Programació Avançada i Estructura de Dades

Grup: S2-GI-G2

Alex Cano Gallego (alex.cano@students.salle.url.edu)

Pol Cuadriello Bravo (pol.cuadriello@students.salle.url.edu)

Eric Faya Esteban (eric.faya@students.salle.url.edu)

Adria Martinez Gallifa (adria.mg@students.salle.url.edu)

Index

Explicació del llenguatge de programació escollit

Hem utilitzat el llenguatge de programació Java, llenguatge orientat a objectes, ja que gràcies a l'arquitectura GRASP ens permet crear tres mòduls on a cadascun tenim les classes que pertanyen a aquest i permet la comunicació entre classes en un nivell molt superior a altres llenguatges aconseguint fer que sigui molt més pràctica la codificació.

Un altre punt a favor de Java és llegir fitxers de dades, ja que ens permet crear una classe que estigui relacionada amb la informació a tractar, un cop llegida es pot guardar a una estructura de dades d'aquesta classe i a través de mètodes accedir a cada camp de la informació obtinguda de manera molt ràpida. També l'hem escollit pels nostres coneixements adquirits al llarg de l'assignatura DPOO que ens permeten anar molt més confiats a l'utilitzar Java.

S'ha de comentar que també té els seus desavantatges optar per aquest llenguatge com poden ser que el llenguatge és dependent, ja que necessita la màquina JVM que ens permet simular el codi, sense aquesta màquina no podem executar programes escrits.

Un altre desavantatge que té utilitzar Java respecte a altres llenguatges coneguts per part nostra com seria el llenguatge C és que és bastant més lent a l'hora de compilar i executar-se que el llenguatge C.

Disseny de l'estructura i justificació de decisions tècniques

Grafs

Per a la implementació del nostre graf en memòria, hem considerat acuradament les opcions disponibles i hem pres decisions basades en l'eficiència i l'optimització de memòria. Després d'avaluar les alternatives, hem optat per utilitzar una representació basada en una llista d'adjacències.

L'elecció d'una llista d'adjacències es basa en diverses consideracions. En primer lloc, una matriu d'adjacències podria resultar ineficient en termes de memòria, especialment si el graf no és densament connectat. En el nostre cas, on estem treballant amb rutes conegudes entre llocs interessants, és poc probable que tots els llocs estiguin directament connectats entre si. Per tant, una llista d'adjacències ens permet estalviar memòria en emmagatzemar només les connexions directes entre els llocs d'interès.

La implementació de la llista del graf es realitza utilitzant la classe ArrayList de Java. Aquesta elecció es basa en les característiques que ofereix ArrayList, com la seva capacitat de créixer dinàmicament i el seu eficient accés als elements. A l'utilitzar la classe ArrayList, podem emmagatzemar tant la llista de llocs d'interès com la llista de rutes conegudes de manera eficient i manipular fàcilment els elements segons sigui necessari.

A més, l'ús d'ArrayList ens facilita l'ús de genèrics, la qual cosa ens permet garantir la coherència dels tipus de dades emmagatzemades en la llista. Això és especialment útil quan treballem amb objectes complexos com els llocs d'interès i rutes conegudes, ja que podem assegurar-nos que les dades emmagatzemades siguin del tipus correcte i evitar possibles errors de tipus.

Arbres

En aquest cas s'ha implementat un arbre AVL el qual és una estructura autobalancejada de l'arbre BST (Binari Search Tree).

L'arbre BST funciona tenint nodes amb dos fills, un major i un menor. Per tant, en el node en què ens trobem, sempre tindrem el subarbre del fill menor que tots els seus nodes tindran un valor menor al node actual i, per altra banda, el subarbre del node major sempre tindrà nodes amb un valor superior.

El posicionament del nostre arbre està dictat pel pes dels habitants. En cada node guardem la classe Habitant la qual conte l'id que serà únic, el nom de l'habitant, el seu pes i el nom del regne al qual pertany com a dades del mateix habitant. A més a més, contindrà dos punters cap als nodes fills i l'altura del node (característica dels arbres AVL) com a dades per estructurar l'arbre.

