Distribució de productes a un supermercat

Algorismes i estructures de dades

Identificador de l'equip: subgrup-prop41.1
Antonia Laura Apolzan: antonia.laura.apolzan@estudiantat.upc.edu
Hajweria Hussain Shaheen: hajweria.hussain@estudiantat.upc.edu
Ariadna Mantilla Puma: ariadna.mantilla@estudiantat.upc.edu
Guillem Sturlese Ruiz: guillem.sturlese@estudiantat.upc.edu

Índex

1. Estructura de dades i algorismes utilitzats

1.1 Brute Force Algorithm

L'objectiu principal de la classe BruteForce es determinar la millor disposició d'un conjunt de productes basant-se en una matriu de similituds. Per aconseguir això, es genera cada permutació possible dels productes, seleccionant la que maximitza la similitud total. Aquesta implementació es basa en un algorisme de força bruta, adequat per a conjunts petits de dades, tot i les seves limitacions de rendiment en conjunts grans.

-double[][] matSimilituds: Conté les similituds entre parelles de productes. Cada valor *matSimilituds[i][j]* representa la similitud entre el producte ***i*** i el producte ***j***.

-Producte[] vecProductes: Conté els productes que volem disposar de manera òptima a la prestatgeria. Els índexs del vector vecProductes estan directament relacionats amb els índexs de la matriu de similituds. És a dir, l'element *vecProductes[i]* correspon al producte que ocupa la fila ***i*** i la columna ***i*** a la matriu de similituds.
Aquesta relació és essencial per calcular la similitud entre productes en una permutació donada, ja que permet utilitzar els índexs d'una permutació per accedir als valors de similitud a la matriu de forma directa i eficient.

Inicialment, vam considerar treballar directament amb el vector de productes (vecProductes) durant la generació de les permutacions. La idea era generar permutacions del vector complet d'objectes Producte i, per a cada permutació, fer una cerca (find) per determinar l'índex de cada producte dins del vector original vecProductes. Però fer cerques (find) repetides per trobar l'índex de cada producte durant cada permutació és ineficient, especialment quan el nombre de productes creix. A més, aquest enfocament feia que l'algorisme fos més complicat del necessari.
Per solucionar això, generem les permutacions no sobre els productes directament, sinó sobre un vector d'índexs que representa la seva posició en *vecProductes*. Els índexs es fan servir per accedir directament a *matSimilituds* per calcular la similitud total.
Només al final del procés, quan ja s'ha determinat quina permutació és la millor, s'utilitza el vector original *vecProductes* per reconstruir la solució final amb els objectes Producte.

-Producte[] vecResultat: Conté la disposició òptima dels productes, obtinguda després d'explorar totes les permutacions.

-double millorSimilitud: Emmagatzema el valor de la millor similitud trobada fins al moment. Inicialment, està definit com -1.0 per identificar que encara no s'ha trobat cap resultat vàlid. Al final contindrà la similitud total de la solució òptima.

-Algorisme: El pas clau de l'algorisme és generar totes les possibles disposicions dels productes. Això es fa mitjançant una tècnica recursiva que intercanvia elements dins del vector d'índexs. Per a cada permutació generada, es calcula la similitud total de la disposició i es guarda en *millorSimilitud* i en *vecResultat* si és la millor disposició que hem trobat fins al moment.

