UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Projectes de Programació

Distribució de productes a un supermercat

Estructures de dades i Algorismes

Dídac Dalmases Valcárcel (didac.dalmases)

Rubén Palà Vacas (ruben.pala)

Pau Martí Biosca (pau.marti.biosca)

Eric Medina León (eric.medina)

Grup 12.2

Versió 1.0

Salvador Medina

2024/2025 Q1

Índex

1 Estructures de dades

El programa és un gestor de productes dissenyat per cadenes de supermercats. Es poden definir productes, botigues, i prestatges dintre una botiga. L'objectiu del programa és ajudar a trobar una bona solució d'una distribució de productes a un prestatge (o la botiga sencera), tenint en compte que els productes estan relacionats entre ells (comprar un producte fa més probable que es compri un altre producte).

Una cadena de supermercats té un catàleg de tipus de productes i botigues. Cada botiga té prestatges (circulars) i ofereix certs productes del catàleg.

1.1 Definició de productes i botigues

1.1.1 Product

Un (tipus de) producte (*Product*) està definit pel seu codi de barres (int positiu, únic), i també té un nom, un preu (float positiu) i el tipus de prestatge on pot anar (string, que ha de coincidir amb el tipus d'algun prestatge definit, si es vol col·locar a una botiga). Un exemple de tipus de prestatge seria "Nevera", o "Fruita", o el que l'usuari vulgui definir. En els nostres testos, fem servir el tipus de prestatge "Normal"per prestatges i productes que no tenen restriccions.

1.1.2 Shelf

Un prestatge ($Shelf$) té un identificador (int únic), una mida (nombre d'espais per col·locar producte), un tipus (ja mencionat al paràgraf anterior) i una llista (indexada) de productes. Els productes es guarden en forma de codi de barres (o null si la posició està buida), per no tenir dependència de la classe *Product* des de la classe $Shelf$. Cal recordar que tots els prestatges són circulars, per tant el fet que sigui una llista només ens importa per l'ordre entre productes, però no la posició en si, ja que un desplaçament de totes les posicions acaba donant el mateix resultat (considerant que l'última posició està connectada amb la primera també).

1.1.3 Store

Una botiga (*Store*) es defineix pel seu nom (string únic). Conté prestatges, que s'emmagatzemen en un map ($HashMap$) de id a instància de $Shelf$ per facilitar accés ràpid al prestatge desitjat mitjançant el seu identificador.
Cada botiga del programa ofereix uns productes determinats (que normalment no són tots els productes que s'han definit). Òbviament, tots els productes que ofereix una botiga (que es guarda com a $HashSet$ d'enters, que corresponen als codis de barres, ja que l'ordre no importa i no n'hi ha de repetits, i volem fer consultes ràpides sobre codis de barres en concret) han d'estar definits. També té guardat un conjunt de productes "col·locables". Hem interpretat que un producte només pot col·locar-se en un sol lloc d'una botiga. Per tant, aquest conjunt de productes col·locables són els productes oferts excloent els productes que estan col·locats a algun prestatge. Per mantenir aquesta llista vàlida, s'ha d'actualitzar cada vegada que hi ha un canvi de posició de producte a un prestatge o un canvi en productes oferts.

1.2 Relacions

Ens interessa guardar relacions entre productes per poder calcular la distribució òptima d'uns productes a un prestatge. Un producte està relacionat amb un altre si al comprar el primer producte és més probable que es compri el segon producte.

Com que es poden crear i eliminar tipus de productes dinàmicament, no ens serveix guardar les relacions en una matriu, ja que l'hauríem de recrear sencera cada cop que hi hagués un canvi d'aquest tipus.
És per això que les relacions es guarden en un $HashMap$ "2D", és a dir, un HashMap que té com a clau codis de barres (int) i com a valor un altre HashMap que té com a clau codis de barres i com a valor la relació entre els dos productes (1r codi de barres amb 2n codi de barres, la relació és de tipus float). És a dir, és un $Map < int, Map < int, float >>$. D'aquesta manera no es força simetria de relacions (algú que compra guacamole quasi segur que compra nachos, però no tant al revés). Amb aquesta estructura de dades, crear o eliminar un producte (i per tant les relacions amb la resta de productes) té cost lineal amb el nombre de productes ja definits.