El problema dels arbres BST és que en el pitjor cas, un arbre pot estar estructurat en línia recta perquè cada dada que s'introdueix és, per exemple, menor a l'anterior. Això resultarà en un arbre que tindrà forma de línia fent que la cerca sigui molt més lenta i no aconseguint l'objectiu de la base de dades. S'obtindria una complexitat en la cerca de $\Theta(\log(n))$ en el millor cas, la qual és bona i una complexitat de

O(n) en el pitjor. Aquesta complexitat no és dolenta, però es pot millorar. Aquí és quan entren els arbres AVL.

La diferència entre un arbre BST i un AVL és que l'AVL guarda l'altura dels nodes perquè sempre estiguin compensats i fent que l'arbre sempre estigui en el millor cas (és a dir, amb els habitants repartits equitativament). Quan es modifica l'arbre i deixa d'estar balancejat es roten els nodes per tindre un altre cop el millor cas (explicat més en detall en la implementació dels algoritmes). Això ens dona una complexitat total de la cerca de O(log(n)).

Arbres R

L'objectiu dels arbres R és optimitzar per guardar, representar, però sobretot cercar i consultar diferents tipus d'informació multidireccional o espacial. Vull fer consultes de forma eficient de la informació en l'espai i, com per exemple cerques en àrea d'una regió, o donat un punt dir quin valor o quins valors estan més proper (kNN). És una consulta logarítmica que passes per tots els punts i té un cost n.

Essencialment, el que farem és anar afegint punts (punts són com les fulles als arbres binaris), el nostre arbre és un conjunt d'àrees o regions, en les que anirem dividint l'espai per ajudar-nos a demanar i a trobar la informació que hi ha en aquest conjunt de dades. És comú trobar-se subàrees dins de les àrees, ja que la idea és anar partint l'espai per poder trobar dades concretes i millor encapsulades o fins i tot les millor organitzades.

També té una inserció amb punts decimals dins d'un espai continu. El seu ordre, equival a la mida màxima d'un node en el que serà un nombre màxim de rectangles o punts.

El funcionament de l'arbre R es pot entendre a través d'un exemple com el que hem vist a classe: es té un node que pot contenir fins a 3 punts, els quals estan situats en un eix x i y. Llavors, s'agreguen punts a aquest node fins que s'excedeix la capacitat màxima del node. En aquest moment, es crea un nou rectangle on es reinseriran els punts, aplicant la lògica inicial (seleccionant els nodes més separats).

Finalment, és desitjable que els nodes de l'arbre R siguin tan específics com sigui possible, o almenys que tinguin nodes pròxims, per a minimitzar el solapament entre àrees. Si un rectangle se superposa amb un altre, dificulta la presa de decisions, per la qual cosa la grandària del rectangle ha de ser suficient per a cobrir tots els punts.

Taules

L'estructura de dades utilitzada en el projecte és una taula hash, implementada mitjançant la classe TablaHash. Una taula hash és una estructura de dades que permet emmagatzemar i recuperar elements de manera eficient.

En una taula hash, els elements s'emmagatzemen en una matriu (o arranjament) de grandària fixa, on cada element es col·loca en una posició determinada per la seva funció hash. La funció hash presa el valor de la clau de l'element i el transforma en un índex de la matriu. Això permet un accés ràpid als elements, ja que la cerca es realitza directament en la posició calculada a partir de la clau.

L'elecció d'una taula hash com a estructura de dades es basa en les següents consideracions:

- Eficiència en la cerca: La taula hash permet un accés ràpid als elements, ja que la cerca es realitza en temps constant, en mitjana.
- Gestió de col·lisions: En cas que dos elements tinguin la mateixa clau hash i es col·loquin en la mateixa posició de la taula, s'utilitza una tècnica de resolució de col·lisions, com l'ús de llistes enllaçades. Això garanteix que els elements amb col·lisió es puguin emmagatzemar correctament i es puguin accedir de manera eficient.
- Flexibilitat en la grandària: En utilitzar una taula hash, no és necessari conèixer per endavant la grandària màxima de l'estructura. La taula pot créixer o reduir-se dinàmicament segons sigui necessari.

