ESTRUCTURES DE DADES I ALGORISMES

Professor:
Iván Paz
Assignatura:
Projecte de Programació
Subgrup:
Subgrup-prop 41.5
Autors:
Axel David González Saldivar
axel.david.gonzalez@estudiantat.upc.edu
Marcel Masip Bagán
marcel.masip@estudiantat.upc.edu
Aleix Parrilla Martín
aleix.parrilla@estudiantat.upc.edu
Versió:
Lliurament versió 1.1

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
2024

ÍNDEX

ESTRUCTURES DE DADES DE LES CLASSES

En els següents apartats explicarem quines estructures de dades i algorismes hem utilitzat per les diverses classes del projecte, i el motiu pel que hem escollit utilitzar-les envers d'altres.

➢ Prestatge

- ArrayList<String> distPrestatge
  - Avantatges:
    - És una forma simple i fàcil per emmagatzemar els noms dels productes que tindrem en el prestatge. També ajuda visualment a representar un prestatge.
    - La classe ArrayList de Java proporciona diverses operacions útils, com add(), remove(), contains()..., i permet accedir als elements de manera eficient (amb el mètode get()).
    - La redimensió d'aquesta estructura és automàtica.
  - Inconvenients:
    - Tot i que el redimensionament és automàtic, aquest procés té un cost.
    - Si es necessita fer canvis en la distribució del prestatge, l'accés seqüencial i la manipulació amb operacions com add() o remove() poden tenir cost O(n).

➢ Producte

- String nom
  - Avantatges:
    - És fàcil d'utilitzar i permet identificar un producte de forma única, amb un nom llegible.
    - Es pot comparar de manera eficient, utilitzant l' equals().
  - Inconvenients:
    - Encara que és eficient comparar l'string, seria més eficient comparar un integer, i també consumeixen més memòria que si utilitzessim identificadors numèrics.

- ArrayList<Double> similituds
  - Avantatges:
    - Permet afegir i eliminar similituds de manera fàcil, ja que la redimensió d'aquesta estructura és automàtica.

            - Es pot accedir als elements utilitzant l'índex, en temps constant O(1), cosa que ens ajuda a l'hora de modificar una similitud específica.
        - Inconvenients:
            - Tot i que aquesta estructura es pot redimensionar automàticament, aquest procés té un cost computacional.
            - Cada cop que s'afegeix un nou valor, el tipus primitiu double es converteix a l'objecte Double.

➢ AlgorismeFB

- Map<String, double[]> Shelf
    - Avantatges:
        - Emmagatzema l'input de productes i similituds
        - Eficiència a l'hora d'accedir a les dades
    - Inconvenients:
        - Major consum de memòria
- String[] defSolution
    - Avantatges:
        - Emmagatzema de manera simple la distribució final
        - Eficiència de memòria
        - Accés ràpid a elements
    - Inconvenients:
        - Mida Fixe
        - Operacions d'inserció i eliminació costosa
- ArrayList<String> LlistaProductes
    - Avantatges:
        - Mantenim l'ordre d'inserció
        - Podem accedir els elements per índex
        - Permet productes duplicats
    - Inconvenients:
        - Redimensió costosa : O(n)
        - Inserció i eliminacions costoses si són al mig de la llista

➢ DistribucioKruskal

- LinkedHashMap<String, double[]> Mapa
    - Avantatges:
        - Podem mantenir l'ordre d'entrada dels productes
        - Eficiencia a l'hora de trobar les similituds (cost promig O(1))

            - Podem accedir fàcilment als productes amb la crida .entrySet()
        - Inconvenients:
            - És lleugerament més lent que HashMap en operacions bàsiques
    - ArrayList<Aresta> Arestes
        - Avantatges:
            - Emmagatzema cada parell de productes en una aresta el pes del qual sera la similitud entre aquests
        - Inconvenients:
            - Es podria utilitzar una estructura de dades que utilitzés menys memòria com un HashMap

➢ Costos

Sent n la mida de les estructures.

	Accés	Buscar	Afegir/Eliminar
ArrayList	O(1)	O(n)	O(n)
HashMap	O(1)	O(1)	O(1)
String[]	O(1)	O(n)	No es pot redimensionar.
LinkedHashMap	O(1)	O(1)	O(1)

ALGORISMES

Algorisme FB

Aquest algorisme busca trobar la millor distribució possible d'un conjunt de productes, maximitzant la similitud entre aquells que estan en contacte. algorisme fa servir una estratègia de força bruta per examinar totes les combinacions possibles de distribució i seleccionar la més òptima.

Preparació de dades

Inicialment, rebem una entrada amb un conjunt de productes i les seves similituds. A cada producte se li assigna un vector de similituds amb els altres productes. Aquesta informació es guarda en una estructura Map<String,double[]> Shelf, una taula de mapes que relaciona cada producte amb un array de dobles, representant les seves similituds amb els altres productes.

