Estructures de dades i algorismes

Equip 43.3

Tercera entrega

Professor: Jose Miguel Urquiza

Membres del grup:

Lluc Santamaria Riba
lluc.santamaria

Efraín Tito Cortés
efrain.tito

Alejandro Lorenzo Navarro
alejandro.lorenzo

Eulàlia Peiret Santacana
eulalia.peiret

Índex

1. Introducció

En aquest projecte hem desenvolupat un sistema innovador per a supermercats que permet als usuaris gestionar un catàleg de productes i organitzar-los de manera òptima en prestatgeries. L'objectiu principal és maximitzar els beneficis derivats de les similituds entre productes mitjançant la implementació de tres algorismes diferents que ofereixen distribucions òptimes.

En aquest document es troba una descripció de les estructures de dades i algorismes utilitzats per a implementar les funcionalitats principals de tot el projecte.

Les funcionalitats extres i les millores respecte a la versió del primer lliurament es pot trobar en el document “Descripció de les classes”.

1.1 Canvis i millores respecte a les estructures de dades de la primera entrega

Respecte a la primera, hem millorat l’estructura de dades de les solucions. Ara tenim les solucions en forma de prestatgeria internament també: la solució és una matriu de strings on cada string representa un producte i cada fila de la matriu un prestatge.

1.2 Canvis i millores respecte a els algorismes de la primera entrega

Donat que els algorismes es troben en la capa de domini, no s’han introduït nous algorismes per aquesta entrega. Principalment, s’han fet algunes millores en l’algorisme voraç. També, s’ha posat èmfasi a discutir perquè considerem que les estratègies, les estructures de dades i els passos escollits per construir els algorismes són correctes dins el conjunt d’alternatives a la disposició. Finalment, s’han afegit els costos temporals de cadascun dels passos que resolen el problema, comparant-los amb les alternatives.

2. Estructures de dades

2.1 Control i creació de catàleg i productes

En aquesta aplicació, representem la prestatgeria dins la classe de controlador del catàleg, “CtrlCataleg”, utilitzant un *ArrayList* de la llibreria d'utilitats de Java. Aquest *ArrayList* conté instàncies de *Productes*, que pertanyen a la classe “Producte”.

Per representar les similituds de cada instància de producte, es fa ús d’un altre *ArrayList* de tipus *Double*, amb un valor per a cada producte dins del catàleg, incloent-hi el mateix producte. En aquest cas, el valor associat d’un producte amb ell mateix serà 0.

El resultat de combinar tots dos *ArrayList* és una matriu N×N, on N és el nombre de productes del catàleg. Cada element de la matriu indica el valor de la similitud entre dos productes, de manera que l’element *[i][j]* representa la similitud entre el producte amb índex *i* i el producte amb índex *j*. De la mateixa manera, l’element *[j][i]* tindrà el mateix valor, ja que la similitud és simètrica.

Per tant, aquesta matriu de similituds és una matriu simètrica amb una diagonal nul·la. El valor nul es representa amb un 0.0, indicant que no existeix cap similitud entre un producte i ell mateix.

Un exemple d’aquesta implementació:

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

Figura 4.1: Matriu similituds, exemple

La matriu *double[][]* és un element clau, ja que és utilitzada pels algorismes per calcular les distribucions. Aquesta matriu, combinada amb l'atribut *índex* de cada producte, permet optimitzar els càlculs.

L’atribut *índex* emmagatzema la posició de cada producte dins de l’*ArrayList* del catàleg i segueix el mateix ordre que la matriu de similituds. Això significa que el producte situat en la posició *i* del catàleg també correspon a la fila i columna *j* de la matriu de similituds.

Aquesta correspondència permet reduir costos computacionals en diverses funcions, ja que no cal fer recorreguts lineals per accedir als valors associats a un producte. Si es coneix l’índex d’un producte, es pot accedir directament tant a la seva informació al catàleg com al seu valor de similitud dins la matriu. Això resulta especialment útil durant l'execució dels algorismes que treballen amb la matriu *double[][]*, ja que millora l’eficiència i redueix el temps de càlcul.

