Documentación de distribución de productos en un supermercado

Tercera entrega

Subgrupo PROP42.1

Biel Llabrés Raurell, biel.llabres

Marcel Alabart Benoit, marcel.alabart

Àngel Cantos Girón, angel.cantos

Adrià Cebrián Ruiz, adria.cebrian

Versión de la entrega: V3.0
(Sin querer se usaron otros usuarios aparte de los que nos ofrecía la FIB)

Índice

Mejoras del proyecto

Algoritmos

Previamente, disponíamos de dos algoritmos para distribuir los productos en la solución, para esta entrega hemos añadido un tercer algoritmo, *Simulated Annealing*, con una cantidad de pasos fija que permite ejecuciones inferiores a un segundo independientemente del tamaño del problema. Esta es un arma de doble filo, puesto que si la entrada es inmensa la calidad podría ser muy baja, aunque es una mejor alternativa que los otros dos algoritmos que no terminarían rápidamente o quizás nunca.

Además, los dos algoritmos que ya existían previamente han sido paralelizados, ambos para operar con 8 threads. Esta es una cantidad estándar hoy en día y nos permite aplicar *Strong Scaling* al algoritmo de fuerza bruta para un *speed-up* cercano a 8. En cuanto al otro algoritmo, *Hill Climbing*, utilizamos los threads adicionales para ejecutar en su totalidad el algoritmo, es decir que cada llamada al algoritmo implica 8 ejecuciones de las cuales seleccionamos la mejor, este es un buen ejemplo de *Weak Scaling* en paralelización. Para más detalles, ver el estudio incluido en la documentación.

También hemos añadido la funcionalidad de detener los algoritmos, por si sus ejecuciones son demasiado largas. Para evitar situaciones en las que la ejecución podría tardar años, cada algoritmo tiene un límite en cuanto al tamaño de la estantería que puede ordenar, S.A. no tiene límite por la ventaja que hemos mencionado previamente.

Interfaz

Por suerte ya habíamos diseñado una interfaz para la primera entrega, hemos realizado algunas mejoras adicionales a esta interfaz. Primeramente, hemos estilizado el diseño mediante un archivo .css para darle un toque synthwave. Además, hemos añadido una barra de carga que aparece al calcular una solución, esto es especialmente útil para el algoritmo de fuerza bruta, ya que el usuario puede optar por cancelar la ejecución si juzga que esta tarda demasiado. También hemos añadido ciertas ventanas de advertencia o de error originadas de una mejor prevención de errores en el código.

Estructura

La estructura del código fuente en nuestra primera entrega no seguía la arquitectura en tres capas. Refactorizar las clases y crear los controladores necesarios para garantizar un funcionamiento no fue tarea fácil. Finalmente, todos los métodos pertenecen a una de las tres capas y solo son accesibles mediante los controladores pertinentes de estas capas.

Historial de cambios

Como funcionalidad extra, se ha hecho un historial de cambios, con el objetivo de que el usuario pueda deshacer cambios no deseados, o realizados por equivocación.
Para ello se tuvo que crear un nuevo atributo de la clase GestorPersistencia, este atributo es de tipo ArrayList<Estado>, de esta forma el historial no tiene una forma fija, y las operaciones son de coste constante, debido a que solamente se debe acceder al último elemento.
Para el funcionamiento del historial, hay 3 métodos principales en GestorPersistencia:

- guardarEstadoEnHistorial: Guarda una copia del estado actual en el ArrayList de Estados
- deshacerCambio: Recoge el último estado guardado en el ArrayList de Estados, lo elimina del ArrayList y lo restaura.
- compararEstados: Este método compara dos estados, si son iguales, devuelve true, false en caso contrario.

Mediante estas 3 funciones, se pueden guardar los cambios cuando se realiza uno, restaurar estados anteriores, y en caso de que se añadan 2 estados iguales, podríamos detectarlo y saltárnoslo.
Y para mantener la estructura en 3 capas, se han añadido métodos para guardar el estado y deshacer un cambio en el DomainEstadoController.

Barra de progreso

Durante el cálculo de distribuciones, según las dimensiones y el algoritmo usado, este puede durar mucho. Para que el usuario pueda ver el progreso, se ha añadido una barra de carga que muestra el progreso del cálculo de la distribución. En los algoritmos de aproximación es una estimación de las iteraciones máximas que podría llegar a tardar dado el caso del problema a resolver.

Asignación de las clases

A continuación se muestra como se han repartido entre los integrantes las diferentes clases que existen en nuestro código. Debido a problemas personales durante el periodo de desarrollo del código, muchas de las clases, sobre todo controladores se han terminado haciendo entre múltiples personas. También incluimos en esta tabla los diferentes test, y el estudio que se ha llevado a cabo para los algoritmos.