Abans d'executar un algorisme per calcular una distribució de productes a un prestatge, la informació de relacions es converteix a un array 2D de relacions ($float[][]$) seguint un ordre determinat de codis de barres per indicar fila i columna per facilitar els càlculs, ja que la informació de relacions (que es converteix a distàncies pels algorismes) es fa servir en moltes ocasions sense ser modificada.

1.2.1 Tiquets

Com que determinar relacions entre productes pot ser bastant complicat sense una bona base de dades processada, el programa també dona la opció de calcular relacions a través de tiquets de la compra. Un tiquet és un fitxer de text on a cada fila hi ha un producte (codi de barres + nom) que no es guarda internament al programa.

Si es prefereix calcular relacions a partir d'un conjunt de tiquets, hi ha un "parser"que llegeix els tiquets especificats i assumeix que tots els productes d'un tiquet estan relacionats. Al final, queda una "taula"de relacions on cada parell de productes té un nombre de relació equivalent al nombre de tiquets que apareixen ambdós productes.

El cost de fer aquests càlculs és quadràtic per cada conjunt de productes d'un tiquet, perquè s'han de considerar totes les parelles de productes, i lineal amb el nombre de tiquets, ja que cadascun és independent de la resta. El parsing és lineal sobre el nombre de tiquets i productes, ja que guardem els tiquets llegits en una llista de $HashSet < Integer >$ (una entrada per tiquet), i així s'eviten productes repetits (la quantitat comprada d'un producte no afecta a relacions entre productes).

1.3 Controladors

Els controladors emmagatzemen totes les instàncies d'un tipus/classe, i serveixen per comunicar entre classes.

Tant *ProductController* com *StoreController* guarden les instàncies en un $HashMap$ de codi de barres a instància de producte i id de botiga a instància de botiga, respectivament. Això permet que la resta de classes treballin amb identificadors (codi de barres en el cas de producte) en comptes d'instàncies, i que l'accés a un producte/botiga/prestatge a partir del seu identificador sigui constant.

DomainController, la classe que es comunica amb la resta de capes, té una instància de $ProductController$, una de *StoreController* i una de *Relations*. D'aquesta manera, es pot aïllar la lògica de la capa de domini en tres parts.

1.4 Algorisme

Existeixen diversos algorismes implementats per poder fer el càlcul de distribucions de productes a un prestatge. Aquests algorismes són implementacions de la classe $AbstractAlgorithm$, que guarda una matriu ($float[][]$) de distàncies (no confondre amb relacions, tenen valors ïnvertits"). Com que els algorismes són solucions del TSP, aquesta matriu de distàncies és la manera més eficient de guardar el "grafçomplet generat a partir de les relacions entre productes (assumint que dos productes no relacionats tenen distància infinita).

La informació sobre estructures de dades d'implementacions d'aquesta classe abstracta d'algorisme es troba a l'apartat d'algorismes.

2 Algorismes

En el nostre programa, tractem els algorismes de calcular una distribució bona de productes a un prestatge com una caixa negra. Diferents algorismes són implementacions de la classe *AbstractAlgorithm*, que fa que la manera d'interactuar amb qualsevol algorisme sigui la mateixa.

La classe *AbstractAlgorithm* només reb una matriu de distàncies (immutable, i per tant es passa i es guarda com a $float[][]$) com a paràmetre durant inicialització, i els algoritmes implementats intenten minimitzar el cost d'un cicle que passi per tots els vèrtexs (hamiltonian path).

Les funcions importants, utilitzades per interactuar amb l'algorisme, són:

- $canUseAlgorithm()$, que analitza la matriu de distàncies i determina si un algorisme en concret és apte per trobar una solució. *AbstractAlgorithm* comprova que la matriu de distàncies sigui quadrada i que no tingui valors negatius (per tant cost $O(n^2)$, ja que ha de comprovar cada valor), i les implementacions fan comprovacions addicionals (després de cridar $super()$).
- $calculateCycleCost()$, que reb un array de vèrtexs i calcula el cost de recórrer aquest cicle (incloent últim vèrtex amb el primer). No depèn de l'algorisme implementat, i té cost $O(m)$, on m és el nombre de vèrtexs que forma el cicle.
- $calculateDistribution()$, que executa l'algorisme en si (sobre la matriu de distàncies) i retorna el resultat trobat.