- Cerca per clau: La taula hash proporciona una forma eficient de buscar elements per la seva clau, en aquest cas, el nom de l'acusat.
- Comptatge i agrupació d'elements: La taula hash també permet realitzar operacions de comptatge i agrupació d'elements. Això és utilitzat en les funcions optionFourE() i optionFourF() del projecte.

Explicació dels algorismes implementats

Grafs

Exploració del regne

El mètode per a l'exploració del regne, implementa un algorisme que brinda a l'usuari la capacitat d'explorar un lloc específic dins del graf. Comença sol·licitant a l'usuari que proporcioni un ID de lloc. A continuació, es realitza una cerca en la llista dels llocs d'interès per a trobar una coincidència amb l'ID proporcionat. Si es troba un lloc corresponent, es mostra la seva informació, incloent-hi el seu nom, regne i clima. A més, s'itera a través de les connexions del lloc seleccionat i es mostren aquells llocs als quals es pot accedir des d'allí, sempre que pertanyin al mateix regne. En cas de no trobar-se llocs en el mateix regne, es mostra un missatge indicant que no hi ha llocs del regne als que es pot arribar.

En situacions en les quals no es trobi un lloc corresponent a l'ID introduït, es mostra un missatge informant que el lloc no existeix. Aquest algorisme empra la iteració i condicions de coincidència per a mostrar la informació rellevant i manejar adequadament els casos en els quals no es troben coincidències.

Finalment, aquest mètode permet a l'usuari explorar un lloc específic dins del graf, visualitzar la seva informació i descobrir altres llocs relacionats en el mateix regne. Proporciona una forma eficient i estructurada de navegar pel graf, permetent als usuaris obtenir una visió més completa dels llocs d'interès i les seves connexions.

Detecció de trajectes habituals

S'ha utilitzat l'algorisme MST prim per a implementar la detecció de trajectes habituals. Aquest algorisme es va triar perquè connecta tots els llocs d'interès i minimitza la distància total a recórrer. En aplicar-ho al nostre cas pràctic, ens va permetre identificar tots els llocs d'interès mentre minimitzàvem la distància entre ells. A més, donat la grandària considerable dels nostres fitxers de dades proporcionat, ens va interessar enfocar-nos més en l'escala per vèrtexs que per nodes.

L'algorisme MST prim (també conegut com MstPrim) funciona de la següent manera: s'emporta un comptatge dels nodes i les arestes. Després, es comparen els comptadors interns amb els comptadors de l'altre i es procedeix a recórrer els nodes no visitats, seleccionant aquells que es connecten amb nous nodes. No obstant això, existeix la condició que es tria la distància més petita, la qual cosa s'aconsegueix mitjançant una funció auxiliar que selecciona l'aresta de menor pes entre les quals es connecten amb el nou node.

Missatgeria premium

L'algorisme comença amb un bucle while que continua fins que es trobi un camí vàlid o no hi hagi cap camí. Després es marca el node actual com visitat i es recorren tots els nodes veïns del node actual.

A continuació es verifiquen diverses condicions per a determinar si un node veí és vàlid per a ser explorat. Aquestes condicions inclouen verificar la connexió entre els nodes, si el node veí ha estat visitat abans i si compleix amb unes certes característiques.

Es calcula el pontatge actual del node veí sumant el pontatge del node actual amb el temps de viatge entre els nodes, si el pontatge actual és millor que el pontatge emmagatzemat anteriorment per al node veí, s'actualitza el pontatge i s'estableix el node actual com el node previ per al node veí.

Després d'explorar tots els nodes veïns, es verifica si el node actual és el node de destí. Si és així, es reconstrueix el camí europeu emmagatzemant els nodes visitats en ordre contrari.

S'actualitza el node actual seleccionant el següent node amb el pontatge més baix que no hagi estat visitat. Una vegada que es troba el camí europeu o no hi ha camí europeu vàlid, es calcula el pontatge i el camí per al camí africà utilitzant el mateix procediment, es calcula la distància total per al camí europeu i africà sumant les distàncies entre els nodes.

