Documentació 1a Entrega PROP

Estructures de dades i algorismes

Quadrimestre de tardor, curs 2023/24

Versió lliurament: 1.0

Grup 31.3

Marc Mostazo

Arnau Tajahuerce

Agustí Costabella

Francisco Torredemer

Algorismes i estructures de dades

Taula de continguts

Branch and Bound:

L'algorisme Branch and bound és el mètode amb el qual resoldrem el nostre problema NP-complet. Per resoldre aquest problema, primer necessitem tindre per cada parell d'ubicacions la seva distància i per cada parell de lletres la seva freqüència. El primer que farem serà, per tant, guardar la distància de cada parell de posicions al layout que rebi l'algoritme, i les freqüències de cada parell de lletres segons la llista de paraules i freqüències que es rep com paràmetre, en ambdós casos en una matriu de doubles.

La nostra estratègia per fer el branching serà Eager. Anirem construint un arbre de solucions, que s'aniran emmagatzemant en una cua de prioritat segons la seva cota. Començarem fent un algoritme Greedy que ens donarà una solució inicial a partir de la qual farem el branching i el bounding per arribar a la solució òptima. Quan trobem una solució final, és a dir, amb un total de lletres usades igual a les lletres de l'abecedari, haurem encontrat la solució. En cada procés de branching afegirem una lletra encara no utilitzada, i tindrem una solució parcial nova cadascuna amb aquesta lletra en una posició no ocupada diferent. Guardarem a la cua la millor solució d'aquestes.

Principals estructures de dades:

- **double[ ][ ] Mat_dist**
    - La matriu de distàncies,per cada element ‘i’ ‘j’ de la matriu tenim la distància de la posició ‘i’ del layout a la posició ‘j’. Per tant la diagonal d’aquesta matriu serà tot 0. El tamany serà sent n el nombre de lletres de l'abecedari, de n per n, i llavors si el layout introduït té més posicions que lletres en l'abecedari, deixarem en blanc les posicions sobrants, només omplirem les primeres n posicions del layout, és dir, durant tota l’execució del algoritme mai no accedim a aquestes darreres posicions.

- **double[ ][ ] Mat_traf**
    - La matriu de trànsit ,per cada element ‘i’ ‘j’ de la matriu tenim el trànsit de la lletra ‘i’ de l'abecedari a la lletra ‘j’. El tamany serà sent n el nombre de lletres de l'abecedari, de n per n. Per el càlcul d’aquesta

matriu recorrem la llista de paraules i les seves freqüències. Haurem de treure tots els accents i signes de cada lletra de les paraules, perquè si no fallaria al buscar el caràcter al abecedari.

- **char[ ][ ] best_sol**
    - Quan l'algoritme tingui una solució definitiva, guardarem en aquest paràmetre el layout resultant. Serà del tamany introduït pel usuari, es dir, el nombre de files i el nombre de columnes passats per paràmetre.
- **Map<Character, Integer> letra_pos:**
    - Aquesta estructura l'utilitzarem per trobar el índex de cada lletra a les matrius de distància i trànsit. L'omplirem al principi de l'execució, posant per cada lletra en ordre de l'alfabet el seu índex, és dir, per la primera lletra de l'abecedari un 0, la segona 1, la tercera 2, i així fins la última que tindrà el n-1(n es el nombre de lletres).
- **PriorityQueue<Nodo> q = new PriorityQueue<>(new NodoComparator()):**
    - És la estructura que utilitzem per emmagatzemar las solucions que anem explorant. S'ordena segons NodoComparator: segons la cota del node, de forma que sempre surten primer les solucions amb menor cota, i últimes les que tenen major cota.

<u>Solucions parcials(Nodo) i càlcul de la cota:</u>

Cada solució que obtenim del procés de branching es un objecte de la classe Nodo. Per cada node calcularem la seva cota segons el mètode de Gilmore-Lawler. Aquest mètode estima el cost d'una solució segons tres termes. El primer terme l'obtindrem sumant la distància per la freqüència de cada parell de lletres col·locades a la

solució actual. Per tant per cada lletra utilitzada recorrem la matriu solució i sumem al total el trànsit per la distància amb les instal·lacions que hi hagin.
Sigui m el nombre de lletres encara no utilitzades:
Per el segon terme retornarem una matriu C1 de tamany m per m, on cada element 'i' 'j' es el cost que implica colocar la lletra no utilitzada 'i' a la posició no ocupada 'j', respecte les instalacions ja fetes.