2.1.1 Costos d’operacions bàsiques del controlador del catàleg:

Els productes tenen un nom i un índex. Per fer qualsevol cerca d’un producte, si tenim l’índex el cost és constant. Si tenim el nom, el cost serà O(n), ja que haurem de recórrer tot l’*ArrayList* de productes per obtenir-ne l’índex.

Cost d’afegir producte:

Cas inicial: amb un catàleg buit, el cost és constant, ja que només s’afegeix un únic producte al *ArrayList*. Quan hi ha productes al catàleg, l’usuari proporciona el nom del nou producte i una llista amb les similituds en forma de *Pair*: <valor de la similitud, nom del producte>. Per a cada element de la llista (hi haurà tants elements com productes al catàleg), cal buscar el seu índex per tal de poder relacionar el producte amb el nou a la matriu de similituds. Per tant, el cost és $O(N^2)$.

Cost d’eliminar producte:

Per eliminar un producte es produeixen dos recorreguts lineals del *ArrayList* de productes, la primera per eliminar la relació de similitud de cada producte amb el producte a eliminar, i la segona el recorregut per actualitzar el valor de l'atribut “índex” de cada producte. Cost final del mètode O(N).

Cost d’editar similituds

Gràcies a mantenir informació dels índexs de cada producte, un cop obtinguts, aquest mètode es fa de manera constant. Amb dues crides de O(1).

2.2 Restriccions

Amb una subclasse de “CtrlCataleg”, es crea un nou controlador de productes que permet tenir restriccions entre productes, impedint que dos productes pugin estar junts en la distribució . Les restriccions es representen amb una matriu de booleans NxN, on N és el número de productes al catàleg. Per un producte amb índex *i* i un producte amb índex *j*, existeix una restricció entre ambdós productes sí en la matriu la posició [i][j], és true. Simètricament, la posició [j][i] també serà true.

2.2.1 Costos d’operacions bàsiques del controlador amb restriccions

El cost de les operacions d’afegir i eliminar restriccions, és igual al d'afegir i eliminar productes, ja que la informació de les restriccions es guarda en una matriu amb el mateix comportament que la matriu de similituds entre productes.

2.3 Control i creació de prestatgeria i distribucions

Les distribucions són instàncies de la classe “Solucio”. Cada solució compta amb un nom únic i una estructura de matriu que representa la prestatgeria, on s'ordenen i col·loquen els productes de manera determinada. Cada fila representa un prestatge.

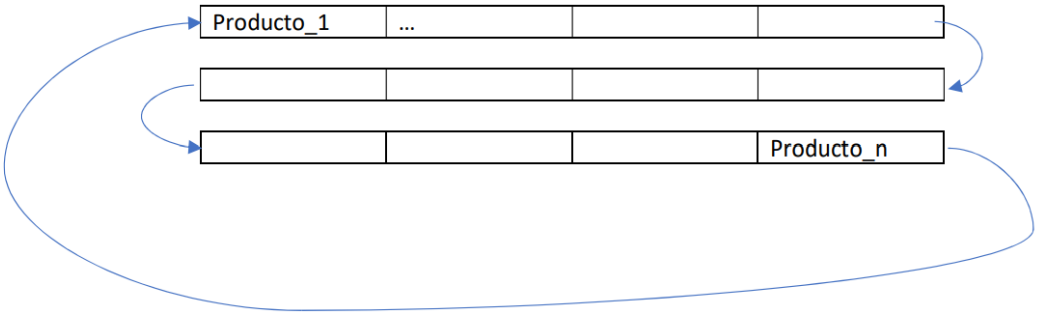

Figura 4: estructura de la prestatgeria

La classe “CtrlSolucio” és responsable de gestionar el procés de generació, importació i exportació de distribucions per als productes del catàleg. En qualsevol moment, l’usuari pot iniciar aquest procés i seleccionar entre diverses opcions segons les necessitats específiques.

Una de les opcions que el sistema ofereix és escollir el tipus d’algorisme que s’utilitzarà per buscar la millor distribució dins d’una prestatgeria. Es detallen amb profunditat a l’apartat 5. La classe “CtrlSolicio” sempre té una instància d’un “Algorisme”. Si l’usuari no especifica quin tipus vol, tindrà una instància d’”Aproximacio”, per defecte.

