

Ninguna.

11.6. Interfaces

La clase no posee métodos públicos.

11.7. Procesamiento

La clase no define código ejecutable, solamente funciona como un contenedor.

11.8. Datos

- **linea:int** representa la línea donde fue encontrado este token.
- **coluna:int** representa la columna donde fue encontrado este token.

12. modelo.parser.sym.java

12.1. Tipo

Archivo .java

12.2. Propósito

Trabajar en conjunto con las otras clases de modelo.parser para llevar a cabo el requerimiento REQ-2.

12.3. Función

La función principal es definir identificadores constantes enteros para los posibles nodos terminales de la gramática. Estos identificadores serán utilizados por el programa para reconocer el tipo de token con el que se está lidiando.

12.4. Dependencias

Ninguno.

12.5. Restricciones

Ninguna.

12.6. Interfaces

Esta clase no posee métodos.

12.7. Procesamiento

Esta clase no posee métodos ejecutables.

12.8. Datos

Cada uno de los tokens terminales reconocidos por la gramática:

- **MAS:int** identificador del simbolo “+”.
- **MENOS:int** identificador del simbolo “-”.

- **COEFICIENTE:int** identificador del símbolo coeficiente, representa una fracción numérica con el valor de un coeficiente que acompaña una variable.
- **MAYORIGUAL:int** identificador del simbolo “>=”.
- **MENORIGUAL:int** identificador del simbolo “<=”.
- **IGUAL:int** identificador del simbolo “=”.
- **CAMBIOLINEA:int** identificador del simbolo “\n”.
- **VARIABLE:int** identificador del simbolo que representa una variable del problema.
- **MAX:int** identificador del simbolo “max”.
- **MIN:int** identificador del simbolo “min”.

13. Controlador.AbstractControlador

13.1. Tipo

Archivo .java.

13.2. Propósito

Proveer la lógica de control para la mayoría de los requerimientos: REQ-1, REQ-3, REQ-5, REQ-6, REQ-8, REQ-9, REQ-10, REQ-11, REQ-12, REQ-13. Provee todos los métodos que podrán ser utilizados por la vista para cumplir con las funcionalidades que el cliente requiere.

13.3. Función

Posee métodos para manejar la interacción entre el modelo y la vista. Es el encargado de tomar las peticiones de IVista, realizar la acción en el modelo y actualizar la vista con el nuevo estado del modelo. El múltiple comportamiento que puede tomar el método Simplex de dos fases es donde yace la necesidad de que se haya implementado esta clase como abstracta, varios controladores pueden necesitar ejecuciones o pasos diferentes para cumplir con los distintos casos de uso solicitados por el cliente, donde el controlador se adapte según la acción que se encuentre realizando el usuario. En resumen, define todos los métodos que un controlador concreta debe implementar para cumplir con todas las funcionalidades que el cliente pueda realizar. Toda interacción entre el usuario y el sistema pasa por esta capa de controlador o una subclase de ella.

13.4. Dependencias

- **AbstractSolucionadorSimplex:** El controlador necesita una referencia a una implementación concreta de AbstractSolucionadorSimplex para poder cumplir todas sus funciones.

El controlador solamente posee lógica de comunicación y no cambia el estado del programa de manera directa, sino con peticiones a alguna clase de la capa modelo.

- **DtoSimplex:** La clase almacena cada paso realizado por el algoritmo simplex como un objeto de tipo DtoSimplex, debido a que en este objeto está contenido todo el estado que tenía el problema en un momento determinado, para de esta manera poder mostrarle al usuario el estado completo del problema en cualquier momento.
- **IParser:** La clase depende de un objeto que implemente la interfaz IParser para que provea los servicios de convertir un problema representado en modo textual a un objeto de tipo DtoSimplex utilizable por el resto de componentes del sistema.
- **sym:** Necesario para importar los identificadores enteros globales de los diferentes tokens que posea el problema.
- **IVista:** Un objeto que provea las funcionalidades descritas por IVista es necesario para mostrar al usuario mensaje de información, error o mostrar el estado del problema.

13.5. Restricciones

- El controlador posee una referencia válida a un objeto que implemente la interfaz IVista.

13.6. Interfaces

A continuación se describen los métodos de la clase:

- **solucionar(String : problema, fraccionario : boolean):** Ejecuta la validación del problema ingresado por medio del parser. Completa el problema ejecutando el método **completarProblema** del atributo solucionador y con este mismo soluciona el problema por medio del método **solucionar** obteniendo inmediatamente todas las iteraciones intermedias para obtener un resultado sobre el problema ingresado. El resultado se almacena en el atributo listaPasos. Actualiza la vista con los resultados del problema.
- **siguientePaso(String problema, fraccionario : boolean):** inicializa la ejecución de la solución de un problema por medio iteraciones intermedias. Ejecuta la validación del problema ingresado por medio del parser. Completa el problema ejecutando el método **completarProblema** del atributo solucionador y agrega el problema completado a la lista de pasos. Actualiza la vista con los datos creados.