Generació de la distribució

Es defineix una funció generarPrestatge(). En aquesta funció començarem inicialitzant les estructuras per a emmagatzemar el resultat “DefSolution” i un vector de visitats inicialitzat a fals, que ens servirà per a saber quins productes han estat visitats dins del backtracking.

Es fa servir un enfocament de backtracking, un mètode que explora totes les opcions i descarta les que no aporten la millor solució. Seria l’equivalent a crear un arbre i consultar cada node fulla.

L'algorisme calcula la similitud total de cada configuració i manté la millor solució trobada fins al moment (defSolution). Si apareix una millor, la reescriu.

Càlcul de la similitud

La funció scoreCalc() calcula la similitud d'una disposició de productes. Per a cada parell consecutiu de productes en la disposició (considerant que la disposició és circular), suma la similitud entre aquests dos. Aquesta funció permet comparar diverses disposicions i seleccionar la que maximitza la similitud.

```
for (int i = 0; i < N; ++i)

       {

           String uno = solution[i];    //Producte 1

           String dos = solution[(i+1)%N];//Producte 2

           double aux[] = Shelf.get(uno);

           score += aux[getIndex(dos)];

      }

      return score;
```

Backtracking per buscar la millor solució

El mètode backtracking() es crida recursivament per generar totes les disposicions possibles. Per cada distribució calculem la seva puntuació amb l'anterior funció. Si la solució actual és millor que la millor trobada fins ara, es guarda com a solució definitiva, mitjançant el mètode clone(). Aquest procés continua fins a haver explorat totes les combinacions possibles, és a dir, totes les fulles de l'arbre.

Cost

El cost d'aquest algorisme no varia gaire per sobre d'un altre backtracking convencional. Per a N productes, el cost total en el pitjor serà de N!, és a dir (O(N!)). Això és degut a que el algorisme explora totes les permutaciones posibles (N!) i, para cada permutació, calcula la puntuació de similitud en O(N).

Algorisme 2-Aprox

Com ja sabem, els algorismes d'aproximació són algorismes que troben solucions sub-òptimes per a problemes d'optimització. Una 2-Aproximació és una solució que és com a molt dos vegades pitjor que la solució òptima.

Així doncs, l'algorisme que plantegem parteix d'un input, un conjunt de productes. Aquests productes tenen similituds entre ells.

L'objectiu serà trobar una distribució, un prestatge, amb la màxima similitud entre productes possible. De manera que tindrem en compte les similituds dels productes que estiguin disposats en contacte amb altres. Sigui una distribució tal que p1, p2, p3 ... pn, tindrem en compte les similituds p1-p2, p2- p3... pn-p1, ja que, com hem dit, serà una distribució circular.

Inicialització Dades

A partir de l'entrada, construirem una estructura per a guardar els productes i les similituds, un mapa LinkedHashMap similituds. És important que sigui LinkedHashMap per a poder mantenir l'ordre d'entrada, ja que sinó se'ns ordenaria per ordre alfabètic i ens donaria problemes a l'hora d'accedir a la similitud de cada producte. Posteriorment crearem un ArrayList d'Arestes. Aquestes arestes estaran formades per dos vèrtexs (Producte A i Producte B) que tindran una similitud, un double, que representarà la similitud entre els dos productes.

Un cop inicialitzades les estructures, el primer pas serà trobar un arbre mínim d'expansió T, Minimum Spanning Tree (MST). Un MST és un graf connex que passa per tots els vèrtexs sense generar cap cicle, i que les seves arestes tenen el menor cost possible. En el nostre cas, ens interessa que aquestes arestes siguin les de major pes possible, per a que la similitud que trobem sigui màxima, com hem indicat en els nostres objectius. Per aquesta raó, després d'inicialitzar "Arestes", el nostre array d'arestes, el que farem serà ordenar-les de manera descendent. Així, tindrem al principi les arestes de major similitud.

Trobar MST: Algorisme KRUSKAL

L'algorisme Kruskal recorre les arestes ordenades prèviament, en el nostre cas descendentment, i va creant l'arbre d'expansió T afegint ordenadament les arestes que no formin cicles a T.

El funcionament d'aquest algorisme és interessant. Consisteix en seguir l'estructura Merge Find Set.

Inicialitzarem tots els productes en el seu propi grup, de manera que al inici de l'algorisme, cada producte estarà en un grup diferent, i per tant, cada producte serà el seu propi pare (Set) . Per a guardar aquesta informació l'estructura més eficient és un Map per a emmagatzemar el producte i el seu "pare", que com hem dit abans, al principi és ell mateix. Per tant, en el map tindrem el producte (key) que apuntarà al pare (value).