Sabem que l'estanteria és cíclica, és a dir, que el primer producte està connectat amb l'últim. Aquesta propietat implica que la disposició final no només depèn de l'ordre dels productes consecutius, sinó també de com tanquen el cicle entre el primer i l'últim producte. Gràcies a aquesta propietat cíclica, podem simplificar el problema sense perdre cap combinació rellevant: podem fixar sempre el primer producte en una posició específica, reduint així el nombre total de permutacions que cal generar.
Si no tinguéssim aquesta propietat cíclica, hauríem d'explorar totes les permutacions possibles dels n productes, cosa que generaria un total de n! permutacions, però ara només hem de fer (n - 1)! permutacions.
Inicialment, per calcular el factorial, es feia servir un tipus de dada int, ja que semblava suficient per a la majoria dels casos previstos. Durant les primeres proves de l'algorisme de força bruta, el rendiment semblava correcte en problemes petits. Quan vam començar a comparar el seu funcionament amb l'algorisme de 2-aproximació per a problemes lleugerament més grans, ens vam adonar que el programa deixava de funcionar o resolia el problema de manera anormalment ràpida. En revisar detalladament el codi i els resultats, vam descobrir que el problema no estava en l'algorisme en si, sinó en les limitacions del tipus int. Com que el factorial creix molt ràpidament, els valors excedien la capacitat d'un int, provocant desbordaments i fent que els càlculs fossin incorrectes.
Per solucionar-ho, vam decidir substituir l'ús d'int pel tipus BigInteger, que permet treballar amb nombres enters molt més grans. No obstant això, cal tenir en compte que l'algorisme de força bruta no està dissenyat per a problemes amb un nombre molt gran d'elements.

1.2 Dos Aproximació

L'algoritme Dos Aproximació intenta aconseguir un resultat similar a l'algoritme anterior amb un temps molt més reduït, de tal manera que la suma de similituds resultant sigui com a molt dues vegades pitjor. La idea principal serà construir un MST per tal de trobar les arestes amb més similitud, i d'allà construïr la nostra estanteria resultant.

L'algoritme comença creant una llista d'arestes, el conjunt de les quals formen un Maximum Spanning Tree. Per això, es crida a la funció MST. Aquesta funció comença ordenant el conjunt de totes les arestes amb un MergeSort en ordre descendent (ens interessen les arestes més grans, és a dir, amb més similitud). Un cop obtingudes les arestes ordenades, fem ús de l'algoritme de Kruskal per generar aquest MST. Aquest algoritme consisteix en anar penjant arbres MST fins que et queda un únic MST. Inicialment, hi ha n MSTs, on cada node del graf és el seu propi arbre. Aquest algoritme fa ús de la funció Find i la funció Uneix:

- Find: Retorna l'arrel de l'arbre al qual pertany el node.
- Uneix: Penja l'arrel de l'arbre amb més nodes a l'arrel de l'arbre menys nodes.

Mentres totes les arestes no formin part del mateix arbre, es va iterant per totes les arestes en ordre descendent. Si els dos vertex d'una aresta pertanyen a arbres diferents (es mira amb la funció Find), llavors s'uneixen amb la funció Uneix, i s'afegeix l'aresta a la llista resultant. Finalment, l'algoritme retorna la llista d'arestes que formen aquest arbre.

Tot i que ens encantaria tenir una estanteria de disseny amb calaixeres repartides arbitràriament, la nostra estanteria és una fila de calaixeres circular, i per tant, hem d'interpretar aquest arbre com a un vector, on l'últim element es comunica amb el primer. Aquí comença la segona part de l'algoritme: hem de construir un cicle eulerià (és a dir, un

cicle que passi per totes les arestes). Per poder crear aquest cicle, primer hem de crear un graf dirigit de l'arbre resultant anterior. Per això, es crea una llista de llistes d'enters, on cada posició de la primera llista representa la aresta, i el contingut (la segona llista) són els vertex amb els que es comunica. Finalment, es crida a la funció findEuleria, la qual s'encarrega de trobar aquest cicle.
Aquesta funció va guardant en una pila els vèrtex del graf i, per cada vèrtex, es van afegint els seus veïns a la pila i s'eliminen les connexions corresponents del graf. Quan un vèrtex no té més connexions, s'afegeix al cicle i es retira de la pila. Aquest procés es repeteix fins que s'han recorregut totes les arestes.
Un cop obtingut el cicle eulerià, ja comptem amb una disposició adient de la prestatgeria, però hi ha un problema… Hi poden haver nodes repetits! Inicialment, no ens vam preocupar massa, i eliminavem els nodes repetits començant desde la primera posició. Més endavant, testejant jocs de prova, ens vam adonar que en alguns casos, la solució era molt ineficient:

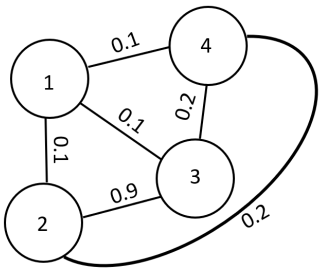


Figura 1

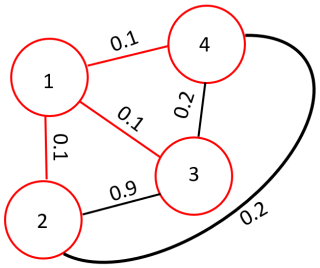


Figura 2

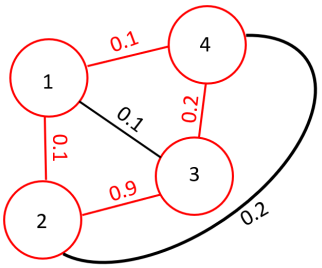


Figura 3

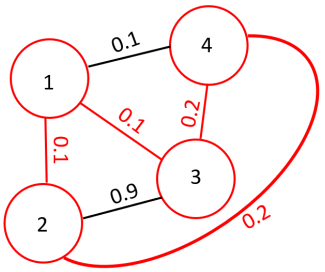


Figura 4

En aquest exemple, on es busca la suma mínima, la *Figura 1* mostra un graf de similituds, mentres que la *Figura 2* remarca en vermell l'MST d'aquest graf. Després d'executar el nostre algoritme, el cicle eulerià ens retornaría un vector que seria: [1, 2, 1, 3, 1, 4]. Eliminant els elements repetits, el vector resultant seria [1, 2, 3, 4], tal com es mostra a la *Figura 3*, amb similitud total 1.3. Però aquesta solució és molt més ineficient que la solució òptima, la de la *Figura 4*, amb similitud 0.6 i vector resultant [2, 1, 3, 4].

És per això, que vam canviar aquesta última part. La idea és generar una disposició tal que maximitza la suma de similituds. Una manera de resoldre el problema seria anar calculant totes les sumes començant desde cada element del cicle Eulerià, tot i que encara es poden optimitzar més. El que fa el nostre algoritme és, per a cada element del cicle, calcular quants cops hi apareix. Després, per a cada element que hi apareix més d'una vegada, es mira el vector resultant eliminant els repetits. Això es realitza amb la funció calculaSuma, que recorre el cicle i calcula la similitud acumulada per cada possible ordre dels productes. Si es troba una configuració millor que l'actual, aquesta es guarda com la millor solució. Finalment, retorna el millor vector trobat, que es va guardant a la variable res_productes.
La funció calculaSuma també es crida abans de començar a calcular totes les sumes, ja que pot ser que no hi hagin elements repetits, i per tant el vector que s'estaria retornant seria Null.

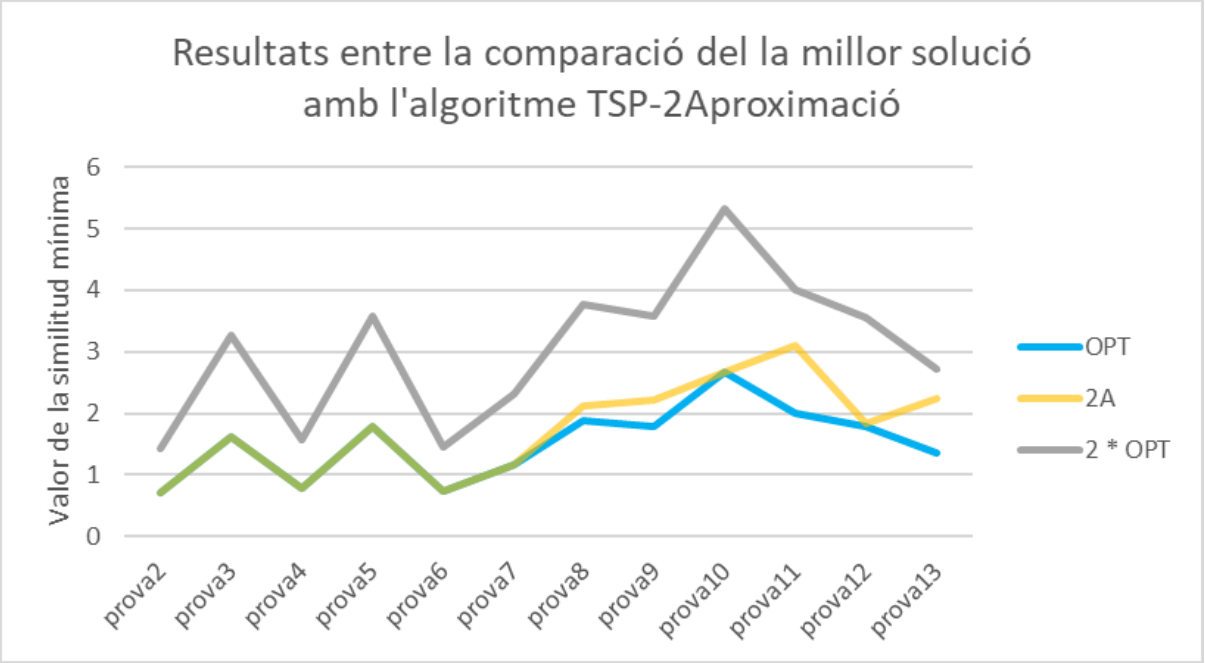
La funció pública getMillorSimilitud retorna la similitud del vector resultat, mentres que la funció pública getResultat retorna el vector de Productes resultat.