Finalment, es compara el pontatge del camí africà i europeu. Si el pontatge africà és millor i no és -1 (infinit), s'imprimeix la informació del camí africà i es retorna el camí africà. En cas contrari, si el pontatge europeu no és -1, s'imprimeix la informació del camí europeu i es retorna el camí europeu. Si no hi ha camí vàlid, es retorna null.

Arbres

Afegir i eliminar habitant

En afegir un nou habitant, primer de tot es busca que l'id no estigui repetit. Si ho està es retorna un false referint-se que no s'ha pogut afegir l'habitant. Aquest procés es fa amb una cerca mirant primer l'id del pare i després entrant als fills de manera recursiva (Mira tot l'arbre).

Aquesta cerca no es produeix quan s'inicialitzen els arbres dels datasets perquè els ids són únics. És a dir, només passa quan l'usuari introdueix un nou habitant el qual sense saber-ho ha introduït un id repetit.

Si resulta que l'id de l'habitant no hi és en l'arbre, es comença a buscar la seva posició. Aquest procés es realitza comparant el pes de l'habitant. Si el pes del nou habitant és major o igual al del node actual s'entra recursivament al fill dret (o en el nostre programa majorNode) i, si no ho és, s'entra al fill esquerre (minorNode). Al final recursivament s'arribarà a un node null. Aquesta serà la posició del nou habitant.

Després d'això per cada node per on s'ha passat per col·locar el nou node s'ha d'actualitzar el factor de balanceig, la característica dels arbres AVL. Funciona restant les altures dels nodes fills. Si aquesta resulta ser diferent d'1, 0 o -1 significa que l'arbre no està balancejat.

Quan l'arbre no està equilibrat s'ha de reorganitzar. Aquest procés s'anomena rotació i pot ser cap als dos sentits (dret o esquerra). Es poden donar dues situacions: En el primer cas, l'arbre només està descompensat en un sentit, és a dir, el node té un fill menor que té un altre fill menor o al contrari, que tingui un fill major i que aquest tingui un fill major. En aquest cas només s'aplica una rotació en el sentit contrari. I en el segon cas, l'arbre està descompensat en dos sentits alhora. És a dir, que un node té un fill menor que aquest té un fill major o a l'inrevés, que el pare té un fill major que aquest té un fill menor. En aquest cas s'ha d'aplicar primer una rotació cap a fora perquè es doni el primer cas i després s'aplica el mètode del primer cas, és a dir, una rotació en sentit contrari.

En el cas de voler esborrar un habitant l'usuari haurà d'introduir l'id. Primer es buscarà el nom de l'id per retornar-lo a la view per mostrar el nom de l'habitant esborrat. D'aquesta manera, si no es troba l'id, la funció retornarà null i significarà que no existeix tal id. Si resulta que si existeix l'id, es busca altra vegada l'habitant amb la intenció d'esborrar-lo.

Al trobar l'habitant amb la mateixa id, és reemplaçat depenent dels seus fills. Si aquest no tenia nodes fill, és reemplaçat per null. Si només té un fill és reemplaçat per aquest, és a dir, si només té un fill menor i no un fill major, es reemplaça el node a esborrar amb el seu fill menor i de la mateixa manera al revés. Si es dona el cas que té els dos nodes fill el procés canvia. Primer es busca inordre l'habitant, és a dir, començant pel fill major, es busca el fill més petit. Aquest com és a la meitat continuarà mantenint la norma dels arbres BST, ja que a la seva dreta tots els nodes seran majors que ell.

Després de tot aquest procés s'actualitza l'altura dels nodes pels quals s'ha passat i es mira si l'arbre no està balancejat. Si ho està s'aplica el mateix procés per reordenar-lo que en la funció d'addició.