Per tant, per cada algorisme implementat, especificarem quins càlculs són necessaris fer durant la inicialització (si n'hi ha), a la funció $canUseAlgorithm()$ i a la funció $calculateDistribution()$.

Alguns algorismes accepten paràmetres, que canvien com fan algun pas de l'algorisme. Quan sigui el cas, s'especificarà quin impacte té cada paràmetre en l'execució, i les diferències de cost que pot comportar.

2.1 Greedy

La implementació greedy troba una solució molt ràpidament i poc optimitzada. Com el seu nom indica, comença en un vèrtex en concret, i a cada pas mira quin vèrtex que encara no ha visitat té la menor distància des de l'actual.

La funció $canUseAlgorithm()$ no fa comprovacions addicionals, i tampoc hi ha càlculs addicionals durant la inicialització.

Existeix un paràmetre booleà anomenat "$TestAllStartingNodes$" que, si és $true$, executarà l'algorisme n vegades (n sent el nombre de vèrtexs = mida de la matriu de distàncies), una considerant un vèrtex diferent com a inicial (que en alguns casos pot ajudar a trobar una solució millor). Si és $false$, només s'executarà un cop, amb el vèrtex 0 com a inicial.

La funció *calculateDistribution*(), per tant, té cost $O(n^2)$ si només s'executa un cop amb el 0 com a node inicial, i té cost $O(n^3)$ si s'executa n vegades. L'explicació és que a cada iteració, s'ha de trobar el vèrtex no visitat amb distància mínima a l'actual ($O(n)$), i tenim n iteracions.

2.1.1 Pseudocodi Greedy

Algorithm 1 Greedy Algorithm for TSP

Require: *distanceMatrix* (matrix of distances between nodes)
Require: *testAllStartingNodes* (boolean)
Ensure: *bestOrder* (sequence of nodes minimizing cost)

```
1:
2: function GreedyTSP
3:     if distanceMatrix is empty then
4:         return []
5:     end if
6:     if not testAllStartingNodes then
7:         return FindGreedyPath(0)
8:     else
9:         bestOrder ← FindGreedyPath(0)
10:        bestCost ← CalculatePathCost(bestOrder)
11:        for i ← 1 to n − 1 do
12:            currentOrder ← FindGreedyPath(i)
13:            currentCost ← CalculatePathCost(currentOrder)
14:            if currentCost < bestCost then
15:                bestCost ← currentCost
16:                bestOrder ← currentOrder
17:            end if
18:        end for
19:        return bestOrder
20:    end if
21: end function
22:
23: function FindGreedyPath(startNode)
24:    Mark all nodes as non-visited
25:    current ← startNode
26:    path ← [current]
27:    while there are non-visited nodes do
28:        Select the nearest non-visited node to current
29:        Mark the selected node as visited
30:        Add the selected node to path
31:        current ← selected node
32:    end while
33:    return path
34: end function
```

2.2 2-Aproximació amb Kruskal

L'algorisme de 2-Aproximació consisteix en seleccionar un nombre molt reduït d'arestes a considerar (en aquest cas, l'arbre mínim d'expansió), i treballar sobre aquestes arestes per trobar un bon cicle que intenti minimitzar la distància total.
El major problema que té és que necessita que el graf d'entrada compleixi la desigualtat triangular. Això limita bastant en quins casos es pot arribar a fer servir. Quan es pot fer servir, és un algorisme ràpid que garanteix una solució prou bona per la majoria dels casos.

2.2.1 Constructora

La constructora d'aquest algorisme guarda també el graf (definit per la matriu de distàncies) com a llista d'arestes (no dirigides) amb pes ($WeightedEdge$, que guarda node origen, node destí i pes (float)), ordenades per pes. Aquesta conversió té cost $O(n^2 \log(n))$, ja que crear la llista té cost $O(n^2)$ perquè té $n(n-1)/2$ elements (cada node amb la resta, la meitat perquè és graf no dirigit), i ordenar la llista per tant té cost $O(m \log(m))$, on $m = n(n-1)/2$, és a dir, $O(n^2 \log(n))$ (ja que $\log(n^2) = 2\log(n)$).