1.3 Comparació dels Algoritmes

Per veure les diferències entre els dos algoritmes, hem executat un programa que compara els dos algoritmes, imprimint-ne el cost i el temps que ha trigat cadascun. La següent taula mostra, per a diversos jocs de prova, els resultats. Els jocs de prova mostrats van augmentant de mida (prova2 té 2 productes, prova3 en té 3, etc). Per fer aquesta prova i poder assegurar que la solució 2-aproximada quedava acotada per 2 cops la solució òptima (calculada amb l'algoritme de força bruta), s'han canviat els criteris dels algoritmes per buscar la solució mínima. Aquest canvi no està present als algoritmes finals, ja que la solució que busca el nostre problema és la màxima.

provaX	**OPT**	**2 A**	**2 OPT**	**temps BF**	**temps 2A**
prova2	0,72	0,72	1,44	18,92	21,91
prova3	1,63	1,63	3,26	27,19	29,87
prova4	0,79	0,79	1,58	20,02	19,15
prova5	1,79	1,79	3,58	29	14,91
prova6	0,73	0,73	1,46	28,32	30,05
prova7	1,16	1,16	2,32	30,33	28,7
prova8	1,88	2,11	3,76	36,67	45,71
prova9	1,79	2,21	3,58	19	44,69
prova10	2,66	2,66	5,32	123,57	36,64
prova11	2,01	3,1	4,02	250,28	41,58
prova12	1,78	1,84	3,56	2090,43	48,58
prova13	1,36	2,24	2,72	22125,78	67,93

Observant les 3 primeres columnes, obtenim la següent gràfica:



Amb aquest experiment es veu clarament que l'algoritme BF té un temps d'execució que creix exponencialment amb el nombre de productes. Això es fa especialment evident a partir de *Prova10*, on el temps de BF arriba a més de 120 mil·lisegons i continua incrementant-se dràsticament amb la mida dels jocs de prova. De fet, ens va ser impossible provar jocs amb mida més gran que 13, ja que el cost computacional era massa elevat. En canvi, l'algoritme 2 Aproximació (2A) manté un temps d'execució molt més estable i eficient, fins i tot per a jocs de prova més grans. Aquesta diferència de rendiment fa que 2A sigui l'opció preferida quan es treballa amb conjunts de dades de gran mida. L'algoritme BF es torna impracticable per a jocs de prova amb més de 12 productes, mentre que 2A pot continuar processant eficientment la informació, fins i tot amb proves que tenen centenars de productes.

