Supermercado Digital

Distribución de productos en un supermercado

Creadores: Christian Conejo, Alex Lafuente, Pol Garcia y Daniel Mejias
Número de grupo: 43.5
Versión: 1.0

1. Introducción

Este programa, “Supermercado Digital”, es un sistema que permite a un usuario gestionar el conjunto de productos que serán usados en un supermercado.

Se trata de una estantería virtual la cual tiene una longitud infinita y contiene todos los tipos de productos que el usuario haya añadido. Con estos productos podrán ver sus características, cambiar sus nombres y borrarlos.

Este grado de similitud es el centro de una de las funcionalidades principales del sistema, ordenar la estantería. Una vez se ha configurado toda la información de la estantería, se pueden ordenar todos los productos de manera que la suma de todas las similitudes entre productos contiguos (incluyendo el primero y el último) en la estantería sea alta.

Al usar estos algoritmos hay dos opciones en la versión actual, algoritmo fuerza bruta, el cual da la ordenación más óptima posible de los productos, y algoritmo aproximado, el cual te dará una ordenación de productos eficiente pero no necesariamente máxima.

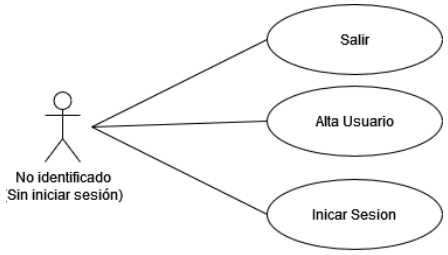
El programa también permite gestionar múltiples usuarios, mover objetos manualmente y cambiar el grado de similitud entre dos productos entre otras funcionalidades varias.

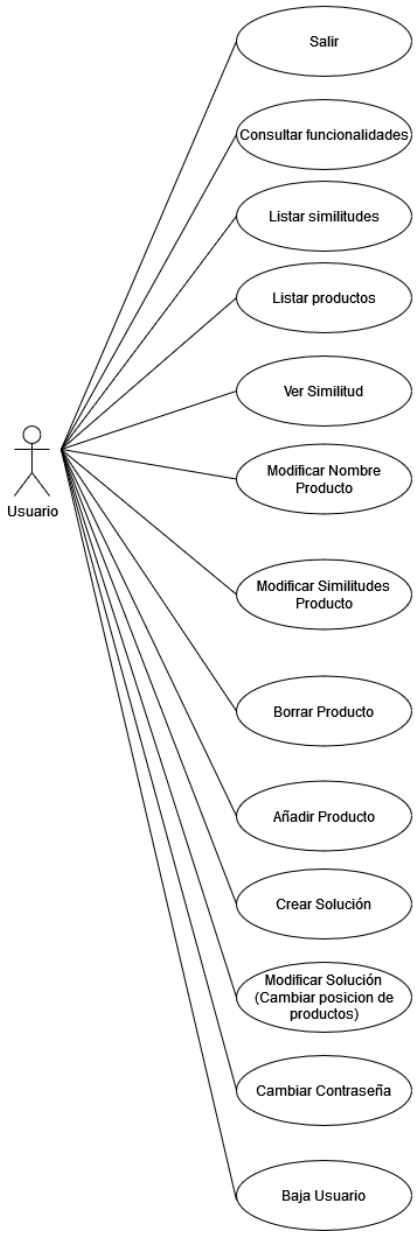
Para explorar más a fondo la funcionalidad de este programa, también se puede ver los archivos de código fuente, un JavaDoc generado, un UML que muestra las relaciones entre clases, múltiples juegos de pruebas y archivos de texto con explicaciones alrededor de todo el proyecto.

2. Casos de uso

2.1. Diagrama de casos de uso visual

Esto es el diagrama de casos de uso del programa en su versión 1.0:





2.2. Descripción casos de uso

Todos los casos de uso son obligatorios. En futuras entregas se podrán hacer modificaciones o adiciones de funcionalidades de cada caso de uso.

Modificar Nombre Producto

Actor: Usuario
Comportamiento:
1) El actor quiere cambiar el nombre de un producto.
2) El actor Introduce el nombre del producto a cambiar y el nuevo nombre.
3) El sistema se guarda los cambios.
Casos alternativos:
2a) No hay ningún producto con ese nombre: El sistema avisa del error y vuelve al paso 2), el actor reescribe el nombre del producto.
2b) Ya hay un producto con el nombre al que se quiere cambiar: Reescribir el nombre deseado.