Biel	GestorPersistencia, DomainEstadoController, DomainProductoController, DomainSolucionContorller, MatrizAdyacencia, Solución
	IOControllerTest, HistorialTest, MatrizAdyacenciaTest, ProductoControllerTest, SolucionControllerTest, SolucionTest
Àngel	GestorPersistencia, DomainProductoController, Estado, DomainSolucionContorller, IOController, ProductoCell, PropController, RelacionCell, VisualSolucionController, VisualProductoController
Marcel	AlgoritmoUltraRapido, ListaProductos, Producto.
	Estudios.pdf, AlgoritmoUltraRapidoEstudio. AlgoritmoOptimoTest, AlgoritmoRapidoTest, AlgoritmoTest, AlgoritmoUltraRapidoTest, ListaProductosTest, ProductoTest
Adrià	Algoritmo, AlgoritmoRapido, AlgoritmoOptimo, DomainProductoController, DomainSolucionContorller, IOController, IntercambioController, ProductoCell, PropController, VisualSolucionController, VisualProductoController
	AlgoritmoRapidoEstudio

Cambios en la estructura de la segunda entrega

Principalmente, los cambios en la estructura, se deben a la aplicación de la estructura en 3 capas en el proyecto. También, en menor medida, debido a las mejoras del programa explicadas en el primer apartado.

Capa de presentación

En la capa de presentación, se ha modificado sobre todo el "PropController", en el que se han añadido las siguientes funciones:

- onDeshacer: Este método llama al GestorPersistencia para cambiar el estado actual por el estado anterior.
- actualizarAfterSwap: Este método permite actualizar los productos después de que haya un intercambio de productos en la distribución de la solución.
- actualizarSolucionProductosVista: Este método permite actualizar la vista de la distribución sin tener que acceder a los productos, de esta manera se puede mantener una solución que no dependa de los productos cargados.

Algunos atributos de PropController también han sido cambiados con tal de facilitarnos la creación de código, principalmente:

- ❖ solucionView: representa la solución, para poder acceder por la id y no solo depender del nombre, se ha añadido un Integer que indica la id del producto
- ❖ observableProducts: tabla de los productos que se asignan en a ListView
- ❖ observableProductPairs: tabla de pares de productos que se asignan a relacionesView
- ❖ observableSolutionProducts: donde se guarda la distribución que se asignará a solucionView

También, en "VisualSolucionController" se ha añadido un atributo (barra), que representa la barra de progreso que se muestra durante la ejecución de un algoritmo.

Capa de dominio

La capa de dominio es bastante similar a la segunda entrega, en cuanto a estructura se ha añadido un controlador "DomainEstadoController" y se han modificado ciertas conexiones para garantizar la estructura en tres capas.

Las funciones añadidas para esta entrega están explicadas en parte en el primer apartado y algunas con detalle en los estudios contenidos en el archivo de Estudios en la documentación. Sobretodo, los algoritmos han tenido muchos cambios en nuevas funciones y atributos, necesarias para aplicar las mejoras del primer apartado. Las principales adiciones son los siguientes atributos en algoritmos Óptimo y Rápido:

- dist_files: número de filas de la solución
- dist_columnes: número de columnas de la distribución

- stopRequested: indica si el algoritmo ha recibido una petición de parar
- pasosTotales: la suma de pasos dados para encontrar la solución
- maxIter: estimación de pasos máximos que puede dar el algoritmo, usado para barra de carga

Así como las funciones:

- ❖ maxIters(): retorna estimacion de pasos máximos para el algoritmo
- ❖ stopExecution(): set stopRequested a true

Mientras que la nueva clase "AlgoritmoUltraRapido" tiene los atributos explicados y estudiados en el estudio para que funcione adecuadamente el código. Además de atributos básicos como "numPasos" y funciones para detener el código igual que en los algoritmos Óptimo y Rápido.

Además, algunos de los controladores tienen nombres distintos respecto a los de la primera entrega, esto se debe a que había algunos controladores en capa de presentación que se llamaban igual que en capa de dominio, de manera que para evitar errores cambiamos los nombres de estos controladores.