- **siguientePaso():** itera sobre el elemento de la lista de pasos donde se encuentra el paso actual. Utiliza el método **siguientePaso** del solucionador. Luego valida si la iteración brinda que el problema es factible, acotado o se logró llegar una solución óptima. Actualiza la vista con los datos de la siguiente operación.
- **anteriorPaso():** decrementa el contador de pasoActual. Actualiza la vista con los datos del paso anterior.
- **siguientesOperaciones(coordenada : Point) :** obtiene un String que representan a las siguientes operaciones sobre el elemento de la listaPasos donde encuentre el contador pasoActual. Dicho cadena de texto se obtiene por medio del método **siguientesOperaciones** del solucionador.
- **generarRadios(int Columna):** genera un arreglo de String con el valor representativo de los radios generados en la columna indicada, cada valor es correspondiente a cada fila. Si se encuentre el infinito agrega un String representativo ya sea “oo” o “-oo”.
- **generarRadios() :** genera un arreglo de String con el valor representativo de los radios generados en la columna indicada dentro del punto que se encuentra en el paso actual de las iteraciones, cada valor es correspondientemente por cada fila. Si se encuentre el infinito agrega un String representativo ya sea “oo” o “-oo”.
- **modificarEntradaMatriz(fila : int, columna : int, valor : String) :** Cambia el valor de una entrada en la matriz de coeficientes actual, por un nuevo valor en la posición fila y columna indicado dentro de la iteración del paso actual.
- **generarResumen():** recorre el arreglo de las iteraciones (**listaPasos**), ejecutando el método toString de cada DtoSimplex.
- **agregarColumna(posicion : int) :** modifica la matriz que se encuentra dentro del DtoSimplex de la iteración actual. Agrega una columna en la posición indicada ejecutando el método **agregarColumna** del solucionador.
- **agregarFila(posicion : int) :** modifica la matriz que se encuentra dentro del DtoSimplex de la iteración actual. Agrega una fila en la posición indicada ejecutando el método agregarFila del solucionador.
- **agregarRestriccion(tipo : int):** agrega una nueva restricción indicando el tipo por medio de un número entero que se encuentra dentro del archivo sym.java. Valida que la restricción

agregada en el momento indicado y donde permita que el problema logre culminar su solución de manera estable. Ejecuta el método **agregarRestricción** del solucionador.

13.7. Procesamiento

A continuación se describe el flujo de ejecución de los métodos públicos provistos por esta clase:

- **solucionar(String : problema, fraccionario : boolean):**
    1. Se inicializa la variable DtoSimplex de retorno y se asigna al atributo problemaOriginal el valor textual del problema pasado por parámetro. Este atributo se utiliza para volver a mostrar en el menú principal el problema ingresado en caso de error durante el parseo.
    2. Se llama la función **parse** de la interfaz IParser con el valor textual del problema para ser analizado y creado un DtoSimplex que lo represente
        a. En caso de lograrlo, se continúa con la ejecución.
        b. En caso de error, se lanza una excepción debido a un error en el formato de entrada.
    3. Se llama al método **completarProblema** de AbstractSolucionadorSimplex para que complete el dto del paso 2 y esté listo para ser solucionado por la instancia respectiva de AbstractSolucionadorSimplex.
    4. Se llama al método **solucionar** de la clase AbstractSolucionadorSimplex para obtener la lista con todos los pasos necesarios para resolver el problema. Se asigna esta lista al atributo *listaPasos*.
    5. Se asigna al atributo pasoActual el índice del último paso de *listaPasos*, es decir, el paso actual será el último de la lista retornada.
    6. Se invoca al método **mostrarMatriz** de la interfaz IVista para que la interfaz gráfica muestre el nuevo estado del problema.
    7. Se verifica si el problema es infactible, no acotado o ha finalizado para llamar a los métodos **mostrarMensajeError** o **mostrarMensajeInformación** con un mensaje informativo para el usuario.
- **siguientePaso(String problema, fraccionario : boolean):**
    1. Se inicializa la variable DtoSimplex de retorno y se asigna al atributo problemaOriginal el valor textual del problema pasado por parámetro. Este atributo se utiliza

para volver a mostrar en el menú principal el problema ingresado en caso de error durante el parseo.

2. Se llama la función **parse** de la interfaz IParser con el valor textual del problema para ser analizado y creado un DtoSimplex que lo represente
    a. En caso de lograrlo, se continúa con la ejecución.
    b. En caso de error, se lanza una excepción debido a un error en el formato de entrada.
3. Se llama al método **completarProblema** de AbstractSolucionadorSimplex para que complete el dto del paso 2 y esté listo para ser solucionado por la instancia respectiva de AbstractSolucionadorSimplex en el siguiente paso.
4. Se asigna al atributo *pasoActual* el valor 0 pues se está al inicio de la lista de pasos, el algoritmo aún no ha iniciado.
5. Se inicializa la lista de pasos y se agrega el problema completado.
6. Se muestra el problema completado en pantalla llamando al metodo **mostrarMatriz** de IVista.

- **siguientePaso():**
    1. Se obtiene el paso actual de la lista de pasos utilizando los atributos *listaPasos* y *pasoActual*.
    2. Se obtiene el siguiente paso llamando al método **siguientePaso** de la clase AbstractSolucionadorSimplex con el dto obtenido en el punto 1.
    3. Se agrega el nuevo paso a *listaPasos*.
    4. Se incrementa el contador de pasoActual.
    5. Se muestra el problema completado en pantalla llamando al metodo **mostrarMatriz** de IVista.
    6. Se verifica si el problema es infactible, no acotado o ha finalizado para llamar a los métodos **mostrarMensajeError** o **mostrarMensajeInformación** con un mensaje informativo para el usuario.
- **anteriorPaso():**
    1. Si no se encuentra en el primer paso
        a. Se decrementa el atributo *pasoActual.*
        b. Se muestra el paso anterior en pantalla llamando al metodo **mostrarMatriz** de IVista con el dto en la posicion *pasoActual* de la *listaPasos*.
    2. Si se encuentra en el primer paso, se muestra el menú principal de nuevo llamando a la función **menu** de IVista.