Quan l’usuari vol crear una solució, “CtrlSolucio” demana a “CtrlCataleg” la matriu de similitud entre productes. Amb això, la instància d’”Algorisme” pot calcular la millor distribució. El resultat que torna l’algorisme permet crear una instància de la classe “Solucio”. Totes les solucions creades tenen nom únic, una instància de l’algorisme que l’ha calculat i un *ArrayList* de *Strings* que representa els productes de la prestatgeria.

El “CtrlSolucio” guarda en un *ArrayList* totes les instàncies de “Solucio” . Un cop creades una o més solucions, l’usuari pot voler modificar-ne alguna. El sistema permet realitzar modificacions mitjançant l’intercanvi de productes dins d’una solució concreta.

Quan es realitza un intercanvi, el “CtrlSolucio” elimina la instància de la solució original i crea una nova instància de “SolucioModificada”, una subclasse de “Solucio” dissenyada per gestionar aquestes modificacions. D’aquesta manera, el sistema assegura que cada modificació queda registrada.

El sistema també permet eliminar solucions existents. L’usuari ha d’especificar el nom de la solució que desitja suprimir. El “CtrlSolucio” localitza la solució corresponent al ArrayList on s’emmagatzemen totes les solucions del sistema i l’elimina.

Aquestes funcionalitats fan que la classe “CtrlSolucio” sigui una eina flexible i poderosa per gestionar la vida útil de les distribucions al sistema. A continuació analitzarem el cost de les diferents operacions.

2.3.1 Costos d’operacions bàsiques del controlador de solucions

Cost de crear una solució:

El cost de crear una nova solució és proporcional al nombre de solucions existents al sistema, ja que cal verificar que el nom de la nova solució sigui únic. A aquest cost s’hi afegeix el temps d’execució de l’algorisme seleccionat per resoldre el problema, i el cost de traduir els índexs dels productes retornats per l’algorisme als noms dels productes mitjançant “CtrlCataleg”, un procés que depèn del nombre de productes inclosos en la solució. Per últim, es fa un últim recorregut per tots els productes de la solució per crear la matriu on cada fila simula un prestatge.

Alternativament, també es pot crear una solució passant la matriu de productes ja feta. Tot i així el cost no disminueix perquè s’ha de comprovar igualment que el nom sigui únic i que la matriu sigui vàlida: no tingui productes duplicats i tots els prestatges tinguin la mateixa quantitat de productes.

Cost total = #solucions_al_sistema + #productes_de_la_nova_solucio

Cost d’eliminar una solució:

El cost d'eliminar una solució és lineal amb relació al nombre de solucions existents al sistema, ja que en el pitjor cas el sistema ha de comprovar que la solució a eliminar existeix. Aquesta comprovació es fa mitjançant una cerca al ArrayList de solucions, recorrent-lo element per element fins a trobar la solució desitjada. Per tant, el temps de processament dependrà directament de la quantitat de solucions al sistema.

Cost total = #solucions_al_sistema

Cost de modificar una solució:

El cost de modificar una solució si ho intentes fer amb el nom dels productes serà proporcional al nombre de solucions existents al sistema i al nombre de productes que té la solució a modificar. Això és així perquè, per realitzar la modificació, s’ha de revisar que la solució contingui els dos productes indicats per l’usuari. Aquest procés implica una cerca de les solucions al sistema i una verificació dels productes dins de la solució específica, fet que fa que el cost depengui tant de la quantitat de solucions com de la mida de la solució a modificar.

Per millorar el cost d’aquesta funció hem implementat una funció amb la que pots intercanviar dos productes a partir de les seves posicions en la matriu. Utilitzar els índexs fa que el cost de trobar els productes i veure si són vàlids passa a ser constant (només cal accedir a l'element de la matriu).

Cost total amb la millora = #solucions_al_sistema

Cost de guardar les solucions a un fitxer:

El procés per guardar les solucions en un fitxer implica diversos passos. Primer, es recorren totes les solucions existents al sistema. Per a cada solució, es recorre la llista de productes que conté per tal

d'obtenir el nom de cada producte i l'ordre en què estan col·locats. A més, es guarda si la solució ha estat modificada o no, i el seu nom.