Modificar Similitudes Producto

Actor: Usuario
Comportamiento:
1) El actor quiere cambiar el grado de correlación entre dos productos.
2) Escribe el nombre de un producto.
3) Escribe el nombre de otro producto.
4) Escribe el nuevo grado de correlación.

Casos alternativos:
2a) Si el producto no está definido en la aplicación. El sistema vuelve al paso 2), y pide el nombre del producto.
3a) Si el producto no está definido en la aplicación. El sistema vuelve al paso 3) y pide el nombre del producto.
4a) El usuario introduce un grado de correlación menor a cero o mayor a uno. El sistema vuelve al paso 2), y pide el nombre del producto.

Borrar Producto

Actor: Usuario
Comportamiento:
1) El actor quiere eliminar un producto.
2) Escoge el nombre del producto que quiere eliminar.

3) El producto de la estantería con ese nombre es eliminado.
Casos alternativos:
2a) Ese producto no está en el conjunto de la estantería: El sistema avisa del error y vuelve al paso 2), el actor reescribe el nombre.

Añadir Producto
Actor: Usuario
Comportamiento:
1) El actor quiere añadir un producto.
2) Escoge el nombre del producto que quiere añadir.
3) En la estantería se añade un producto con ese nombre y similitudes 0 a todos los otros productos.
Casos alternativos:
2a) El producto ya existe en el conjunto de la estantería: El sistema avisa del error y vuelve al paso 2), el actor reescribe el nombre.

Listar Productos
Actor: Usuario
Comportamiento:
1) El actor quiere ver los productos.
2) Muestra el nombre de todos los productos que tenemos.

Listar Similitudes
Actor: Usuario
Comportamiento:
1) El actor quiere ver los productos.
2) Muestra todas las similitudes entre productos que tenemos.

Ver Similitud
Actor: Usuario
Comportamiento:
1) El actor quiere ver la similitud entre dos productos.
2) Muestra la similitud.
Casos alternativos:
2a) Uno de los productos no existe: El sistema avisa del error y no muestra nada.

Crear Solución
Actor: Usuario
Comportamiento:
1) El actor quiere crear el conjunto de soluciones óptimas para los productos que tenemos almacenados.

2) El sistema devuelve una propuesta de estantería ordenada.

Casos alternativos:

1a) No hay ningún producto almacenado, por lo que el sistema da error y cancela la operación.

Modificar Solución

Actor: Usuario

Comportamiento:

1) El actor quiere modificar la estantería.
2) El sistema lista la estantería actual.
3) El usuario introduce un producto.
4) El usuario introduce otro producto.
5) El sistema intercambia ambos productos de lugar.

Casos alternativos:

3a) Si el producto a no está en la estantería. El sistema vuelve a pedir un nombre.

4a) Si el producto b no está en la estantería. El sistema vuelve a pedir un nombre.

Iniciar Sesión

Actor: No identificado

Comportamiento:

1) El usuario quiere entrar en el sistema
2) El sistema le pide al usuario que ingrese el nombre de usuario y la contraseña
3) El usuario ingresa su usuario y su contraseña

Casos alternativos:

4a) Si el usuario y la contraseña son correctos se entra al sistema

4b) Si el usuario es incorrecto. Vuelve a pedir el usuario y la contraseña.

4c) Si el usuario es correcto pero la contraseña incorrecta vuelve a pedir la contraseña.

Cambiar contraseña

Actor: Usuario

Comportamiento:

1) El usuario quiere cambiar la contraseña.
2) El sistema pide al usuario la contraseña actual.
3) El usuario introduce su contraseña actual.
4) El sistema pide la nueva contraseña.

5) El usuario introduce la nueva contraseña.
6) El sistema cambia la contraseña.

Casos alternativos:

3a) La contraseña introducida es errónea por lo tanto se informa al usuario y se muestran las funcionalidades a hacer.

Dar de baja usuario

Actor: Usuario

Comportamiento:

El usuario quiere eliminar su usuario del sistema
El sistema le pide la contraseña al usuario
El usuario introduce la contraseña
El sistema da de baja al usuario y le saca del sistema.