- **siguientesOperaciones(coordenada : Point):**
    1. Llama al método **getOperaciones** del DtoSimplex presente en la *listaPasos* en la posicion *pasoActual*. Este método retorna las próximas operaciones a modo textual en la posición de pivote actual, no en la requerida por el usuario.
    2. Si el Dto se encuentra en la primera fase (**esBloqueoDosFases** del objeto DtoSimplex), se retornan las operaciones de la posición original. Esto se debe a que durante la primera fase del algoritmo simplex, no es posible cambiar la coordenada de pivote.
    3. Si el Dto no se encuentra en la primera fase
        a. Se actualiza la posición actual de pivoteo mediante el método **setCoordenadaPivote** con el valor ingresado por parámetro.
        b. Se vuelve a llamar al método **getOperaciones,** esta vez con la posición pedida por parámetro ya actualizada dentro del Dto.
        c. Se retorna el resultado devuelto en el paso 3.b.
- **generarRadios(columna : int):**
    1. Se llama a la función **calcularRadio** de la clase AbstractSolucionadorSimplex con el índice de la columna de la cual se requiere calcular los radios, pasado por parámetro. Esta función devuelve un arreglo de tipo String con la representación textual de los radios.
    2. Una vez que se obtienen los textos de los radios, es necesario, en caso de ser el valor infinito (representado lógicamente como el valor Integer.MAX_VALUE), cambiar este valor numérico por el símbolo “oo”.
        a. Se recorre el resultado obtenido en el punto 1 con un for. Si el resultado equivale al valor textual de Integer.MAX_VALUE, se cambia este valor por “oo”.
    3. Se retorna la lista de strings obtenida en el punto 1 y modificada en el punto 2.a.
- **generarRadios():**
    1. Se llama a la función **calcularRadio** de la clase AbstractSolucionadorSimplex con el índice de la columna seleccionada en el paso actual. Esta función devuelve un arreglo de tipo String con la representación textual de los radios.

2. Una vez que se obtienen los textos de los radios, es necesario, en caso de ser el valor infinito (representado lógicamente como el valor Integer.MAX_VALUE), cambiar este valor numérico por el símbolo “oo”.
    a. Se recorre el resultado obtenido en el punto 1 con un for. Si el resultado equivale al valor textual de Integer.MAX_VALUE, se cambia este valor por “oo”.
3. Se retorna la lista de strings obtenida en el punto 1 y modificada en el punto 2.a.

- **modificarEntradaMatriz(fila:int, columna:int, valor:String) :**
    1. Se declaran los valores del nuevo numerador y el nuevo denominador. Se inicializan en 0 y 1 respectivamente.
    2. Si el valor enviado por parámetro tiene formato fraccional (esto es, contiene el símbolo “/”)
        a. Se separa el string en el símbolo “/”, se toma la parte a la izquierda del “/” y se asigna como numerador, la parte derecha se asigna como denominador.
    3. Si el valor enviado por parámetro tiene formato decimal (esto es, contiene el punto decimal “.”)
        a. Se llama la función **setEntradaMatriz** del dto actual con los valores fila, columna y el valor del double que representa el valor enviado por parámetro.
        b. Termina la ejecución del método.
    4. Si el valor enviado por parámetro es un número entero o se asigna este valor al numerador. El denominador mantiene su valor de 1.
    5. Se llama la función **setEntradaMatriz** del dto actual con los valores fila, columna, numerador y denominador para que actualice el valor en su matriz interna.
- **generarResumen():**
    1. Inicializa el string que retornará como resultado.
    2. Recorre *listaPasos* desde el inicio hasta el final, agregando al resultado la invocación al método **toString** de cada DtoSimplex contenido en *listaPasos.*
    3. Retorna el resultado.
- **agregarColumna(posicion : int) :**
    1. Se llama al método **agregarColumna** de la clase AbstractSolucionadorSimplex indicándole por parámetro el índice donde debe agregar la nueva columna.

2. Se actualiza el valor del paso actual con el resultado obtenido del paso 1.
3. Se muestra la nueva matriz invocando al método **mostrarMatriz** de IVista.

- **agregarFila(posicion : int) :**
  1. Se llama al método **agregarColumna** de la clase AbstractSolucionadorSimplex indicándole por parámetro el índice donde debe agregar la nueva columna.
  2. Se actualiza el valor del paso actual con el resultado obtenido del paso 1.
  3. Se muestra la nueva matriz invocando al método **mostrarMatriz** de IVista.
- **agregarRestriccion(tipo : int):**
  1. Se valida que el paso actual sea el primero, caso contrariose invoca el método **mostrarMensajeError** para indicarle al usuario que debe estar en el primer paso para agregar una restricción.
  2. Se valida que si se intenta agregar una restricción tipo “>=” ó “=”, el problema actual sea de dos fases. Esto es necesario pues no se pueden agregar variables artificiales en cualquier momento del simplex, solamente cuando el problema es de dos fases y se encuentra en la primera tabla.
  3. Se llama al método **agregarRestriccion** de la clase AbstractSolucionadorSimplex indicándole por parámetro el id de la restricción que se desea agregar. Este id puede ser sym.MAYORIGUAL, sym.MENORIGUAL ó sym.IGUAL.
  4. Se actualiza el valor del paso actual con el resultado obtenido del paso 3.
  5. Se muestra la nueva matriz invocando al método **mostrarMatriz** de IVista.