Un cop recopilada tota aquesta informació, es converteix en una cadena de text (String) que es passa al controlador de persistència (“CtrlPersistenciaSolucio”) per ser emmagatzemada. Aquesta conversió es realitza al controlador de domini (“CtrlSolucions”) per evitar l’intercanvi d’instàncies complexes entre classes. D’aquesta manera, entre classes només es comparteixen cadenes de text i estructures de dades simples.
Cost total = #solucions_del *sistema* × *#productes*per_solució.

Cost de carregar solucions des d’un fitxer:

El procés per carregar solucions des d’un fitxer és similar al de guardar-les, però en sentit invers. Primer, el controlador de persistència (“CtrlPersistenciaSolucio”) llegeix el fitxer i verifica que les dades estiguin en el format correcte. Aquesta verificació inclou assegurar que el fitxer compleix amb l’estructura esperada i que les dades són vàlides.

Un cop verificades, les dades es converteixen en solucions individuals dins del controlador de persistència. Posteriorment, les solucions es passen al controlador de domini (“CtrlSolucions”) una a una, on es valida que siguin consistents i coherents amb el sistema. Per exemple, no es permet afegir dues solucions amb el mateix nom.

El cost total d’aquest procés també depèn del nombre de solucions que es volen carregar i del nombre de productes que conté cadascuna. Es pot expressar com:
Cost total = #solucions_a_afegir × #productes_per_solució.

3. Algorismes

3.1 Algorisme d'aproximació

En aquest projecte, hem de trobar una distribució de la prestatgeria que maximitza els graus de similituds. Aquest problema es pot considerar una variant del famós problema Travelling Salesman Problem (TSP). Sabem que el TSP és un problema NP-complet on trobar l'òptim té cost exponencial. És per això que l'algorisme d'aproximació permet obtenir una solució subòptima amb una garantia de qualitat de la solució i en un temps polinòmic. En el nostre cas, l'algorisme emprat és 2-Aproximació i garanteix un cost com a molt dues vegades pitjor que l'òptim global. L'entrada de l'algorisme és la matriu de similituds (equivalent a una matriu adjacència) de talla $n \times n$. La sortida és una llista de la forma $(v_1, v_2, \ldots, v_n)$ on el cicle $v_1 v_2 \ldots v_n v_1$ correspon a un cicle hamiltonià que maximitza (dins la suboptimalitat) el grau de similitud de la prestatgeria. Durant tot l'algorisme fem ús de l'estructura de dades Pair<T1, T2> per representar un parell de dades de tipus qualsevol (per exemple, Pair<Integer, Double>). L'algorisme permet prioritzar no tenir dos vèrtexs amb restricció contigus, però no pot garantir-ho. Treballant sobre la matriu de similituds com si fos un graf d'ordre n i mida m amb alguna restricció sobre arestes, els passos de l'algorisme són els següents:

1. **Ordenar les arestes per pes creixent**

   Per ordenar les arestes creixentment, discutirem sobre quin és l'algorisme més adient. L'entrada és un graf $G = (V, E, \omega)$ on V són els vèrtexs, E són les arestes i ω: $E \rightarrow [0, 100]$ és la funció que assigna pes a les arestes. La sortida ha de ser $E' = \{e_1, e_2, \ldots, e_{|E|}\}$ tal que $\omega(e_1) \leq \omega(e_2) \leq \ldots \leq \omega(e_{|E|})$.

   Tinguda la talla de l'entrada $|V| + |E|$ la qual no està acotada, no podem aplicar l'ordenació digital (per exemple Counting Sort). Per tant, sabem que l'ordenació ha de ser per comparacions. D'algorismes d'ordenació per comparacions n'hi ha molts, cal escollir el més eficient i que millor s'adapti. Per resoldre aquest problema, considerarem els més usats a causa de la seva senzillesa i eficiència; el mergesort, el quicksort, el selectionsort i el bubblesort. Destaquem el selectionsort (en cada iteració es selecciona el menor) i bubblesort (els de més grans pugen mitjançant intercanvis) com els més senzills d'implementar, tanmateix, el seu cost temporal és $O(|E|^2)$ mentre que mergesort ens assegura $\Theta(|E|log|E|)$. El mergesort (basat en dividir i vèncer) ens assegura cost $\Theta(|E|log|E|)$, mentre que quicksort (basat en ordenació per pivot i dividir i vèncer) només en mitjana ens assegura cost $O(|E|log|E|)$. Podria donar-se algun cas en que quicksort té cost $O(|E|^2)$. De totes maneres, el quicksort surt guanyant en mitjana, ja que el mergesort sempre realitza tots els passos d'ordenació. Per exemple, considerem un vector ordenat al 97%, el mergesort farà tots els passos d'ordenació independentment del factor d'ordenació, mentre que el quicksort aprofitarà que el vector ja està força ordenat per fer pocs passos d'ordenació. Així doncs, com en el nostre projecte preferim més velocitat en mitjana sense importar-nos massa que en alguns casos trigui més, utilitzarem quicksort.