Representació visual

Per a la implementació de la representació visual de l'Arbre, vam intentar implementar el nostre propi algorisme utilitzant la funció per recórrer l'arbre inordre. Després de moltes hores d'implementació i provant l'algorisme que anàvem creant, només vam arribar a implementar a la perfecció el fitxer més petit, el XXS. Aleshores, vam començar a cercar informació per internet per poder obtenir l'algorisme per tal de mostrar qualsevol grandària del fitxer. Llavors, vam trobar per internet a la web de stackoverflow (referència a la bibliografia) un repositori on hi havia diferents maneres de mostrar l'arbre, per tant, vam veure on fallàvem en el nostre algorisme i vam implementar la solució en el nostre codi.

L'algorisme de la funció rep tres paràmetres: l'arbre d'habitants, representat pel node arrel; un booleà que indica si el node actual és el node major o no; i una cadena de text que representa el nivell anterior en la representació gràfica de l'arbre.

L'algorisme funciona de la següent manera: primer verifica si el node actual té un node major. Si és així, es crida de manera recursiva a la funció amb el node major com a paràmetre, actualitzant el booleà major i concatenant espais en blanc a la cadena anterior. Això permet avançar al node major i ajustar la representació visual.

A continuació, s'imprimeix la cadena anterior de la representació visual de l'arbre. Després, s'imprimeix un caràcter de connexió (/ o \ ) depenent de si el node actual és el node major o no. Després, s'imprimeix una línia de guions per a representar la connexió entre el node actual i els seus fills.

Finalment, es crida a la funció printTree per mostrar la informació del node actual en un format específic. Finalment, es verifica si el node actual té un node menor, i si és així, es torna a cridar de manera recursiva a la funció amb el node menor, actualitzant el booleà major i ajustant la representació visual.

```
|               /------|--- King Chergine Conse IX the Minstrel (40492, Aaargh): 145.231
|       /------|--- Goyeal Humberttti IX (268, East Anglia): 135.924
|       |       \------|--- Prince Wllis Knderson the Giant (4712987, Antioch): 125.278
\------|--- Stiara Wrooks the Inquisitor (607, Sussex): 104.51
        |       /------|--- Lbbott Becun the Shrubber (331, Aaargh): 98.208
        \------|--- Baron O'Conrd Janolfsdottir X (833, Anthrax): 87.488
                \------|--- Prince Andrs Lelaie V the Rabbit (673, Essex): 56.599
                        \------|--- Sir Morar Meely the Dead Collector (9628, East Anglia): 50.181
```

Identificació de bruixes

Amb l'estratègia explicada al principi del semestre en els grafs anomenada DFS, l'hem utilitzat en aquest cas amb l'estructura de l'arbre per separar en dos recorreguts diferents. Aquests dos recorreguts s'anomenen preordre i post ordre, bàsicament el preordre té com a objectiu mostrar recursivament els fills del primer node o del node arrel. I el postordre fa totalment el contrari, abans de mostrar el node arrel, comença amb els fills dels fills i va recorrent els "fills dels pares" recursivament fins que acaba en el pare inicial que és el node arrel.

En la funcionalitat d'identificació de bruixes s'havia de cercar els nodes o habitants que tinguessin el mateix, major o menor pes que el pes introduït. Per fer això s'ha escollit l'inordre per fer una cerca més ràpida, perquè només al principi de tot ja ens estalviem cercar la meitat de l'arbre binari, i en el segon pas ens estalviem un quart de l'arbre binari.

Per exemple, per cercar el pes igual s'ha utilitzat el preordre mateix, ja que no teníem clar si era més eficient que utilitzar l'inordre. En el cas de saber si el pes era major o menor, sí que hem utilitzat una cerca amb l'estratègia d'inordre pel motiu que s'ha comentat just abans.

Batuda

La funció implementada per a la batuda s'encarrega de mostrar per pantalla una llista de bruixes dins d'un rang de pesos especificat per l'usuari. L'algorisme utilitzat és bastant senzill i consta dels següents passos.

Primer, es declara una LinkedList anomenada batuda, que serà utilitzada per a emmagatzemar les bruixes que compleixin amb el rang de pesos establert. La llista s'inicialitza mitjançant la crida al mètode getWitches de la classe manager, que rep com a paràmetres dos valors de pes mínim i màxim obtinguts a través d'interaccions amb l'usuari utilitzant la classe viewManager.