13.8. Datos

- **solucionador:AbstractSolucionadorSimplex:** provee la solución de los problemas de programación lineal utilizando el método Simplex. Este atributo puede tomar distintas clases concretas gracias al polimorfismo.
- **parser:IParser:** ejecuta la validación de la cadena de texto que representa al problema de programación lineal.

- **vista:IVista:** ejecuta las acciones que se mostrarán en la pantalla del usuario. Esta interfaz es aplicada en los formularios creados como parte visual del proyecto.
- **pasoActual:int:** valor numérico que indica la iteración actual en la que se encuentra el controlador y se está mostrando al usuario.
- **listaPasos : ArrayList<DtoSimplex>:** arreglo dinámico que contiene las iteraciones que se han generado sobre el problema de programación. El atributo **pasoActual** indica por cuál de sus elementos se encuentra y no puede ser mayor que la cantidad de elementos de este arreglo.
- **problemaOriginal : String:** cadena de texto que representa al problema de programación lineal ingresado por el usuario al inicio de la ejecución.

14. controlador.SimplexControlador

14.1. Tipo

Archivo .java

Proveer la lógica de control para la mayoría de los requerimientos: REQ-1, REQ-3, REQ-5, REQ-6, REQ-8, REQ-9, REQ-10, REQ-11, REQ-12, REQ-13. Provee todos los métodos que podrán ser utilizados por la vista para cumplir con las funcionalidades que el cliente requiere.

14.2. Función

La función de esta clase es extender de su clase padre para heredar la implementación de sus métodos y poder instanciar objetos de este tipo, debido a que AbstractControlador es una clase abstracta y no puede ser instanciada. No hace override de ningún método y solamente utiliza los métodos definidos en su superclase.

14.3. Dependencias

- **AbstractControlador:** Superclase. Hereda todos sus métodos e implementaciones.
- **AbstractSolucionadorSimplex:** El controlador necesita una referencia a una implementación concreta de AbstractSolucionadorSimplex para poder cumplir todas sus funciones. El controlador solamente posee lógica de comunicación y no cambia el estado del programa de manera directa, sino con peticiones a alguna clase de la capa modelo.
- **DtoSimplex:** La clase almacena cada paso realizado por el algoritmo simplex como un objeto de tipo DtoSimplex, debido a que en este

objeto está contenido todo el estado que tenía el problema en un momento determinado, para de esta manera poder mostrarle al usuario el estado completo del problema en cualquier momento.

- **IParser:** La clase depende de un objeto que implemente la interfaz IParser para que provea los servicios de convertir un problema representado en modo textual a un objeto de tipo DtoSimplex utilizable por el resto de componentes del sistema.
- **sym:** Necesario para importar los identificadores enteros globales de los diferentes tokens que posea el problema.
- **IVista:** Un objeto que provea las funcionalidades descritas por IVista es necesario para mostrar al usuario mensaje de información, error o mostrar el estado del problema.

14.4. Restricciones

- El controlador posee una referencia válida a un objeto que implemente la interfaz IVista.

14.5. Interfaces

Misma que la de su superclase. Puede observarla en el apartado 13.5.

14.6. Procesamiento

Misma que la de su superclase. Puede observarlo en el apartado 13.6.

14.7. Datos

Mismos que los de su superclase. Puede observarlos en el apartado 13.7.

15. controlador.MatrizControlador

15.1. Tipo

Archivo .java

15.2. Propósito

Proveer la sobreescritura de los métodos definidos en AbstractControlador (superclase) para que esta clase pueda cumplir los requerimientos REQ-7 y REQ-8.

15.3. Función

Esta clase se encarga de controlar todas aquellas funcionalidades que impliquen trabajar con una matriz numérica ingresada por el usuario. Provee a la vista con el acceso a las funciones que trabajan sobre una matriz numérica, como lo son por ejemplo pivotar sobre una entrada en específico o modificar la entrada de una matriz y no un problema.

Se comporta prácticamente igual a la clase AbstractControlador, con unas pequeñas diferencias en la implementación de ciertos métodos que fueron modificados porque el DtoSimplex tiene una representación un poco diferente en el caso de que se represente una matriz numérica en lugar de un problema de porgrmación lineal. Por ejemplo, los valores *listaNombreColumnas* y *listaNombreFilas* no contienen nombres de variables, sino que todos los campos de estas listas contienen el valor “-”, pues no existen variables en una matriz.