Llavors, recorrem les lletres no utilitzades, i per cadascuna, recorrem totes les posicions lliures a la matriu, i ara que tenim la lletra no utilitzada 'i' i la posició no ocupada 'j', recorrem cada instal·lació feta per veure el cost distància per freqüència acumulat que suposaria afegir la lletra a aquesta posició.
Per el tercer terme retornarem una matriu C2 de m per m, on cada element 'i' 'j' es el cost que implica colocar la lletra no utilitzada 'i' a la posició no ocupada 'j', respecte les instalacions no fetes. Aquest cost ho calcularem com el producte escalar de dos vectors, T i D. T és un vector creixent del trànsit de cada lletra lliure amb les altres lletras lliures, de tamany m-1. D es un vector decreixent de les distàncias de cada posició no utilitzada amb les altres, de tamany m-1.

Llavors, recorrem les lletres no utilitzades, i per cadascuna, primer calculem el seu vector T recorrent les altres lletres lliures i després recorrem totes les posicions lliures a la matriu. Ara per cada posició lliure calculem el seu vector D recorrent les altres posicions lliures, i calculem el producte escalar de D i T, que serà l'element que guardarem a la posició 'i' 'j' de la matriu.
Per retornar la cota total, ara primer tindrem que calcular la suma de les matrius C1 i C2, i calcular la seva assignació lineal òptima que farem amb el Hungarian Algorithm, i el resultat de la cota serà la suma del terme 1 més el resultat del Hungarian Algorithm.

Principals estructures de dades:

- **char[][] layout:**
    - Cada node guarda la seva solució actual, que és una matriu de tamany n_filas per n_columnes (dades introduïdes pel usuari), amb un nombre m de lletres ubicades a m posicions, i tot la resta buida.

- **Map<Character, pos> letres_usades:**

    - Cada node guarda les lletres que ha instal·lat i les ubicacions on han sigut emplaçades, de forma que podem consultar fàcilment quines lletres ja tenim i la seva posició.
- **Double[][] Mat_C1**
    - És la matriu on guardem el càlcul de C1. Amb m igual al nombre de lletres no instal·lades, té tamany m per m.
- **Double[][] Mat_C2**
    - És la matriu on guardem el càlcul de C2. Amb m igual al nombre de lletres no instal·lades, té tamany m per m.

- **ArrayList<pos> pos_libres:**
    - És un vector de posicions lliures del layout actual. Aprofitem que hem d'accedir a les posicions lliures tant en el càlcul de C1 com de C2 per omplir aquest vector quan calculem C1 i així aprofitar-ho quan calculem C2.
- **ArrayList<Character> letras_libres:**
    - És un vector de lletres lliures del layout actual. Com amb el vector anterior, aprofitem que en C1 hem d'accedir a les lletres lliures per omplir el vector i després utilitzar lo en C2.
- **PriorityQueue<Double> T:**
    - És la estructura que utilitzarem per guardar el vector T en ordre creixent. Per després fer el producte vectorial, ho recorrem amb un iterador, i accedim a cada element ordenat de menor a major.
- **PriorityQueue<Double>D=new PriorityQueue<>(Comparator.reverseOrder());**
    - És la estructura que utilitzarem per guardar el vector D en ordre decreixent. Per després fer el producte vectorial, ho recorrem amb un iterador, i accedim a cada element ordenat de major a menor.

<u>Greedy</u>

L'algoritme Greedy serveix per trobar una bona cota i solució inicial per començar el algoritme de branch and bound. En aquest cas el que fem és buscar quina és la millor i la pitjor posició, i quina és la lletra més freqüent i la menys. La millor posició serà la que tingui menys distància en total a les altres, i la pitjor serà la que tingui

més. Per tant el que fem és primer recórrer la matriu de trànsit i encontrar la lletra més i menys freqüent i després el mateix per les posicions en la matriu de distàncies. Hem de tindre en compte que l'algoritme fallaria en cas de tenir una matriu en el cual la pitjor i millor posició sigui la mateixa o totes les lletres amb les mateixes freqüències. Per això fem que el primer if en els dos casos la comparació sigui només '>' i en el segon if sigui '<=', per tal de que en el cas de per exemple una matriu 2 per 2, no s'iguali la pitjor i millor posició a la mateixa.
Després retornarem un node amb layout igual a una matriu buida de n_files per n_columnes, amb la lletra més freqüent a la millor posició i la lletra menys freqüent a la pitjor posició. Aquest node ha de tenir el camp letres_usades actualitzat amb les lletres posades al Greedy i les seves posicions.