A continuació, es realitza una verificació per a determinar si la llista batuda és nul·la. Si és així, significa que no es van trobar bruixes dins del rang especificat, per la qual cosa es crida al mètode printNumWitches de la classe viewManager per a mostrar per pantalla que no s'han trobat bruixes.

En cas contrari, es crida al mètode printNumWitches de la classe viewManager per a mostrar la quantitat de bruixes trobades de la llista batuda. Després, es realitza un bucle for per a recórrer cada element de la llista i cridar al mètode printWitch de la classe viewManager per a mostrar els detalls de cada bruixa en pantalla.

Finalment, aquest algorisme sol·licita a l'usuari un rang de pesos mínim i màxim per cercar les bruixes que compleixin amb aquest rang i mostra la quantitat de bruixes trobades juntament amb els seus detalls.

Arbres R

Afegir i eliminar bardissa

En afegir una nova bardissa, primer l'algoritme busca la millor posició. Això ho fa primer buscant si està contingut en algun dels rectangles del node actual. Si està contingut fa el mateix procés recursivament fins que arriba a un node fulla. Si no està contingut en cap dels rectangles, busca el rectangle menys allunyat de la bardissa a introduir. Això es fa comparant el creixement que haurien de fer els rectangles del node i el rectangle que tingui l'àrea més petita a l'haver afegit la bardissa serà el correcte. Si el node és un de tipus fulla, s'afegeix sense comparar res.

Quan s'ha trobat el node fulla s'afegeix la bardissa a l'array de bardisses i si aquest supera la mida màxima, s'ha de fer un split. En retornar recursivament al bare del node es retorna si és necessari fer un split.

Això es fa separant el rectangle actual en dos, començant per col·locar les dues bardisses més allunyades (que això es calcula amb l'àrea que creen entre elles). Després utilitzant la mida de les àrees que es crearien es van sumant bardisses a cada node. Repartint-se en dos rectangles. En acabar el split es retorna si s'ha de fer un split al node pare perquè l'array de rectangles també pot haver superat la capacitat màxima.

El procés de split és el mateix, només que, en canvi, de punts ara tenim rectangles. És a dir, la tècnica de les àrees és la mateixa, però ara hem de tindre en compte els marges dels rectangles i la posició d'aquests respecte als dels altres rectangles.

Això es va repetint recursivament fins a arribar a l'arrel on el procés és una mica diferent. En aquest cas, si a l'arrel arriba una ordre de split, l'arbre creix en mida. S'ha de crear un nou rectangle que tingui com a rectangle l'actual rectangle arrel i fer un split. Quan això acabi tindrem dos nodes nous, el pare, i el resultant del split de l'arrel. Després el que hem de fer és fer que el punter arrel apunti al pare.

Per la part d'eliminar una bardissa, tenint en compte que hem de cercar la bardissa per la latitud i la longitud, només hem de fer una cerca com la de l'addició. Hem de mirar si el punt està contingut en un dels rectangles, però en aquest cas els hem de mirar tots, ja que potser hi ha rectangles que estan sobreposats uns amb els altres. I amb la diferència també que si en algun punt la bardissa no està en cap dels rectangles, tornem recursivament al node pare perquè miri en el següent rectangle. La funció també retorna un booleà que marca si s'ha trobat la bardissa, perquè llavors, en aquest cas sí que s'ha de deixar de buscar en la resta de rectangles.

Visualització

Per realitzar la visualització del arbre R, hem implementat una classe que es diu RTreeView on implementem una vista en Swing per a la visualització del arbre binari.

La classe RTreeView estén de JPanel i rep un objecte RTree en el seu constructor. En inicialitzar la vista, es crea un JFrame que mostra el títol "RTree VIEW" i s'estableixen les dimensions i la ubicació del marc en funció de la grandària de la pantalla. Es crea un panell d'arbre (treePanel) que estén de JPanel i s'afegeix al marc en el centre.