15.4. Dependencias

- **AbstractControlador:** Superclase. Hereda todos sus métodos e implementaciones.
- **AbstractSolucionadorSimplex:** El controlador necesita una referencia a una implementación concreta de AbstractSolucionadorSimplex para poder cumplir todas sus funciones. El controlador solamente posee lógica de comunicación y no cambia el estado del programa de manera directa, sino con peticiones a alguna clase de la capa modelo.
- **DtoSimplex:** La clase almacena cada paso realizado por el algoritmo simplex como un objeto de tipo DtoSimplex, debido a que en este objeto está contenido todo el estado que tenía el problema en un momento determinado, para de esta manera poder mostrarle al usuario el estado completo del problema en cualquier momento.
- **IParser:** La clase depende de un objeto que implemente la interfaz IParser para que provea los servicios de convertir un problema representado en modo textual a un objeto de tipo DtoSimplex utilizable por el resto de componentes del sistema.
- **sym:** Necesario para importar los identificadores enteros globales de los diferentes tokens que posea el problema.
- **IVista:** Un objeto que provea las funcionalidades descritas por IVista es necesario para mostrar al usuario mensaje de información, error o mostrar el estado del problema.

15.5. Restricciones

- El controlador posee una referencia válida a un objeto que implemente la interfaz IVista.

15.6. Interfaces

Misma que la de su superclase. Puede observarla en el apartado 13.5.

15.7. Procesamiento

El procesamiento es el mismo que el de su superclase, con unas pequeñas excepciones. Puede observar el procesamiento en el apartado 13.6 y en el siguiente apartado se mencionarán solamente las diferencias de ciertos métodos con esos de su superclase:

- **solucionar(problema:String, fraccional:boolean)**
  - Se elimina la implementación de este método y retorna solamente nulo, pues este método no es aplicable si el texto ingresado representa una matriz y no un problema de programación lineal.
- **siguientePaso(problema:String, fraccional:boolean)**
  - No se llama a la función **completarProblema** debido a que no aplica esta llamada si no es un problema de programación lineal.
  - Se llama a la función de DtoSimplex **setMensaje** para que el mensaje informativo mostrado en pantalla indique la manera de cumplir conel requerimiento REQ-8.
- **siguientePaso()**
  - No se valida si el problema se encuentra acotado, es factible u óptimo pues no aplica para el caso matricial.
- **modificarEntradaMatriz(fila:int, columna:int,valor:String)**
  - No llama al método setEntradaMatriz, sino que directamente modifica el valor de la matriz contenida dentro del dto del paso actual.

15.8. Datos

Mismos que los de su superclase. Puede observarlos en el apartado 13.7.

16. controlador.GomoryControlador

16.1. Tipo

Archivo .java

Proveer la lógica de control para los requerimientos: REQ-9, REQ-10. Provee todos los métodos que podrán ser utilizados por la vista para cumplir con las funcionalidades que el cliente requiere.

16.2. Función

La función de esta clase es extender de su clase padre para heredar la implementación de sus métodos y poder instanciar objetos de este tipo, debido a que AbstractControlador es una clase abstracta y no puede ser instanciada. No hace override de ningún método y solamente utiliza los métodos definidos en su superclase.

16.3. Dependencias

- **AbstractControlador:** Superclase. Hereda todos sus métodos e implementaciones.
- **AbstractSolucionadorSimplex:** El controlador necesita una referencia a una implementación concreta de AbstractSolucionadorSimplex para poder cumplir todas sus funciones. El controlador solamente posee lógica de comunicación y no cambia el estado del programa de manera directa, sino con peticiones a alguna clase de la capa modelo.
- **DtoSimplex:** La clase almacena cada paso realizado por el algoritmo simplex como un objeto de tipo DtoSimplex, debido a que en este objeto está contenido todo el estado que tenía el problema en un momento determinado, para de esta manera poder mostrarle al usuario el estado completo del problema en cualquier momento.
- **IParser:** La clase depende de un objeto que implemente la interfaz IParser para que provea los servicios de convertir un problema representado en modo textual a un objeto de tipo DtoSimplex utilizable por el resto de componentes del sistema.
- **sym:** Necesario para importar los identificadores enteros globales de los diferentes tokens que posea el problema.
- **IVista:** Un objeto que provea las funcionalidades descritas por IVista es necesario para mostrar al usuario mensaje de información, error o mostrar el estado del problema.

16.4. Restricciones

- El controlador posee una referencia válida a un objeto que implemente la interfaz IVista.

16.5. Interfaces

Misma que la de su superclase. Puede observarla en el apartado 13.5.

16.6. Procesamiento

Misma que la de su superclase. Puede observarlo en el apartado 13.6.

16.7. Datos

Mismos que los de su superclase. Puede observarlos en el apartado 13.7.

17. controlador.BranchAndBoundControlador

17.1. Tipo

Archivo .java.

17.2. Propósito

La necesidad de interacción entre el usuario por medio de la capa gráfica y los datos procesado en las funcionalidades descritas en los requerimientos REQ-11 y REQ-12 orientados hacia la resolución de problemas de programación lineal utilizando el algoritmo de Branch and Bound, se define el propósito de crear esta clase concreta.

17.3. Función

Este tipo de controlador, se encarga de controlar el flujo de los datos entre la vista destinada a mostrar y recibir datos del algoritmo BranchAndBound ingresados por el usuario y el modelo que define el algoritmo de BranchAndBound denominado SolucionadorBranchAndBound. Además este componente verifica el estado del problema en cualquiera de las iteraciones por ello se encarga de indicarle a la vista que muestre un mensaje de información o error, también luego de iterar una vez el problema actualiza el árbol que se debe mostrar en la vista.