En termes de la qualitat dels resultats, ambdós algoritmes produeixen resultats similars per a la majoria de jocs de prova. Això indica que 2A és capaç d'aproximar-se molt bé a la solució òptima trobada per BF, especialment en casos on el nombre de productes no és extremadament gran. Com es veu a la gràfica de l'experiment, entre prova 2 i prova 7 les línies que marquen la solució se sobreposen perquè donen el mateix resultat. A partir d'aquest valor, la línia groga, corresponent al resultat del 2 Aproximació, augmenta respecte a la línia blava, corresponent a la solució òptima. Tot i això, la solució del 2A sempre queda acotada per sota de 2 vegades la solució òptima (línia grisa), confirmant així la correctesa d'aquest algoritme de manera empírica.

Aquest petit experiment demostra que, tot i que l'algoritme BruteForce ofereix solucions òptimes, el seu alt cost computacional el fa inviable per a conjunts de dades grans. DosAproximacio, en canvi, és significativament més ràpid i, a la pràctica, genera resultats amb una qualitat molt propera a l'òptima. Per tant, DosAproximacio es presenta com l'algoritme més adequat per a la majoria d'aplicacions pràctiques, especialment quan es tracta de grans volums de dades.

2. Estructures de dades de les classes

- **Classe Producte:**

Aquesta s'encarrega de representar un producte dins del sistema, amb atributs de tipus simples com Integer o String que defineixen les característiques bàsiques de l'objecte. A més, presenta una estructura de dades de tipus Map<Integer, Double> on emmagatzamem les similituds que té amb els altres productes. La clau representa l'identificador de l'altre producte i el valor Double és un valor numèric entre 0 i 1 (veure restricció de valors al document de consideracions del sistema).

En lloc de crear una classe separada Similitud, hem optat per aquesta opció perquè aquesta relació quedi directament connectada a cada producte. L'ús d'un Map<Integer, Double> per a la similitud fa que l'estructura de la classe Producte sigui més intuïtiva i més flexible de cara a futures entregues. Una estructura d'aquest tipus fa més ràpida i eficient el poder accedir a la similitud entre dos productes, ja que només cal consultar el *map* del producte en qüestió, així com el poder realitzar operacions com afegir, eliminar o modificar de manera més dinàmica sense necessitat de canviar el disseny de la classe.

Aquest disseny ens permet una gestió centralitzada de la informació rellevant del producte, ja que concretem el concepte de les connexions entre productes i simplifiquem tant l'accès com la manipulació de les dades en el sistema.

- **Classe CjtProductes:**

El disseny de la classe permet gestionar un conjunt de productes associats a un usuari concret. Aquesta estructura permet organitzar, accedir i controlar eficientment els productes, a l'igual que la classe Producte, a través d'un mapa clau-valor.

Com a atributs principals, fem servir un String per representar el nom d'usuari al qual pertany aquest conjunt de productes i que actua com a identificador. Això ens garanteix que les operacions i dades del conjunt de productes estiguin restringides a l'usuari associat en aquell moment. D'altra banda, fem ús d'una estructura de dades de tipus Map<Integer, Producte> on la clau representa l'identificador de cada producte i el valor és l'objecte que conté tota la informació del producte (id, nom, posició i similituds).

Com hem mencionat anteriorment, aquest tipus d'estructura ens proporciona un accés ràpid a qualsevol producte a través del seu *id*, amb una complexitat de cerca constant (O(1)), és a dir, amb gran eficiència i una gestió ordenada dels productes, on es pot accedir directament amb l'*id* i podem realitzar operacions d'edició sense afectar altres elements. Degut a que es tracta d'un conjunt dinàmic (el nombre de productes pot variar al llarg del temps segons les necessitats de l'usuari), hem decidit que un *map* és també la millor opció i el fet que associem aquest conjunt a un usuari ens assegura que estigui totalment limitat a un usuari concret, atorgant una seguretat i coherència a les dades.

- **Classe Usuari:**

La funció principal és la de representar a un usuari del sistema, simplificant la informació necessària per tal d'identificar-lo i autenticar-lo. La seva estructura és bastant senzilla però suficient per a garantir un principi de seguretat i personalització dins del nostre projecte.