2.2.2 canUseAlgorithm

La funció $canUseAlgorithm()$ ha de comprovar, a part de les comprovacions ja fetes per $AbstractAlgorithm$, que es compleixi desigualtat triangular a tot el graf, que és necessari per garantir que el procediment d'aquest algorisme trobi una solució com a molt 2 vegades pitjor que l'òptima. Com que per cada parell de vèrtexs (i, j), hi ha $n-2$ vèrtexs que es poden posar al mig: (i, k, j), s'han de fer $n^2(n-2)$ comprovacions de tipus $d[i][j] <= d[i][k] + d[k][j]$ (on d és la matriu de distàncies), per tant la funció tindrà cost $O(n^3)$.

Com a nota addicional, com que per construir el MST es fan servir arestes no dirigides, en aquest pas s'ha d'assumir simetria. En cas que la matriu donada no sigui simètrica, la funció $canUseAlgorithm()$ imprimirà un "warning"avisant del problema, però l'algorisme es pot executar igualment ignorant una meitat de la matriu de distàncies. Aquesta comprovació té cost $O(n^2)$, per tant no afecta a la complexitat de la funció.

2.2.3 calculateDistribution

L'execució de l'algorisme es pot dividir en X passos.

Inicialment, es troba el $MinimumSpanningTree$ del graf (que té $n(n-1)/2$ vèrtexs inicialment, perquè és no dirigit). Per fer-ho, es fa servir $DisjointSetUnion$ (també conegut com a $MergeFindSet$):

El DSU s'utilitza per trobar el MST agrupant els nodes d'un graf sense crear cicles, mantenint la connexió mínima possible. Al principi, cada node és el seu propi "pareï no existeixen arestes. Es van afegint arestes al graf, però abans de fer-ho, es comprova si els dos nodes a connectar estan en el mateix grup. Si ja ho estan, no s'afegeix la connexió, perquè crearia un cicle.

Quan es troben dos nodes que no estan connectats, s'uneixen en un sol grup (es crea una aresta entre els dos nodes). Això es fa actualitzant el "pare"d'un dels dos grups de nodes al "pare"de l'altre grup de nodes. Com que la crida de trobar el "pare"és recursiva, s'aprofita per actualitzar el "pare"de cada node a l'arrel de l'arbre actual dels nodes que formen el grup (és a dir, s'intenta que tots els nodes d'un grup tinguin el mateix pare). Així, cada vegada que es vol connectar, es pot trobar el "pare"del grup (el mateix per tots els nodes del grup) amb cost constant. Es van afegint connexions fins que tots els nodes estan connectats de la manera més eficient possible, formant el MST.
Hem fet servir la referència de cpalgorithms, que té una explicació de l'algorisme de Kruskal DSU i una implementació en $C++$, per implementar l'algorisme en $Java$ dintre la nostra classe.
Crear el MST té cost $O(m+n)$ (on n és el nombre de vèrtexs, i m és el nombre d'arestes, que és $n(n-1)/2$ en el nostre cas). m perquè s'itera sobre totes les arestes, i n perquè per cada aresta que els dos vèrtexs pertanyen a un diferent grup, s'uneixen els grups (constant perquè es va actualitzant el "pare"dels nodes), i això passa exactament $n-1$ vegades (nombre d'arestes t'un arbre).

A continuació, es converteix el MST en un graf dirigit duplicant les arestes (una aresta no dirigida passa a ser 2 arestes dirigides, una a cada sentit). Aquest graf es guarda en una matriu d'adjacència ($boolean[][]$).

Amb aquest nou graf, intentem trobar un cicle eulerià (camí que passa per totes les arestes una vegada). També ens hem basat en l'explicació i codi de cpalgorithms, simplificant algunes parts ja que la nostra matriu d'adjacència és booleana i el graf té exactament $2(n-1)$ arestes.
L'algorisme utilitza una pila per fer un seguiment del camí mentre explora el graf. Comença posant un node qualsevol a la pila. Mentre hi hagi nodes a la pila, s'agafa l'últim afegit i es mira si hi ha cap connexió disponible. Si n'hi ha, s'elimina l'aresta i l'algorisme es mou al node connectat, posant-lo a la pila. Això permet continuar seguint un camí.
Quan arriba a un node sense més connexions disponibles, el treu de la pila i l'afegeix al camí eulerià resultat. L'algorisme para quan s'han visitat totes les arestes. Al final, el resultat mostra el camí que segueix totes les arestes exactament una vegada, és a dir, un camí eulerià.
Aquest algorisme té cost $O(m)$, on m és el nombre d'arestes, perquè es fa una iteració per cada aresta. Per tant, $O(n)$ en el nostre cas, ja que tenim $2(n-1)$ arestes.