Dins del panell d'arbre, es crida el mètode paintComponent per a dibuixar l'arbre en funció de les dades proporcionades per l'objecte rTree passat en el constructor. Es crida al mètode drawTree per a dibuixar l'arbre, passant com a paràmetres el gràfic (g), l'arrel de l'arbre (rTree.getRoot) i les coordenades mínimes i màximes (-100) utilitzades per a l'escalat.

El mètode drawTree és l'encarregat de dibuixar l'arbre. Si l'arrel és nul·la, es retorna. Si l'arrel és una fulla, es cridarà al mètode drawBardissa per a dibuixar la fulla. En cas contrari, es cridarà al mètode drawRectangle per a dibuixar el rectangle del node.

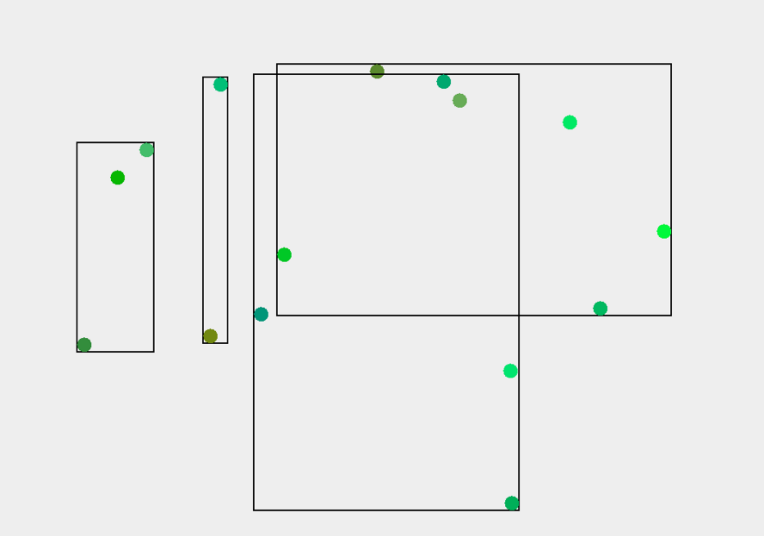
El mètode drawBardissa dibuixa una fulla de l'arbre. Es recorre cada bardissa en el rectangle de la fulla i s'estableix el color. Després, s'escala les coordenades de la bardissa en funció dels límits de la fulla i els límits de la finestra de visualització, i es dibuixa un oval en les coordenades escalades.

El mètode drawRectangle dibuixa el rectangle d'un node de l'arbre. Es recorre cada fill en els rectangles del node. S'escalen les coordenades del fill en funció dels límits del rectangle i els límits de la finestra de visualització. Després, es calcula l'ample i l'altura en funció de les coordenades escalades i es dibuixa un

rectangle en les coordenades escalades. Finalment, es va cridant de manera recursiva el mètode drawTree per a dibuixar els fills del node.

La classe RTreeView també conté mètodes addicionals per a establir l'operació de tancament, l'empaquetat i la ubicació del marc de la vista.

Finalment, hem decidit fer aquest algorisme en una classe per tenir més context en la implementació del Swing i poder implementar els algorismes dins d'aquesta classe i així també evitem possibles col·lapses.



Cerca per àrea

En aquest apartat, realitzem la cerca i visualització de bardisses dins d'una àrea específica. Es comença sol·licitant a l'usuari que ingressi les coordenades del primer i segon punt de l'àrea en format "latitud,longitud". Aquestes coordenades defineixen un rectangle que delimita l'àrea d'interès.

A continuació, es crida al mètode cercaPerArea de l'objecte r_tree_manager per a realitzar la cerca de bardisses dins de l'àrea especificada. Aquest mètode rep les coordenades del primer i segon punt, les processa i realitza la cerca utilitzant el mètode searchBardisses.