Casos alternativos:

3a) Si la contraseña es incorrecta el sistema vuelve a preguntar que funcionalidad quiere llevar a cabo.

Dar de alta un usuario

Actor: No identificado

Comportamiento:

1) El usuario quiere darse de alta en el sistema
2) El sistema le pide un nombre de usuario
3) El sistema le pide una contraseña al usuario.
4) El sistema da de alta al usuario y lo ingresa al sistema.

Casos alternativos:

2a) Si el usuario ya está registrado en el sistema le vuelve a pedir un nombre de usuario

Salir

Actor: Usuario o No identificado

Comportamiento:

4) El usuario quiere salir del sistema
5) El sistema cierra el programa

3. Explicación de las estructuras usadas

3.1. Estanteria Ordenada

Una estantería ordenada es una estructura dedicada a mantener la organización de un conjunto de ids (valores enteros únicos que representan productos), a la vez que el valor del grado de la similitud total con la que se relacionan estas.

Está constituida principalmente por un ArrayList que contiene las ids, y un HashMap de ids y sus posiciones correspondientes para poder acceder inmediatamente a la id del ArrayList.

Se puede acceder y eliminar una id de cualquier punto de la estructura mientras exista, pero únicamente se pueden añadir ids al final. También se puede ofrecer un ArrayList y el valor de su grado de similitud total directamente.

La estructura también cuenta con dos variables auxiliares de optimización. Una es la cantidad de ids de productos existentes. La otra es el valor del grado de la similitud total.

Al eliminar un producto de la EstanteriaOrdenada, su posición dentro del Array se ve intercambiada por una id -1, la cual sus funciones internas tienen en cuenta al momento de operar.

3.2. Gestor Estanteria

Un GestorEstanteria es una estructura que manipula un conjunto de productos y una estructura de EstanteriaOrdenada con el fin de utilizar las funciones a mayor nivel con el nombre de los productos en vez de ids. A su vez, comprueba la mayoría de errores.

3.3. Similitud

Una similitud es una pequeña estructura que contiene tres variables en su interior. Dos representan la id por la cual relaciona esta similitud, y la otra representa el valor del grado de similitud entre estas dos id.

Esta estructura existe con el fin de pasar información a otra clase. El gestor de similitudes no la usa internamente.

Una similitud siempre tendrá una id pequeña y una id grande con el fin de relacionar y optimizar. El valor del grado de similitud estará siempre entre 0 y 100.

3.4. Gestor Similitudes

Un Gestor de Similitudes es una estructura dedicada a mantener una organización eficiente en espacio y tiempo de las similitudes entre los productos.

Está constituida por un primer HashMap que tiene todas las ids (valor único entero representativo de un producto) como llaves a un segundo HashMap. El HashMap de cada llave contiene el conjunto de ids que son más pequeñas que la primera que a su vez están relacionadas con la primera id. Cada llave de este segundo hashmap interior lleva al valor de la similitud (entero) entre las dos ids. Nótese que al hablar de segundo HashMap se hace referencia al interior, y que la cantidad de HashMaps será igual a la cantidad de productos más uno. Un segundo HashMap solo contiene ids más pequeñas que la id que le ha llevado a él para evitar duplicidades de similitudes.

Inicialmente si no se encuentra una llave con la id pequeña dentro del primer HashMap, se considerará que el grado de similitud entre las dos id es 0, aunque un usuario puede añadir este valor de forma manual, y quedará definido en la estructura. Nótese que esto puede cambiar el resultado de un algoritmo aproximado.

3.5. Producto

Un Producto es una pequeña estructura dedicada a mantener la información única de un producto. Esta incluye dos variables por las cuales se define, el nombre (String) y la id (Integer).

3.6. Gestor de productos

Un Gestor de Productos es una estructura dedicada a mantener la unicidad de los productos.

Está constituida por: un HashMap que tiene todos los nombres (valor único de tipo String representativo de un producto) como llaves a sus productos correspondientes; una cola de prioridad de enteros que almacena cual es la pròxima id disponible a coger para un nuevo producto; y una variable que contabiliza el número de productos real.

3.7. Usuarios

Un Usuario es una estructura encargada de almacenar la información de un usuario. Esta información es; Su nombre de usuario, su contraseña y su rol.
El usuario está constituido por: Un String llamado nombre que representa el nombre del usuario. Un String llamado Password que representa la contraseña del usuario.