   Els passos per l'ordenació amb quicksort són els següents:
   Primer de tot, es creen les m arestes com a parells Pair<Integer, Integer> i es guarden en una llista. No es creen les arestes que tenen una restricció, assegurant que en el següent pas (Kruskal) l'arbre no conté arestes amb restricció. A continuació, s'aplica l'ordenació

QuickSort fonamentada en la tècnica dividir i vèncer. Per fer la partició, on es selecciona un pivot i es deixen els menors a l'esquerra i els majors a la dreta, s'utilitza partició de Hoare (el pivot és el primer element). Així obtenim la llista ordenada per pes creixentment.

2. Arbre d'expansió màxim

Amb les arestes ordenades, el següent pas és aplicar un algorisme que troba l'arbre d'expansió màxim. L'entrada és un graf $G = (V, E', \omega)$ on V són els vèrtexs, $E' = \{e_1, e_2, ..., e_{|E|}\}$ són les arestes i ω: $E' \rightarrow [0, 100]$ és la funció que assigna pes a les arestes. Les arestes compleixen que $\omega(e_1) \leq \omega(e_2) \leq ... \leq \omega(e_{|E|})$. La sortida ha de ser el conjunt d'arestes $F \subseteq E'$ tals que formen l'arbre d'expansió màxim en G.

Els principals algorismes per trobar arbres d'expansió màxim són l'algorisme de Janik-Prim i l'algorisme de Kruskal els quals tenen una complexitat temporal $O(|E'|)$, ja que les arestes ja es troben ordenades. Tanmateix, creiem oportú usar Kruskal per aprofitar la documentació proporcionada pel professorat i poder fer ús del Merge Find Set.

Així doncs, podem aplicar l'algorisme de Kruskal per tal de trobar les $n - 1$ arestes que formen l'arbre de cost màxim. L'algorisme de Kruskal és un algorisme voraç que afegeix l'aresta de màxim pes que no forma un cicle. Per determinar si una aresta forma un cicle, la forma més eficient dins de les alternatives és fer ús de l'estructura de dades Merge Find Set (també conegut com a Union Find o Particions). El Merge Find Set implementat manté una partició dels vèrtexs en components connexos i suporta dues operacions: Unir i Buscar. Buscar cerca el representant del subconjunt d'un vèrtex. Unir retorna fals si els vèrtexs a unir són del mateix component connex; uneix els subconjunts i retorna cert en cas contrari. Hem utilitzat tècniques de compressió de camins i unió per talles tal com s'explica en la informació addicional d'algorismes del professorat per tal de simplificar la complexitat temporal.

3. Construcció del digraf i cerca del cicle eulerià

El següent pas és construir un digraf on tota aresta de l'arbre d'expansió màxim es substitueix per dos arcs en sentits oposats. Escrit de forma matemàtica, donat l'arbre d'expansió màxim $T = (V, F)$ on V són els vèrtexs i F són les arestes s'ha de construir $T' = (V, F')$ on per a tota aresta $f = (u, v) \in F$, $(u, v) \in F'$ i $(v, u) \in F'$. És a dir, totes les arestes de F més les inverses de F. Per representar el digraf hem utilitzat llistes d'adjacència implementades amb una llista de sets (List<Set<Integer>>), ja que permeten fer una fàcil inserció a l'hora que no permeten arestes duplicades.