17.4. Restricciones

- **IVista**: el controlador debe poseer una instancia un objeto que implemente la interfaz IVista, con los métodos definidos para poder mostrarle al usuario la información proveída por el modelo al usuario. Este objeto nunca puede estar nulo.
- **Texto del problema:** se debe indicar por medio de una cadena de texto distinta de nulo y no vacía, el problema de programación lineal que se desea resolver. Dicho dato es procesado por el parser para obtener el DtoSimplex con el cual se resolverá el problema. Si la cadena de texto no es válida o vacía se lanzará una excepción generada por el parser.

17.5. Dependencias

- **modelo.parser.SimplexParser**: es necesario poder instanciar un objeto tipo SimplexParser debido a que uno de las funciones de este controlador es verificar que la cadena de texto que representa al problema de programación lineal tenga la estructura válida.
- **modelo.SolucionadorBranchAndBound**: dentro del constructor de esta clase, se define un objeto tipo SolucionadorBranchAndBound con el objetivo de que sea invocado para procesar los datos dentro de un DtoSimplex y poder luego brindarle el resultado al usuario.
- **dto.DtoSimplex**: este tipo de dato es utilizado para encapsular la información de problema del programación lineal brindado por el usuario. Dentro de este controlador no se define ninguna instancia, únicamente la definición por medio del Parser y el SolucionadorBranchAndBound.

17.6. Interfaces

Se describen los métodos definidos dentro de este componente:

- **solucionar(String : problema, fraccionario : boolean) :** Soluciona y obtiene inmediatamente todas las iteraciones del árbol del algoritmo de Branch and Bound. Indica el árbol generado en la solución y la información en el mensaje, cada uno dentro del DtoSimplex de la lista. Llena toda la lista de pasos al ejecutar. Actualiza la vista con la solución del problema.
- **siguientePaso(String problema, fraccionario : boolean) :** este método soluciona y obtiene todas las iteraciones del árbol. Luego de ello apunta al contador pasoActual a la primer iteración para comenzar a navegar entre todos los pasos creados. Actualiza la vista.

- **siguientePaso() :** aumenta el contador del paso actual y verifica que la siguiente iteración de la lista, tenga la solución del problema o el problema no sea factible. Actualiza la vista con los datos de la siguiente iteración.
- **anteriorPaso():** decrementa el contador de pasoActual. Actualiza la vista con los datos del paso anterior.

17.7. Recursos

Este objeto no utiliza recursos externos.

17.8. Procesamiento

- **siguientePaso(String problema, fraccionario : boolean):**
    1. Se valida la estructura de la cadena de texto ingresada por parámetro, donde se ejecuta el método **parse** del atributo **parser** con la cadena de texto como parámetro.
        a. En caso que la estructura de la cadena de texto ingresada sea incorrecto, se lanza un excepción donde por medio del catch se le informa al usuario por medio de un mensaje de error proveído por el método del atributo **vista** llamado **menu**.
    2. Si la cadena texto es correcta, se crea un nuevo DtoSimplex y se le asigna el formato fraccional indicado por parámetro.
    3. Se ejecuta el método **solucionar** del atributo solucionador, donde se obtiene una lista de DtoSimplex los cuales representan a las iteraciones y se almacenan en el atributo **listaPasos**.
    4. Se establece el *pasoActual* como el primer índice del arreglo dado como 0.
    5. Se ejecuta el método **mostrarMatriz** del atributo vista, con el cual la vista se actualizará con los datos contenidos dentro del DtoSimplex.
- **siguientePaso():**
    1. Se valida que el contador denominado *pasoActual* +1, no sobrepase la cantidad de elementos de la lista de pasos.
        a. Si cumple la validación anterior se aumenta el contador *pasoActual*.
    2. Se valida si el DtoSimplex en la posición *pasoActual* dentro del arreglo *listaPasos*, si es no factible.
        a. Si no es factible.

i. Se muestra un mensaje de error por medio del método **mostrarMensajeError** donde se indica que el problema no posee una solución factible.

b. De lo contrario.

i. Se valida si el DtoSimplex en la posición *pasoActual* dentro del arreglo *listaPasos*, si indica que el problema fue finalizado.

1. Se muestra un mensaje en la pantalla por medio del método **mostrarMensajeInformacion** del atributo vista, donde se indica que el problema fue finalizado.
2. Se obtiene el DtoSimplex en la posición *pasoActual* dentro del arreglo *listaPasos.*
3. Se adapta la solución dada dentro del DtoSimplex solución, donde se le agrega el problema original.
4. Se muestra la información contenida dentro del DtoSimplex solución en la vista por medio del método **mostrarMatriz**.

- **anteriorPaso():**
  1. Se valida que el contador denominado *PasoActual,* sea mayor que 0.
     a. Si cumple, se reduce una unidad el contador *PasoActual*.
  2. Se ejecuta el método **mostrarMatriz** del atributo vista, con el cual la vista se actualizará con los datos contenidos dentro del DtoSimplex ubicado en la posición *PasoActual* de la *listaPasos*.
- **solucionar(String : problema, fraccionario : boolean) :**
  1. Se valida la estructura de la cadena de texto ingresada por parámetro, donde se ejecuta el método **parse** del atributo **parser** con la cadena de texto como parámetro.
     a. En caso que la estructura de la cadena de texto ingresada sea incorrecto, se lanza un excepción donde por medio del catch se le informa al usuario por medio de un mensaje de error proveído por el método del atributo **vista** llamado **menu**.
  2. Si la cadena texto es correcta, se crea un nuevo DtoSimplex y se le asigna el formato fraccional indicado por parámetro.
  3. Se establece como *PasoActual* la cantidad de elementos de la *listaPasos* menos una unidad.