También, el nuevo controlador "DomainEstadoController" contiene los métodos

- deshacer() que permite al usuario deshacer su última acción realizada en la interfaz llamando al "GestorPersistencia" de la capa de persistencia.
- actualizarHistorial() que llama a "GestorPersistencia" para guardar el último historial

Capa de persistencia

En esta capa los únicos cambios en la estructura se deben al historial (Explicados en el apartado de mejoras).
En la clase "GestorPersistencia" se ha añadido un atributo llamado "historial" de tipo ArrayList<Estado>. Este se usa para guardar el estado del programa antes de realizar cualquier cambio, de esta forma si el usuario pulsa "Ctrl+Z", solo se debe recoger el último elemento de la lista, restaurarlo y eliminarlo de la lista. Los métodos añadidos para implementar esto son:

- getEstadoActual(): Que devuelve un objeto de tipo "Estado" con los datos actuales del programa.
- guardarEstadoEnHistorial(): Que guarda el estado actual del programa en el ArrayList "historial".
- compararEstados(Estado e1, Estado e2): Sirve para comparar 2 estados, devuelve true si son iguales, false si no.
  La principal utilidad de esta función es asegurarse de que al deshacer cambios no se restaure un estado igual al que había.

Casos de prueba

Productos

Caso Producto con nombre en blanco:

Pestaña de “Productos” → Añadir producto → Nombre en blanco → Añadir producto
Resultado: Error - El nombre no puede estar vacío
Justificación: De esta forma podemos asegurarnos de que no hay productos sin nombre. En el caso de que hubiera podría haber problemas debido a que hay funciones que usan el nombre del producto para funcionar, y en el caso de las soluciones podría haber casilla vacía que se podrían confundir con el producto.

Caso Producto con precio negativo:

Pestaña de “Productos” → Añadir producto → Nombre ‘test’ → Precio ‘-4’→ Añadir producto
Resultado: Error - El precio no puede ser
Justificación: De esta forma podemos asegurarnos de que no hay productos con precio negativo.

Caso Producto con nombre y precio válidos:

Pestaña de “Productos” → Añadir producto → Nombre ‘test’ → Precio ‘5’
Resultado: Creación del producto
Justificación: Podemos comprobar que la creación de productos funciona. Si esta funcionalidad falla, el programa queda inutilizado.

Caso Producto crear producto con nombre usado:

Pestaña de “Productos” → Añadir producto → Nombre ‘test’ → Precio ‘5’
Añadir otro producto → Nombre ‘test’ → Precio ‘24’
Resultado: Error - El nombre ya existe
Justificación: De esta forma verificamos que no hay posibilidad de que dos productos se llamen igual, hay funciones que usan el nombre del producto, si no fuera único podrá dar problemas.

Caso Producto modificar nombre a uno usado:

Pestaña de “Productos” → Añadir producto → Nombre ‘test’ → Precio ‘5’
Añadir producto → Nombre ‘test2’ → Precio ‘5’
Modificar nombre ‘test2’ a ‘test’
Resultado: Error - El nombre ya existe

Justificación: De esta forma verificamos que no hay posibilidad de que dos productos se llamen igual, hay funciones que usan el nombre del producto, si no fuera único podrá dar problemas.

Caso Producto modificar nombre a nombre vacío:

Pestaña de "Productos" → Añadir producto → Nombre 'test' → Precio '5'
Modificar nombre 'test' a ''
Resultado: Error - El nombre no puede estar vacío
Justificación: La justificación es la misma que en "Caso Producto con nombre en blanco"

Caso Producto modificar precio a negativo:

Pestaña de "Productos" → Añadir producto → Nombre 'test' → Precio '5'
Modificar precio '5 a '-5'
Resultado: Error - El precio no puede ser negativo
Justificación: Un precio negativo podría confundir al usuario.

Caso Producto Exportar lista:

Pestaña de "Productos" → Añadir producto → Nombre 'test' → Precio '5'
Exportar Lista → archivo.list
Resultado: Se crea el archivo.list
Justificación: Se debe comprobar que la lista de productos se puede exportar

Caso Producto Exportar lista:

Pestaña de "Productos" → Añadir producto → Nombre 'prova' → Precio '500'
Importar Lista → archivo.list
Resultado: Se reemplazan los productos que habia en la lista por el Producto('test', 5)
Justificación: Se debe comprobar que la lista de productos se puede importar

Caso Producto Eliminar producto:

Pestaña de "Productos" → Añadir producto → Nombre 'test' → Precio '5'
Producto 'test' → Eliminar producto
Resultado: Se elimina el producto
Justificación: Se debe comprobar que el producto se puede eliminar

Relaciones

Caso modificar relación entre dos productos con valor válido:

Pestaña de "Relaciones" → Primer producto "fuet" segundo producto "patatas" → Modificar el valor de la sinergia por 7.0

Resultado: Modificación de la sinergia entre el producto “fuet” y el producto “patatas” en la matriz de adyacencias por 7.0.
Justificación: Podemos comprobar que la modificación de sinergias funciona. De no ser así el programa quedaría inservible.