Principals estructures de dades:

- **char[][] matriz_inicial:**
  - La matriu del node que retornem, és de n-files per n-columnes (dades introduïdes pel usuari) que tindrà la solució inicial del algoritme.
- **Map<Character, pos> ini:**
  - Utilitzem aquesta estructura per inicialitzar el camp lletres_usades del node inicial amb les lletres i posicions utilitzades pel algoritme Greedy.

Hungarian Algorithm:

L'algorisme hongarès, també conegut com l'algorisme de l'assignació de Munkres, és una potent tècnica d'optimització. Va ser presentat per Harold W. Kuhn el 1955 i més endavant revisat per James Munkres el 1957. Aquest algorisme és utilitzat per resoldre problemes d'assignació en temps $O(n^3)$.

.Mitjançant una sèrie d'iteracions, l'algorisme hongarès redueix una matriu de costos, identifica zeros independents i ajusta les assignacions per trobar la millor

combinació possible entre elements de dues llistes. Aquest procés permet trobar la correspondència òptima entre els elements, essent una eina valuosa en la resolució de problemes d'assignació amb diverses aplicacions pràctiques.

En aquest projecte, el farem servir per a calcular l'assignació òptima de les posicions del teclat a un conjunt determinat de lletres. Donada una matriu quadrada de costs, calcularà el cost mínim d'assignar cada lletra a una posició concreta del teclat, de tal manera que totes tinguin una posició assignada i la suma dels costs sigui la mínima possible.

Principals estructures de dades:

- **double[ ][ ] matriu:**
    - Aquesta és la matriu inicial que conté els costos o valors associats a l'assignació d'elements. En el context d'aquest algorisme, representa els costos d'assignar elements d'una llista a una altra.
- **double[ ][ ]copiamatriu:**
    - És una còpia de la matriu original. Es crea per preservar els valors originals mentre es realitzen les operacions d'optimització. Aquesta còpia és útil per a la resta de càlculs i per mantenir els valors inicials intactes.
- **int[ ] zeroFila i int[ ]zeroColumna**:
    - Aquestes llistes indiquen les posicions dels zeros marcats a la matriu. Cada posició d'aquests vectors correspon a una fila o columna, i si una fila o columna té un zero marcat, aquestes llistes enregistren la seva posició.
- **int [ ] filaCoberta i int[ ] columnaCoberta**:
    - Aquestes llistes es fan servir per determinar quines files i columnes estan cobertes durant els càlculs. Si una fila o columna està coberta, el seu valor corresponent en aquests vectors serà diferent de zero.
- **int [ ] zerosEstrellaEnFila:**
    - Aquest vector emmagatzema informació sobre els zeros marcats ('0*') a cada fila específica de la matriu. Cada posició d'aquesta llista està associada a una fila i guarda la columna on es troba el zero marcat.

- **Set<Int[ ]> K:**
    - L'estructura de dades "K" és un conjunt de coordenades que s'utilitza durant el pas 5 de l'algorisme d'assignació hongarès. Aquesta estructura emmagatzema les posicions dels zeros i zeros marcats trobats seguint un patró específic a través de la matriu de costos. Funciona com a guia per eliminar zeros marcats innecessaris, actualitzar les assignacions vàlides i modificar les estructures de control, permetent una manipulació eficient de la matriu per trobar la millor assignació entre els elements de les llistes.

Altres estructures de dades rellevants del sistema:

També hi ha altres estructures de dades rellevants que s'usen en diferents classes del sistema (la majoria, dins dels controladors de persistència de la capa de Dades), i són les que s'esmenten a continuació:

- **TreeMap<String, Alfabet> Alfabets**
    - És un map on la clau és el nom d'un alfabet en minúscula i el valor és l'alfabet en qüestió

- **TreeMap<String, Idioma> Idiomes**
    - És un map on la clau és el nom d'un idioma i el valor és l'idioma en qüestió

- **Map<String, LlistaFrequencies> frequencies**
    - És un map on la clau és el nom d'una llista de freqüències i el valor és la llista de freqüències en qüestió

- **Map<String, Teclat> teclats**
    - És un map on la clau és el nom del teclat i el valor és el teclat en qüestió

- **HashMap<String, Perfil> PerfilsActius**

- És un map on la clau és el nom d'un perfil i el valor és el perfil en qüestió.