Es basa principalment en l'ús d'un String que guarda el nom d'usuari com a identificador únic que representa l'usuari al sistema i que ens permet distinguir aquest usuari d'altres i s'utilitza com a referència en diverses operacions, i d'una contrasenya que afavoreix la seva autenticació, ja que s'utilitza per verificar que l'usuari que intenta accedir és el propietari real del compte. Això millora en termes de seguretat les dades associades. Tot i que, som conscients que una implementació real en un sistema complex hauria de protegir aquest atribut amb altres tècniques que no sigui un simple String.

De cara a futures entregues o possibles millores, aquesta estructura es pot ampliar per adaptar-se a necessitats pròximes.

- **Classe CjtUsuaris:**

El seu paper principal és el de contenidor o gestor de tots els usuaris del sistema i ens permet realitzar operacions eficients per crear, modificar i eliminar usuaris (igual que la classe CjtProductes). Aquesta ens proporciona l'estructura necessària per organitzar la comunitat d'usuaris del projecte. Mitjançant un identificador únic en format String, distingim els diferents conjunts d'usuaris, si hi ha més d'un conjunt en el sistema.

Al igual que l'anterior classe conjunt, aquest fa servir una estructura mapa on s'emmagatzema els objectes Usuari, associant-los al seu nom d'usuari com a clau, ja que ens facilita les consultes que volem fer, l'actualització d'informació i un ús eficient en cas de tenir un gran volum d'usuaris, mantenint un rendiment òptim.

- **Classe Prestatgeria:**

Per a la classe hem optat per estructurar-la de forma que tingui un esquema el més senzill possible, ja que es tracta de la representació de la disposició dels productes en el nostre supermercat. Per tant, hem decidit quines característiques volem associar a un objecte Prestatgeria, com són un array (layout) de productes que emmagatzema els objectes Producte de manera ordenada segons la seva posició en la prestatgeria.

Com ja hem explicat en el document de consideracions del sistema, la posició dins l'array en aquesta entrega representa la columna del producte. Amb aquesta estructura podem accedir als productes directament mitjançant la seva posició amb un cost constant, la qual cosa ens facilita realitzar operacions d'intercanvi entre productes, o de modificació i visualització en el cas que l'usuari vulgui veure la distribució de la prestatgeria.

També presenta un identificador únic, donant la possibilitat de gestionar diversos layouts o disposicions de productes (i poder alternar entre ells) i un valor numèric que representa el nombre de productes que conté la prestatgeria i controla l'espai disponible, evitant qualsevol error per excés de mida.

- **Classe CtrlDomini:**

Com a controlador principal, aquest gestiona la lògica del domini en el nostre sistema i unifica les operacions que involucren la resta de les classes, com són els usuaris, productes i prestatgeries, assegurant una gestió coordinada de les dades. Per aquest motiu, fem ús de diverses estructures de dades que ens faciliten aquest treball.

D'una banda, tenim un atribut de tipus Usuari que representa l'usuari que ha iniciat sessió al sistema en aquell moment i que ens permet delimitar les operacions de les classes segons l'usuari actiu. De cara a futures entregues, desenvoluparem l'administració de sessions. De manera que, per mantenir un ordre i seguretat de la independència de les dades, hem disposat d'un conjunt d'usuaris, que estableix tots els usuaris registrats al sistema i que ens garanteix que cadascú tingui només tingui accés al seu conjunt associat de productes i d'un conjunt de productes que engloba tots els productes associats a l'usuari actual, el qual es podrà recuperar o crear quan s'inicia sessió.

Altrament, entra en joc una prestatgeria que representa la distribució actual dels productes de l'usuari en el format d'un objecte Prestatgeria i on podrem realitzar les operacions pertinents de la classe. També hem establert una matriu de valors *double* (mateix tipus que les similituds) que guarden les similituds entre els productes de l'usuari i la fem servir ja que és una estructura clau per generar les solucions òptimes dins la prestatgeria segons quin algoritme decideix l'usuari que vol emplear i un vector de Productes amb tots els productes actuals de l'usuari i que serveix com a base per construir o modificar la prestatgeria en funció dels criteris que tingui.

Ens hem decidit per aquestes estructures principalment per guanyar eficiència en les operacions d'accés i càlculs relacionats amb les classes involucrades.