Finalment, cal convertir el cicle eulerià obtingut en un cicle hamiltonià (camí que passa per tots els vèrtexs un cop, que és el que volem). L'algorisme dona tres opcions (paràmetre) sobre com fer aquest últim càlcul, cadascun amb el seu cost i qualitat de solució:

- **Node inicial fixat**: Procés simple que comença agafant el node 0, recorre el cicle eulerià i quan troba un vèrtex nou el col·loca al cicle hamiltonià. Té cost $O(n)$, ja que només ha de recórrer el cicle eulerià un cop.
- **Millor node inicial**: Repeteix el procés anterior n vegades, una per cada vèrtex com a inicial. Es queda el millor resultat, que correspon al cicle tal que la distància de recorre'l és menor. Com que es fa el procés anterior n vegades, té cost $O(n^2)$, però garanteix una solució igual o millor que el primer cas.

- **Millor node inicial optimitzat**: Abans de trobar un cicle hamiltonià, elimina arestes consecutives del cicle eulerià. És a dir, si en algun lloc del camí hi ha vèrtexs i, j, i, j, k, aquesta seqüència es convertirà en i, j, k. D'aquesta manera, s'eliminen comprovacions innecessàries abans d'executar el procés per cada node inicial. Com que en el pitjor dels casos no hi ha cap optimització, el cost d'aquesta opció també és $O(n^2)$ (perquè fer la "poda"del cicle eulerià és lineal, només cal recórrer el cicle un cop), però en la majoria de casos tardarà menys, donant una solució igual de bona.

Aquest cicle hamiltonià resultant és la solució del problema.
En conclusió, la complexitat d'aquest procediment per trobar un cicle és $O(n^2 \log(n))$ per culpa de la constructora, ja que es necessiten ordenar les arestes per executar el DSU per trobar el MST.

2.2.4 Pseudocodi 2-Aproximació amb Kruskal

Algorithm 2 Kruskal-based 2-Approximation for TSP

Require: *distanceMatrix*, *eliminationType*
Ensure: *bestOrder*

```
function KruskalTSP
    mst ← GetMST(distanceMatrix)
    eulerianPath ← FindEulerianPath(mst)
    if eliminationType = "FirstStartingNode" then
        return RemoveRepetitions(eulerianPath, eulerianPath[0])
    else
        bestOrder ← RemoveRepetitions(eulerianPath, 0)
        for i = 1 to n − 1 do
            currentOrder ← RemoveRepetitions(eulerianPath, i)
            if Cost(currentOrder) < Cost(bestOrder) then
                bestOrder ← currentOrder
            end if
        end for
        return bestOrder
    end if
end function

function GetMST
    Sort edges by weight
    return selected edges forming MST
end function

function FindEulerianPath
    Follow edges to form Eulerian path
    return Eulerian path
end function

function RemoveRepetitions(path, startNode)
    return Hamiltonian path by removing repeated nodes
end function
```

2.3 Simmulated Annealing

L'algoritme de *SimulatedAnnealing* és un algorisme de cerca local que s'encarrega de trobar aproximacions properes a la solució òptima en problemes d'optimització. Combina l'exploració global i local, introduint aleatorietat per escapar de mínims locals.

Aquest algorisme parteix d'una solució inicial, que es modifica iterativament mitjançant l'ús d'operadors que generen solucions veïnes en l'espai de cerca. Aquestes solucions veïnes es poden acceptar o rebutjar en funció del seu cost i d'una probabilitat d'acceptació basada en la distribució de Boltzmann. Aquesta probabilitat permet, inicialment, acceptar solucions pitjors per tal d'evitar quedar atrapats en mínims locals. A mesura que avança el procés, la temperatura del sistema disminueix (segons una funció de refredament), fent que l'algorisme esdevingui més selectiu i prioritzi la millora local de la solució actual.