4. Se ejecuta el método **mostrarMatriz** del atributo vista, con el cual la vista se actualizará con los datos contenidos dentro del DtoSimplex ubicado en la posición *PasoActual* de la *listaPasos*.
5. Se valida si el DtoSimplex en la posición *pasoActual* dentro del arreglo *listaPasos*, si es no factible.
    a. Si no es factible.
        i. Se muestra un mensaje de error por medio del método **mostrarMensajeError** donde se indica que el problema no posee una solución factible.
    b. De lo contrario.
        i. Se valida si el DtoSimplex en la posición *pasoActual* dentro del arreglo *listaPasos*, si indica que el problema fue finalizado.
            1. Se muestra un mensaje en la pantalla por medio del método **mostrarMensajeInformacion** del atributo vista, donde se indica que el problema fue finalizado.
            2. Se obtiene el DtoSimplex en la posición *pasoActual* dentro del arreglo *listaPasos.*
            3. Se adapta la solución dada dentro del DtoSimplex solución, donde se le agrega el problema original.
            4. Se muestra la información contenida dentro del DtoSimplex solución en la vista por medio del método **mostrarMatriz**.
6. Retorna el arreglo de *listaPasos*.

17.9. Datos

No se definen más atributos, solo se usan los heredados de su clase padre.

18. controlador.IVista

18.1. Tipo

Archivo .java. Definición de una interfaz.

18.2. Propósito

Proveer a las clases que serán implementarán la funcionalidad de interfaz gráficas con los métodos que deben poseer para cumplir con todos los requerimientos del presente proyecto, REQ-1, REQ-2, REQ-3, REQ-4,

REQ-5, REQ-6, REQ-7, REQ-8, REQ-9, REQ-10, REQ-11, REQ-12 y REQ-13

18.3. Función

Definir las funciones que toda clase que provea los servicios de interfaz gráfica debe poseer para conectarse con una clase concreta del tipo AbstractControlador. AbstractControlador posee una referencia a un objeto que implemente esta interfaz, así que todo objeto que desee ser vista de la solución debe implementar esta interfaz para que AbtstractControlador pueda invocar a sus métodos para modificar la vista de manera acorde. Esto se hace para disminuir el acoplamiento entre clases y cumplir con el Dependency Inversion Principle de los principios SOLID.

18.4. Dependencias

- **DtoSimplex:** Toda la información que viaja entre las diferentes capas utiliza un objeto del tipo DtoSimplex. La interfaz depende de esta clase para enviar y recibir los datos hasta y desde el modelo.

18.5. Restricciones

- El dto recibido por parámetro en cada uno de los métodos debe estar bien formado:
    - Todos los atributos del dto deben estar inicializados.
    - Todos los valores de la matriz numérica de fracciones deben estar inicializados.
    - El dto debe indicar mediante sus atributos correctamente la información sobre los datos contenidos en ese dto. Esto significa que las variables de control dentro del dto deben reflejar verazmente el formato de los datos allí contenidos.
    - La lista con los nombres de las filas y columnas representan repectivamente las restricciones del problema y las variables asociadas a cada fila y columna de la matriz numérica. Por lo tanto, estas listas deberán tener las dimensiones apropiadas para describir la matriz numérica en su totalidad.

18.6. Interfaces

- **mostrarMensajeError(mensaje : String, encabezado : String) :** muestra un mensaje de error en la vista que lo implemente. El mensaje y el encabezado deben ser indicados.

- **mostrarMatriz(matriz : DtoSimplex) :** toma la matriz de AbstractFraccion dentro del DtoSimplex, y utilizando el toString con un formato indicado lo muestra en la vista que implemente este método.
- **mostrarMensajeError(mensaje : String, encabezado : String):** muestra un mensaje de información en la vista que lo implemente. El mensaje y el encabezado deben ser indicados.
- **menu(mensajeError : String, problema : String) :** método usado para cuando sucede un error de sintaxis en la cadena de texto ingresado. Lanza un mensaje de error y muestra el problema ingresado en la pantalla inicial.

18.7. Procesamiento

Al ser una interfaz de Java no hay implementación de sus métodos y no hay procesamiento.

18.8. Datos

Esta clase no posee atributos.

19. dto.DtoSimplex

19.1. Tipo

Archivo .java.

19.2. Propósito

La representación de un problema de programación lineal en una estructura de datos se materializa por medio de este componente. Todos los requerimientos del alcance del proyecto poseen una relación directa en almacenamiento y uso de esta clase.

19.3. Función

La función dentro del sistema esta estructura es almacenar todos los datos de un problema de programación lineal. Dentro de las características de este componente ayuda a identificar dentro del modelo como solucionar el problema representado y al controlador que acciones tomar. Además nos ayuda a cumplir con el patrón de diseño dto, pues como los datos pasados por parámetros se encuentran dentro de un solo lugar, las dependencias a cambios se encontraran dentro de esta estructura.