Caso modificar relación entre dos productos con valor negativo:

Pestaña de “Relaciones” → Primer producto “fuet” segundo producto “patatas” → Modificar el valor de la sinergia por -7.0
Resultado: Error - Las sinergias entre productos no puede ser negativa.
Justificación: Debido a que el valor por defecto y el mínimo para sinergias entre dos productos, huecos o un producto y un hueco es 0 forzar únicamente la existencia de sinergias positivas facilita la implementación.

Caso modificar relación entre dos productos con valor no válido:

Pestaña de “Relaciones” → Primer producto “fuet” segundo producto “patatas” → Modificar el valor de la sinergia por “buena”.
Resultado: Error - Las sinergias entre productos deben ser números decimales positivos.
Justificación: Sería complejo interpretar la calidad de la relación entre productos con strings.

Caso modificar relación entre dos productos que quizás no existan:

La interfaz solo muestra pares de productos existentes en la lista de productos entrada por el usuario. Por esta razón es imposible intentar modificar una relación entre dos productos que quizás no existan.

Solución

Caso generar solución óptima con dimensiones exageradas

Crear 4 productos, con nombres distintos
Pestaña “Sinergias” → Modificar las sinergias entre productos
Pestaña “Solución” → Generar solución → Columnas 100 → Filas 100 → Algoritmo Optimo → Generar
Resultado: Aviso - La ejecución puede durar mucho tiempo o no acabar nunca.
Justificación: Nos aseguramos de que el usuario recibe el aviso, porque si no deberá esperar sin saber nada.

Caso generar solución óptima con dimensiones aceptables

Crear 4 productos, con nombres distintos
Pestaña “Sinergias” → Modificar las sinergias entre productos

Pestaña “Solución” → Generar solución → Columnas 5 → Filas 2 → Algoritmo Optimo → Generar
Resultado: Se calcula la solución
Justificación: Nos aseguramos de que los algoritmos se ejecuten correctamente

Caso generar solución óptima con dimensiones pequeñas

Crear 4 productos, con nombres distintos
Pestaña “Sinergias” → Modificar las sinergias entre productos
Pestaña “Solución” → Generar solución → Columnas 1 → Filas 2 → Algoritmo Optimo → Generar
Resultado: Error - Ocurrió un error al generar la solución
Justificación: Nos aseguramos de que los algoritmos si no caben los productos salga un error

Estados

Caso guardar estado con path inválido:

Pestaña de “Estado” → Botón “Guardar y salir” o “Exportar estado” → selección del directorio donde se guardará el estado inválido “/hola\quetal/”.
Resultado: Error - El directorio seleccionado no existe o no es válido.
Justificación: El programa no puede guardar archivos en directorios que no existan o que no permitan la escritura.

Caso guardar estado con path a un directorio válido:

Pestaña de “Estado” → Botón “Exportar estado” → selección del directorio donde se guardará el estado válido “/hola/quetal/”.
Resultado: Se escribe un archivo de tipo .estado binario que contiene la lista de productos, matriz de sinergias y distribución de la solución actual.
Justificación: Guardar un estado entero puede ser práctico e incluso esencial.

Caso guardar y salir con path a un directorio válido:

Pestaña de “Estado” → Botón “Guardar y salir” → selección del directorio donde se guardará el estado válido “/hola/quetal/”.
Resultado: Se escribe un archivo de tipo .estado binario que contiene la lista de productos, matriz de sinergias y distribución de la solución actual. Se cierra la interfaz y se detiene el programa.
Justificación: Guardar un estado entero puede ser práctico e incluso esencial. Además es una forma rápida de guardar y salir simultáneamente.

Caso importar estado con un path inválido:

Pestaña de “Estado” → Botón “Importar estado” → selección del path del archivo para importar el estado “/hola\guay.estado”.

Resultado: Error - El archivo seleccionado no existe o no es válido.
Justificación: El programa no puede leer archivos que no existen o que son ilegibles.

Caso importar estado con un path válido a un archivo de tipo incorrecto:

Pestaña de “Estado” → Botón “Importar estado” → selección del path del archivo para importar el estado “/hola/guay.txt”.
Resultado: Error - El archivo seleccionado debe ser de tipo “.estado”.
Justificación: El programa no puede leer archivos que no sean de tipo “.estado”.

Caso importar estado con un path válido a un archivo de tipo correcto:

Pestaña de “Estado” → Botón “Importar estado” → selección del path del archivo para importar el estado “/hola/guay.estado”.
Resultado: Se lee el archivo e importa la lista de productos, matriz de sinergias y distribución de la solución. En caso de ya existir alguno de estos datos se sobreescriben tras un aviso de la interfaz que permite cancelar la operación.
Justificación: Importar estados enteros puede ser práctico para replicar configuraciones enteras rápidamente.