A continuació, cal trobar un cicle eulerià de la forma $C = v_1 v_2 ... v_n v_1$. Els dos principals algorismes fonamentals de la teoria de grafs que permeten fer-ho són el Depth-First Search (DFS) i el Breadth-First Search (BFS). L'única diferencia és que el DFS fa una exploració en profunditat utilitzant una estructura de dades tipus LIFO mentre que BFS fa una exploració en amplitud utilitzant una estructura de dades tipus FIFO. Tots dos algorismes tenen una complexitat temporal $O(|V| + |F|)$. Per implementar l'algorisme em considerat adient utilitzar DFS, ja que podem fer ús de l'estructura implicita LIFO de les crides recursives a funcions.

Així doncs, per construir el cicle eulerià hem utilitzat l'algorisme Depth-First Search (DFS) que realitza una exploració en profunditat partint del vèrtex 0. Tant quan s'explora un nou

vèrtex com quan fa backtracking (retorn d'una crida recursiva) s'afegeix un vèrtex al cicle eulerià. Finalment, el DFS retorna una llista (List<Integer>) amb els vèrtexs del cicle eulerià.

4. **Simplificació en un cicle hamiltonià**

   Per acabar, l'algorisme simplifica el cicle eulerià en hamiltonià. El document d'informació adicional proposa tres alternatives diferents. Encara que la tercera és la més rebuscada, també és la que dona millors resultats. És per això que hem considerat la tercera opció. L'esquema d'aquest algorisme és: Dur a terme varis passos de simplificació. En cada pas buscar el conjunt de nodes repetits consecutius que, al eliminar-los, donen el cicle amb menor cost. Repetir aquest procés fins que no quedin nodes repetits.
   Per fer-ho, recorrem el cicle eulerià guardant parells $((u, v), w)$ on (u, v) són dos índexs del cicle tals que tot vèrtex entre u i v és una repetició i w és la diferència de pes que s'obtindria d'eliminar la repetició. L'estructura de dades emprada és Pair<Pair<Integer, Integer>, Double> perquè permet accedir als camps de forma fàcil i en temps constant. Seguidament, eliminem la repetició que dona un cicle amb pes màxim, afegim l'aresta que tanca el cicle i repetim el procés (tornem a recórrer el cicle buscant repeticions). Notar que l'aresta afegida pot tenir una restricció, per tant, l'algorisme no pot garantir evitar totes les restriccions. Acabem quan hi ha exactament $n + 1$ vèrtexs en el cicle. Aquest cicle és el cicle hamiltonià buscat, per tant, hem acabat.

3.2 Algorisme backtracking

El segon algorisme implementat és un algorisme de backtracking (classe "AlgorismeBT") que busca exhaustivament la millor distribució dels productes en una prestatgeria. És l'únic dels tres que ens assegura retornar la millor solució. L'objectiu és trobar una configuració que maximitzi la suma de totes les similituds entre els productes consecutius de la prestatgeria. Per aconseguir-ho, l'algorisme es basa en una matriu de similituds i una matriu de restriccions, totes dues les rep com a paràmetres.

L'algorisme explora exhaustivament totes les combinacions possibles dels productes i guarda en un vector *millorConfiguracio* la millor solució trobada fins al moment. Aquesta cerca exhaustiva es duu a terme mitjançant una funció recursiva.

El procés recursiu funciona de la següent manera:

1. Exploració de combinacions: A cada pas, l'algorisme afegeix un producte a la configuració actual i marca aquest producte com a visitat.
2. **Cas base:** Quan la configuració actual conté tots els productes, es calcula la suma total de les similituds:
   - Inclou la similitud entre el primer i l'últim producte de la configuració (per tancar el cercle).
   - Si aquesta suma supera la suma de similituds de la millor configuració trobada fins al moment, actualitza el vector *millorConfiguracio* i la màxima similitud.
3. **Cas recursiu:** Quan la configuració encara no està completa:
   - L'algorisme afegeix iterativament els productes no visitats i crida recursivament la funció amb aquesta nova configuració.

- Al final de cada branca, es torna enrere desfent els canvis per explorar la següent possibilitat.