19.4. Restricciones

- La matriz de elementos tipo AbstractFraccion no puede poseer nunca el valor de nulo para ejecutar cualquier solucionador que herede dentro de la jerarquía AbstractSolucionadorSimplex.
- El arreglo de nombre de columnas no puede poseer el valor de nulo para ejecutar cualquier solucionador que herede dentro de la jerarquía AbstractSolucionadorSimplex.
- El método **clonarSinCompletarProfundo** devuelve los datos de coordenadaPivote y nombreFilas en nulo.
- Se debe inicializar el valor de variables básicas antes de ejecutar cualquier solucionador que herede dentro de la jerarquía AbstractSolucionadorSimplex.

19.5. Dependencias

- **java.awt.Point:** las operaciones realizadas sobre la matriz que representa el problema de programación lineal, necesita de una estructura que integra la ubicación de las operaciones fila del algoritmo en una iteración dada. Dicha dependencia contiene un valor entero “x” que indica la columna de la matriz y un valor entero “y” que indica la fila.
- **java.io.Serializable:** la inclusión de esta clase es necesaria para leer datos en archivos que contienen información válida para la ejecución de las pruebas.
- **modelo.AbstractFraccion:** los elementos de la matriz de coeficiente son instancias de objetos que heredan de esta clase. Cualquier método que implique a la matriz o indirectamente tiene que usar este tipo de dato.

19.6. Interfaces

- **getOperaciones():** retorna los elementos del arreglo de String operaciones, agregados todos en una sola cadena separados por saltos de líneas.
- **clonarProfundo():** copia todos los atributos del DtoSimplex y retorna una nueva referencia para el objeto y sus atributos.
- **getMatrizString():** retorna una matriz de elementos tipo String, donde se agregan los elementos del atributo **matriz** ejecutando el método toString de cada Fraccion con el formato indicado en el tributo **formatoFraccional**.

- **toString():** retorna un String con el valor de cada Fraccion del atributo matriz dentro de un marco con caracteres '-' conjunto al nombre de las filas y columnas. Ambos atributos de la clase.
- **setEntradaMatriz(fila : int, columna : int, numerador : int, denominador : int):** actualiza el valor de la **matriz** en la fila y columna indicada. Dicho valor que se agrega es tomado de los parámetros numerador y denominador encapsulados en una instancia tipo AbstractFraccion.
- **clonarMatriz():** Crea una nueva matriz de elementos tipo abstractFraccion donde cada dato posee una diferente referencia a su origen. El origen de esta nueva **matriz** es el atributo matriz.
- **clonarSinCompletarProfundo():** copia todos los atributos del DtoSimplex, excepto coordenada y NombresFilas, y retorna una nueva referencia para el objeto y sus atributos.

19.7. Recursos

Esta clase no utiliza algun recurso.

19.8. Procesamiento

No posee algún procesamiento más que el establecer y retornar datos.

19.9. Datos

- **matriz : AbstractFraccion[ ][ ]:** matriz de elementos tipo AbstractFraccion cual representa al problema de programación mientras se encuentra en solución.
- **nombreColumnas : String[]:** nombre de las variables a la cual se le asigna un columna de la **matriz.**
- **nombreFilas : String[ ]**: nombre de las variables a la cual se le asigna un fila de la **matriz**.
- **listaDesigualdades : int[]:** lista de desigualdades representadas por un número entero que contiene cada restricción del problema de programación lineal.
- **maximizacion : boolean:** Valor binario que indica si el problema se encuentra solucionando un problema de maximización (**true**) o de minimización (**false**).
- **factible : boolean:** indica si el problema de programación lineal contenido dentro, es factible (**true**) o no (**false**).
- **acotado : boolean:** indica si el problema de programación contenido dentro, esta acotado (**true**) o no (**false**).

- **dosFases : boolean:** indica si el problema de programación contenido dentro, se tuvo que solucionar por el método de dos fases y se encuenta en la primera etapa (**true**) o no (**false**).
- **finalizado : boolean:** muestra si el problema esta finalizado (**true**) o no(**false**).
- **bloqueoDosFases : boolean:** indica al inicio del método Simplex si el problema se encuentra eliminando las variables artificiales cuando se soluciona por medio de dos fases. Si es **true** se encuentra eliminando los unos de la función w y de lo contrario **false**.
- **formatoFraccional : boolean:** brinda el formato de salida de las fracciones. Si es **true** usa un formato fraccional y **false** usa decimal.
- **esMatriz : boolean:** si la matriz de fracciones fue ingresada por tipo matriz entonces tendrá un valor de true y false si fue ingresada por medio del formato de programación lineal.
- **variablesHolgura : int:** cantidad de variables de holgura del problema de programación lineal.
- **variablesBasicas : int:** cantidad de variables básicas contenida por el problema original.
- **operaciones : String[ ]:** arreglo de elementos tipo String, donde se contiene las siguientes operaciones filas que se realizarán según la coordenada del objeto.
- **cordenadaPivote : Point:** punto con un valor **x** (columna) y **y** (fila), que significa donde se realizará la operación fila.
- **artificialActual : int:** índice de la columna donde se encuentra una variable artificial por eliminar cuando existe el bloque de dos fases.
- **solucion : String:** String con la solución del problema, contiene el valor de cada variable.
- **mensaje : String :** String con información de lo sucedido en el algoritmo.