Les funcions $calculateCycleCost()$ i $canUseAlgorithm()$ no requereixen cap modificació per aquest algorisme. Realitzen la funció per defecte.

La funció $generateInitialSolution()$, genera una solució aleatòria a partir de la matriu de distàncies. Crea una Array amb n índexs, on n és el nombre de productes (files en la matriu de distàncies), i aleatoritza el seu ordre realitzant un $shuffle$. Aquesta generació es bastant adient pel nostre problema ja que el seu cost es lineal ($O(n)$) i al ser aleatòria evita que l'algorisme quedi fàcilment atrapat en mínims locals.

La funció $generateRandomNeighbour()$ s'encarrega de generar una solució veïna a partir de la solució actual. Aquesta funció escull dos índexs aleatoris de la solució actual i els intercanvia, permetent explorar tot l'espai de solucions. Aquesta operació té un cost constant ($O(1)$), però com es necessita fer una còpia de la solució actual, aquesta té un cost de $O(n)$.

La funció $calculateDistribution()$ s'encarrega de l'execució de l'algorisme. Aquest disposa dels paràmetres $Temperature$, $CoolingRate$ i un factor d'escala K. Primerament genera la solució inicial i calcula el seu cost amb la funció $calculateCycleCost()$. A continuació, entra en un procés iteratiu respecte el paràmetre $Temperature$, que va decrementant multiplicant-se per $CoolingRate$. A cada iteració, es generen K solucions veïnes i es calcula el cost de cada una. Si la nova solució és millor que l'actual (cost de cicle menor), l'accepta immediatament. En cas contrari, l'accepta en el cas de que $P > random(0, 1)$, sent $P = e^{-(newCost - currnetCost)/(K * Temperature)}$. Aquesta probabilitat permet acceptar inicialment solucions pitjors, però a mesura que la temperatura disminueix, esdevé més selectiu ja que el valor de P va acostant-se cada cop més a 0.

El cost del càlcul de la distribució és de $O(|log_{CoolingRate}(Temperature)| \cdot K \cdot N)$, on $|log_{CoolingRate}(Temperature)|$ són les iteracions del bucle extern, K les iteracions del bucle intern i N el cost de les operacions realitzades dintre d'aquest.

Els valors inicials que hem triat pel paràmetres de l'algorisme són els següents:

- $Temperature = 1000$
- $CoolingRate = 0.9$
- $K = 600$

Hem fet proves i hem trobat una combinació de valors que ofereixen un bon equilibri entre temps d'execució i qualitat de la solució. El valor de $Temperature$ es suficientment gran com per poder proporcionar una exploració inicial adequada, el valor de $CoolingRate$ disminueix la temperatura de forma moderada i el valor de K l'hem reduït una mica per evitar temps d'execució excessius quan l'algoritme s'executi amb molts productes. De totes formes, es permetrà personalitzar aquests paràmetres en funció dels gustos de l'usuari.

2.3.1 Pseudocodi Simmulated Annealing

Algorithm 3 Simulated Annealing

Require: $distanceMatrix$
Ensure: $bestSolution$

```
1:
2: function SimulatedAnnealing
3:     currentSolution ← GenerateInitialSolution
4:     currentCost ← CalculateCost(currentSolution)
5:     bestSolution ← currentSolution
6:     bestCost ← currentCost
7:     temperature ← INITIAL_TEMPERATURE
8:     while temperature > 1.0 do
9:         for i = 1 to K do
10:            newSolution ← GenerateRandomNeighbor(currentSolution)
11:            newCost ← CalculateCost(newSolution)
12:            deltaCost ← newCost − currentCost
13:            if deltaCost < 0 or AcceptanceProbability(deltaCost, temperature) > Ran-
    dom(0,1) then
14:                currentSolution ← newSolution
15:                currentCost ← newCost
16:                if currentCost < bestCost then
17:                    bestSolution ← currentSolution
18:                    bestCost ← currentCost
19:                end if
20:            end if
21:        end for
22:        temperature ← temperature ∗ COOLING_RATE
23:    end while
24:    return bestSolution
25: end function
26:
27: function AcceptanceProbability(deltaCost, temperature)
28:    return 1, exp(−deltaCost/(K · temperature))
29: end function
```