Un Tusuario llamado rol que representa el rol del usuario. El tipo Tusuario es un enum el cual tiene por valor EMPLEADO o GESTOR.

3.8. Gestor Usuarios

El gestor de usuarios es la estructura principal dedicada a realizar operaciones sobre los usuarios.

El gestor está constituido por: un Usuario llamado actual el cual tiene por valor el usuario actualmente registrado en el sistema. También tiene un HashMap llamado gestorUsers el cuál tiene por función tener una copia de todos los usuarios registrados en el sistema. (La clave primaria es un String que representa los nombres de usuario y el segundo atributo son Usuarios)

3.9. Algoritmo Ordenación

Algoritmo ordenación es la clase abstracta sin variables que se usa para representar cualquier clase que tenga una función “ordenar” que permita, tras recibir un conjunto de similitudes y productos como parámetros, devolver una lista de enteros que representan una ordenación de esos productos.

3.10. Algoritmo Fuerza Bruta

Esta es una subclase de algoritmo ordenación, un algoritmo fuerza bruta simplemente explora absolutamente todas las ordenaciones posibles de productos, y de entre todos devuelve el que de el valor de suma de similitudes más alta.

Se trata del algoritmo más lento, ya que al probar cada combinación posible, su coste temporal es esencialmente exponencial, pero también es el único que siempre dará una combinación con la máxima eficacia posible.

3.11. Algoritmo Aproximado Kruskal

Esta es una subclase de algoritmo ordenación, un algoritmo aproximado kruskal utiliza el algoritmo de kruskal para generar un MST a partir del conjunto de productos y similitudes, tratando productos como nodos y similitudes como aristas, todas las similitudes no especificadas siendo tratadas como aristas con valor 0.

Una vez genera el árbol, lo utiliza para generar un ciclo hamiltoniano que pase por todos las aristas del MST, siendo este ciclo la solución propuesta. La clase utiliza un conjunto de clases detalladas en la explicación del algoritmo para agilizar su funcionamiento.

Este algoritmo no siempre da el resultado más óptimo, aunque siempre dará como mínimo uno que sea la mitad de bueno, y es mucho más rápido que el fuerza bruta (decenas o centenas de veces más rápido con altas cantidades de productos).

3.12. Gestor Algoritmo

El gestor de algoritmos es la estructura centrada en mandar a hacer ordenaciones uno de los múltiples algoritmos que puede tener el programa.

La clase sólo contiene una lista de algoritmos (subclases de la abstracta algoritmo ordenacion), en la versión actual, dos algoritmos, uno de fuerza bruta y otro de aproximación. La única funcionalidad extra que tiene es detectar si dentro del conjunto de similitudes enviado para ordenar productos se menciona un producto que no exista, en cuyo caso reportará error en vez de llamar un algoritmo.

4. Explicación de los algoritmos usados

4.1. Introducción a los algoritmos

Para encontrar las mejores ordenaciones posibles entre todos los productos de la estantería, de tal manera que sean eficientes en cuanto a la similitud entre productos contiguos, hemos usado dos algoritmos ejecutados a través del GestorAlgoritmo, estos reciben una lista de similitudes y de ids de productos.

La lista de similitudes que recibe incluye solo una por cada relación entre productos, sí el producto 2 y el 3 tienen similitud 0.5, esta similitud solo aparece una vez en la lista, y no tiene las similitudes cuyo valor sea cero. Aunque las funciones pueden funcionar aunque ese no sea el caso (ignorando los repetidos y las similitudes con valor 0 por defecto).

Además cada vez que se ejecuta un algoritmo, este devuelve el valor resultante de sumar todas las similitudes entre productos contiguos (incluyendo el último con el primero) y uno de los órdenes que haya dado, ya que si ha encontrado múltiples soluciones con el mismo valor, solo devolverá uno de esos órdenes.

4.2. Algoritmo fuerza bruta

El primer algoritmo que se puede utilizar en el programa para encontrar una potencial combinación de productos es el algoritmo de fuerza bruta.

Este esencialmente explora todas las combinaciones posibles para la estantería y devuelve la que sea más eficiente, si hay más de una que de máxima eficiencia, solo devolverá una de ellas.