Donat que comprovar totes les possibles distribucions té un cost factorial, s'han introduït les següents optimitzacions per reduir el temps de càlcul:

1. **Suma acumulada de similituds:**
   La funció recursiva inclou un paràmetre que conté la suma acumulada de les similituds fins al punt actual. Això evita haver de recalcular la suma en cada crida, millorant significativament l'eficiència.
2. **Poda per optimalitat:**
   Abans d'explorar una branca, l'algorisme estima el valor màxim possible que es podria aconseguir en aquella branca, assumint que la resta de productes tenen la màxima similitud possible (100.0).
   Si aquesta estimació no supera la millor similitud trobada fins al moment, l'algorisme descarta la branca completament, estalviant temps de càlcul.

En situacions on les restriccions són massa estrictes i impedeixen trobar cap configuració vàlida, l'algorisme llença una excepció (*FormatInputNoValid*). Això indica que no existeix cap solució viable amb les condicions proporcionades.

Sobre el cost de l'algorisme:

1. Totes les permutacions

Té una complexitat base $O(n!)$, ja que l'algorisme explora totes les permutacions possibles dels productes, generant totes les configuracions en les quals es pot ordenar els n productes. Per tant, el nombre màxim de branques explorades és el nombre de permutacions de n. La poda per optimalitat, en cas pitjor, podria no descartar cap branca, a més a més. Determinar si afecta al cost promig de l'algorisme dependria de quin són els rangs de similituds més habituals, si n'hi ha: conjunts amb similituds extremes tendeixen a complir la condició de la poda millor que conjunts amb similituds baixes. Un escenari habitual amb un nombre elevat de restriccions també pot reduir el cost promig, ja que moltes branques es descartarien.

2. Per branca

Les operacions que es donen durant cada crida recursiva tenen un cost constant $O(1)$, donat que son accessos a elements determinats de les matrius i càlculs puntual. Però, degut a l'existència d'una poda per optimalitat, cal calcular la suma de la similitud dels elements en cada solució parcial, en lloc de fer-ho únicament una vegada per branca. Això comportaria un cost $O(n^2)$, versus el cost de mantenir-ne una variable acumuladora per la similitud de la solució parcial que permet que no sigui necessari recórrer tots els seus elements buscant la suma per cada parell de productes consecutius en cada pas, sinó només el de l'última parella, que és $O(n)$.

3. **Total**

   Així doncs, la complexitat de l'algorisme és $O(n * n!)$, un temps força elevat i computacionalment costós. Tot i així, l'existència d'aquest algorisme resulta convenient per poder determinar la solució òptima global.

3.3 Algorisme Greedy

L'últim algorisme implementat és un algorisme voraç desenvolupat a la classe "AlgorismeGreedy". Es basa en seleccionar iterativament els productes amb major similitud amb l'últim producte afegit a la solució, tractant de respectar les restriccions donades.

Aquest mètode tria les opcions que semblen millors en cada pas, fet que redueix el temps de càlcul a costa de perdre garanties d'òptim global.

Per executar aquest algorisme, calen diversos paràmetres d'entrada: una matriu de similituds que representa la similitud entre cada parella de productes. El valor *matriuSimilituds[i][j]* indica la similitud entre els productes *i* i *j*. Una matriu de restriccions consecutives: indica quins productes no haurien de col·locar-se l'un al costat de l'altre. Si *matriuRestrConsec[i][j]* és true, el producte *j* no pot anar després del producte *i*. L'índex del producte inicial: producte amb el qual es comença la distribució a la primera iteració. I per últim el nombre d'iteracions: nombre de vegades que es repeteix l'algorisme començant per productes inicials diferents, en seqüència consecutiva ascendent per índex.

L'algorisme funciona de la següent manera: comença seleccionant un producte inicial i, a partir d'aquest, va triant iterativament el següent producte que té una elevada similitud amb l'últim producte afegit a la configuració. Aquest procés es repeteix fins que s'han col·locat tots els productes en una solució provisional.

Aquest procediment es realitza un nombre determinat de vegades, mai superior al nombre total de productes. A cada nova iteració, es canvia el producte inicial pel següent en ordre ascendent d'índex, avançant consecutivament en la llista de productes.

Si durant qualsevol iteració es troba una solució amb una similitud total millor que la millor solució trobada fins al moment, aquesta es guarda com la nova solució òptima.