Un cop tenim tota la informació, recorrerem lel array d'arestes prèviament ordenat. En cada aresta tindrem els productes P1 i P2, dels que haurem de trobar quins son els pares. Per a trobar els pares farem servir la funció Find:

```
private static String find(Map<String, String> pare, String node)
```

Aquesta funció recursiva consisteix en trobar el pare del producte node. El cas base serà el del inici del algorisme, si pare.get(node).equals(node), és a dir, si el producte és el seu propi pare. Si tenim aquest cas, retornarem el value, osigui el pare. Per contra, seguirem buscant recursivament fins a trobar-lo, i quan això passi, modificarem els camins del map per a que el producte apunti directament al seu pare. Aquest mètode s'anomena path compression, i és una millora en eficiència per a poder fer les properes cerques més eficients.

Així doncs, amb aquesta funció trobarem el pare de cada producte de l'aresta. Si els seus pares no coincideixen, és a dir, formen part de dos subconjunts diferents, unirem els grups, de manera que un producte serà el pare de l'altre producte (Merge). És important recalcar que només unirem els grups quan siguin de grups diferents, ja que d'aquesta manera estem evitant que es formi cap cicle, mantenint la definició de MST.

En aquest cas, no ens cal aplicar cap condició per a realitzar aquesta acció, així que indiferentment, posarem el producte 2 com a pare del producte 1, i afegirem l'aresta al MST. Seguirem fins a arribar a que la mida del MST sigui igual a la LlistaProductes.size() - 1, per assegurar que no tanquem cap cicle.

Finalment haurem trobat el nostre MST.

El cost de l'algorisme de Kruskal es basa en la suma d'ordenar les arestes O(n log n) + inicialitzar cada producte (set): O(1) * O(n) + construir el MST: O(n*log(n)). Per tant tindrem un cost de O(n*log n).

Construcció d'un cicle Eulerià

Un cicle eulerià és un cicle en un graf que recorre cada aresta exactament una vegada i torna al node inicial. S'han de complir un parell de condicions per a considerar un cicle com a Eulerià:

El graf ha de ser connex (s'ha de poder arribar a qualsevol node des de qualsevol node

Tots els nodes han de tenir grau parell

```
public static List<String> generaCicleEuleria(List<Aresta> mst) {
```

Amb tal de poder complir aquesta segona condició, construirem un mapa duplicant les arestes del MST que ens ha donat com a resultat algorisme de Kruskal. Ho guardarem en un map<String, List<String>> connexions, on tindrem una matriu de les connexions de cada producte. Per assegurar-nos de que cada producte té un grau parell, hem afegit una estructura, un mapa que identifica els productes que tenen grau imparell. Posteriorment, haurem de connectar entre sí aquests productes en el mapa general, de manera que tindran grau parell.

Un cop tenim tots els productes al mapa general "connexions" ordenarem les arestes per ordre alfabètic per a seguir un ordre a l'hora d'escollir un camí (no és necessari).
Finalment, farem un recorregut en DFS del mapa de connexions mitjançant una stack<String>. En aquest cas, començarem pel primer producte del MST per a seguir un ordre, de manera que farem push a la pila d'aquest. Seguirem el recorregut del DFS fins que el producte al que visitem no tingui més veïns per a visitar. Quan això passi afegirem al cicle. Seguirem així fins a obtenir el cicle Eulerià.
Hem de tenir en compte que en aquest cicle eulerià apareixen tots els nodes, però hi poden aparèixer de repetits, punt que arreglarem en el següent pas.

Transformació a cicle hamiltonià

Un cicle hamiltonià és un cicle en un graf que recorre cada vèrtex exactament una vegada i torna al vèrtex inicial. No hi han condicions necessàries i suficients per a verificar si un graf té un cicle hamiltonià. Per això és un NP-complet.

Així doncs, per a construir el cicle hamiltonià prendrem el anterior cicle Eulerià i eliminarem els nodes repetits.
Per a millorar el resultat, hem agafat diferents punts de partida, per obtenir diferents cicles hamiltonians i quedar-nos amb la que té similitud més alta.
Així doncs, el punt de partida canviara per cada iteració.
Necessitarem una estructura per a marcar els nodes repetits, un Set<String> visitados per cada cicle Hamiltonià.
Començarem el recorregut pel primer element del cicle Eulerià, el marcarem com a visitat i afegirem al resultat final. Seguirem fent el recorregut i finalment obtindrem una possible distribució màxima.

Calcularem la puntuació de cada distribució similitudActual amb la funció ScoreCalc, de manera que ens guardarem la màxima en una variable double maxSimil, i la millor distribució en ArrayList<String> resultatfinal.

Així doncs, aquest conjunt d'algorismes seguit, conformen la 2-aproximació. Que com sabem, ofereix una solució aproximada. Una solució que com a molt, en el pitjor dels casos és fins al doble del cost del cicle hamiltonià òptim.