Como se puede ver en la clase, la implementación usa solo dos funciones: La primera, ordenar, que devuelve el resultado de la segunda función y la segunda, subAlgoritmoFuerzaBruta, que es la función recursiva que encuentra la mejor solución y a la vez devuelve el valor de la suma.

La función recursiva recibe:

Las similitudes, que no cambiarán y usará para saber cuanto tiene que sumar al terminar (entre el último elemento de la lista más eficiente que haya encontrado en la siguiente recursión y el elemento que va a añadir esa call específica).

Los productos, lista de la que hace una copia recortando el elemento que se "tanteará" en la siguiente recursión, esto se hace para que en una dada recursión, no intente poner un elemento en la lista solución que ya se haya puesto antes. Se puede considerar una lista de elementos que todavía se tienen que poner en la solución.

La "solución", que se trata de una lista que contiene el orden que se está generando, en la capa más superior de la recursión, la lista está vacía, y cuando se llega a un caso base, tiene una ordenación posible.

La función primero prueba cada una de las posibilidades usando los parámetros de la forma especificada, y va subiendo capas de recursión escogiendo la solución con el valor más alto de las propuestas por las siguientes recursiones hasta llegar a la llamada original y devolver la mejor solución posible.

Este algoritmo no es muy eficiente, tiene un coste exponencial, pero es el único algoritmo garantizado para dar la mejor solución posible.

4.3. Algoritmo aproximado

El segundo algoritmo empleado es una aproximación que utiliza MSTs y DFSs para generar una solución que dará como mínimo una suma de similitudes que sea la mitad de la máxima posible.

La primera función llamada es ordenar, la cual recibe los datos y llama a dos funciones, primero, algoritmoKruskal, la cual genera un MST a partir del árbol implícito generado por los productos y similitudes (se asume que entre productos

sin similitud especificada, hay similitud cero), y segundo, crearCicloHamiltoniano, la cual genera la solución a base hacer un dfs del MST y además sumar las similitudes que generaría esa solución.

La función algoritmoKruskal recibe los productos y similitudes y utiliza el grafo implícito para generar un MST en forma de un ArrayList de similitudes. Este algoritmo se basa en ordenar las similitudes e ir añadiendolas al MST de grande a pequeña hasta haber cubierto todos los nodos, con la particularidad de que se tienen que saltar todas las “aristas” (similitudes) que generen un ciclo en el MST que se está formando, para que se genere un árbol puro.

Para la implementación de este algoritmo se utilizan dos clases auxiliares:
CompararPorSimilitud, la cual solo se usa como clase que implementa Comparator<Similitud> poder ordenar los objetos Similitud en orden decreciente según su valor de similitud, ya que Similitud es una clase nuestra y no una primitiva.

DisjointSet, esta es un Union-Find Set, es decir, un array de ints tal que cada posición del array puede indicar un grupo al que pertenece y pueden unirse entre ellos. Esta clase se usa para representar los “nodos” (productos) durante la creación del MST, cada vez que se unen dos productos en el MST, se unen en el DisjointSet, y si al unir dos nodos se ve que ya pertenecían al mismo grupo, significa que al unirlos se generaría un ciclo. Así se pueden encontrar potenciales ciclos de manera eficiente.

La segunda función empleada para encontrar la solución final es crearCicloHamiltoniano. Está primero coge todas las similitudes del MST y los añade a una instancia de ConjuntoProductosConectados, usa esa instancia para llamar a la función doDFS (la cual hace un DFS normal) y al terminar, usa el orden generado por el DFS para encontrar la suma que darían todos los productos.

ConjuntoProductosConectados, junto con ProductoConectado, son las dos clases auxiliares que utilizan crearCicloHamiltoniano y doDFS. La primera simplemente es un HashMap de la segunda, y la segunda contiene una lista de ids representativa de los productos con los que se conecta este producto. Se usa sobre todo en el DFS para poder saber rápidamente con qué productos conecta un producto específico en el MST generado.

Una vez se genera la solución aproximada con el uso de crearCicloHamiltoniano, esta se devuelve directamente en ordenar.

5. Acceso a JavaDoc

La documentación de la estructura, clases y métodos del proyecto, se encuentra organizada en un JavaDoc. Para acceder a este, basta con dirigirse al directorio principal de la entrega, “Entrega 1”. Una vez dentro, se debe entrar a la carpeta “JavaDoc”, y finalmente abrir el documento “index.html” en cualquier navegador.