Per optimitzar aquest procés, l'algorisme utilitza un acumulador per mantenir la suma de similituds durant de la configuració actual, evitant recalcular-la en cada pas. També implementa una poda per optimalitat, descartant les iteracions en les quals el valor màxim possible de similitud des del punt actual (assumint la màxima similitud possible per als productes restants) no supera la millor solució trobada fins al moment. Finalment, es prioritzen les solucions que respecten totes les restriccions sobre aquelles que no les compleixen però tenen una major similitud total, encara que l'algorisme pot donar una solució que incompleix alguna restricció si no en troba cap que les respecti totes.

1. **Procés de selecció**

   Com a millora, en lloc de fer un recorregut lineal entre la resta de productes disponibles per tal de trobar aquell amb la màxima similitud possible, ara es segueix un esquema de selecció

(basat en el problema de la secretària): Dividim els candidats (N) en dos conjunts: “preferits”, que podem seleccionar lliurement, i “restringits”, que només podem seleccionar en última instància. En primer lloc, avaluem els primers ⌊N/e⌋ candidats (aproximadament el 37% del total), sense seleccionar-ne cap. Durant aquesta fase, registrem aquell amb la màxima similitud entre els candidats de “preferits”, així com aquell amb la màxima de “restringits”. A continuació, fem una búsqueda lineal on enregistrem el producte que tingui més similitud del subconjunt restant de restringits que aquell que ha estat abans registrat, si n’hi ha. La búsqueda s’atura quan es trobi un producte amb més similitud que aquell enregistrat del primer subgrup de preferits. Si no hi ha hagut cap amb aquesta condició, es selecciona el que havia estat enregistrat. En el cas de que el conjunt “preferits” sigui buit, es selecciona el producte de “restringits” que hagi estat enregistrat. Finalment, entre iteracions, l’algorisme prioritza aquelles potencials solucions on no s'incompleix cap restricció.

Segons dicta el problema, la probabilitat de trobar el millor candidat a la segona fase és $1/e$. Però, en aquest cas, es guarda també el millor element de la primera fase. Així doncs, la probabilitat de trobar el millor candidat equival a la probabilitat que el millor candidat estigui als primers m candidats, multiplicada per la probabilitat d’èxit en aquest cas, més la probabilitat que el millor candidat estigui als candidats posteriors, multiplicada per la probabilitat d'èxit en aquest cas: $((m/n) * 1) + (((n - m)/n) * 1/e)$, on si substituïm $m = n/e$ trobem una possibilitat d’un 60%. S’ha de tenir en compte que això és aplicable només al cas on hi hagin candidats preferits en ambdues fases.

2. Cost

El cost de l’algorisme augmenta linealment amb el nombre d’iteracions. Té una complexitat quadràtica que depèn del nombre de productes a avaluar. Concretament, si l’algorisme rep com a paràmetre 1 iteració, el cost és el resultant del següent plantejament: després de determinar el primer element, es dóna el procés de selecció entre la resta, que té $O(n - 1)$. A continuació, es torna a fer el procés pel següent element, amb $O(n - 2)$. Aquesta successió, doncs, comporta un cost $O(n^2)$. Tot i així, en promig, l’esquema de selecció presenta una reducció de temps respecte a un recorregut lineal ja que no en tots els casos resulta necessari examinar tots els candidats possibles.

El nombre màxim d’iteracions que pot realitzar l’algorisme és igual al nombre de productes, és a dir, $O(n)$. Per tant, el cost total de l’algorisme és $O(n * n^2)$.

3. Comportament

Aquest algorisme tendeix a trobar solucions on els primers productes del vector tenen molta similitud entre ells i no incompleixen restriccions, al contrari del que succeeix amb els últims. D’aquest mode, donada una prestatgeria on els productes s’endrecen d’amunt cap avall, aquells que es troben més a dalt tindran més similitud entre ells que aquells que es troben als prestatges inferiors. D’aquest mode, resulta convenient la selecció d’un producte inicial, ja que aquest serà el que quedi més amunt i, per tant, serà el més privilegiat en aquesta disposició. Augmentar el nombre d’iteracions revela si seleccionant altres productes inicials es pot trobar una millor configuració en general.