El mètode searchBardisses realitza una cerca recursiva a l'arbre R-tree a partir del node arrel. Es passa el rectangle arrel, les coordenades de l'àrea i una llista buida de bardisses. Aquest mètode verifica si el rectangle intersecciona amb l'àrea especificada. Si hi ha intersecció, es realitzen les següents accions:

D'una banda, si el rectangle és una fulla, es recorren les bardisses contingudes en ell. Si una bardissa es troba dins de les coordenades de l'àrea, s'agrega a la llista de bardisses trobades. D'altre banda, si el rectangle no és una fulla, es realitza una crida recursiva al mètode searchBardissasRecursiu per a cada fill del rectangle actual. Una vegada finalitzada la cerca, es verifica si es van trobar bardisses a l'àrea. Si la

llista de bardisses està buida, mostra un missatge informant a l'usuari que no es van trobar bardisses en l'àrea especificada. En cas contrari, es recorre la llista de bardisses i mostra la informació rellevant utilitzant el mètode printRTree de la classe viewManager.

Optimització estètica

En aquesta secció, se'ns va sol·licitar buscar els (k) punts més pròxims segons el valor proporcionat per l'usuari. Per a aconseguir això, hem implementat una solució amb un cost logarítmic mínim. Una opció seria utilitzar un doble bucle per a recórrer tots els punts, però sabem que això tindria un cost de (n^2). Donat el supòsit que podríem tenir milions de punts, seria inviable trobar els (k) punts més pròxims de manera ràpida i eficient.

Per a abordar aquest problema, hem prioritzat els rectangles que requereixen un creixement menor per a incloure el punt en consideració, de manera similar a com es realitza una inserció. No obstant això, és important tenir en compte que la prioritat no es limita a un sol punt, sinó que busquem els (k) punts més pròxims en general.

El nostre enfocament consisteix a visitar primer els rectangles que necessiten un creixement menor per a incloure el punt. Si hi ha dos rectangles disponibles, seleccionem el més pròxim utilitzant la distància o el creixement, la qual cosa es pot determinar fàcilment.

L'objectiu és calcular la distància entre un punt i un rectangle, i després descartar el rectangle en funció de la distància mínima. Per exemple, si tenim 7 punts i busquem els 4 més pròxims, la idea és agregar en una llista els 4 punts ordenats de més pròxim a menys pròxim. Això és similar a les podes (prunning) que es realitzen en el primer semestre del backtracking.

Si tenim un rectangle i la distància entre un punt i aquest rectangle és major al punt més allunyat en comparació amb els punts que ja hem seleccionat, no és necessari visitar aquest rectangle, ja que no millorarà la distància dels (k) punts més pròxims. Aquesta optimització ens permet estalviar temps de manera eficient.

Taules

Afegir i eliminar acusat

Afegir Acusat: Sol·licita a l'usuari el nom, el nombre de conills i la professió d'un acusat. Després, crea un objecte Acusats amb les dades proporcionades i l'agrega a la taula hash utilitzant el mètode put() de la classe TablaHash.

Eliminar Acusat: Sol·licita a l'usuari el nom d'un acusat i l'elimina de la taula hash utilitzant el mètode remove(). Si l'objecte no es troba en la taula, mostra un missatge indicant que l'execució pública ha estat un èxit.

Edicte de gràcia

Sol·licita a l'usuari el nom d'un acusat i verifica si existeix en la taula hash utilitzant el mètode get(). Si l'objecte existeix, se li pregunta a l'usuari si s'ha de marcar com a heretge (I o N). Depenent de la resposta, s'estableix la propietat d'heretge de l'objecte Acusats i s'actualitza en la taula utilitzant el mètode put(). Si l'acusat indicat no existeix o no pot ser heretge (professió de rei, reina o clergue), es mostren missatges corresponents.

Judici final

Per un sol acusat: Sol·licita a l'usuari el nom d'un acusat i obté l'objecte Acusats corresponent utilitzant el mètode get() Després, mostra en pantalla els detalls de l'acusat, incloent-hi el nombre de conills, la professió i si és heretge o no.

Per rang: Sol·licita a l'usuari un rang mínim i màxim de nombre de conills. Després, obté una llista d'objectes Acusats que tenen un nombre de conills dins d'aquest rang utilitzant el mètode.

Histograma per professions

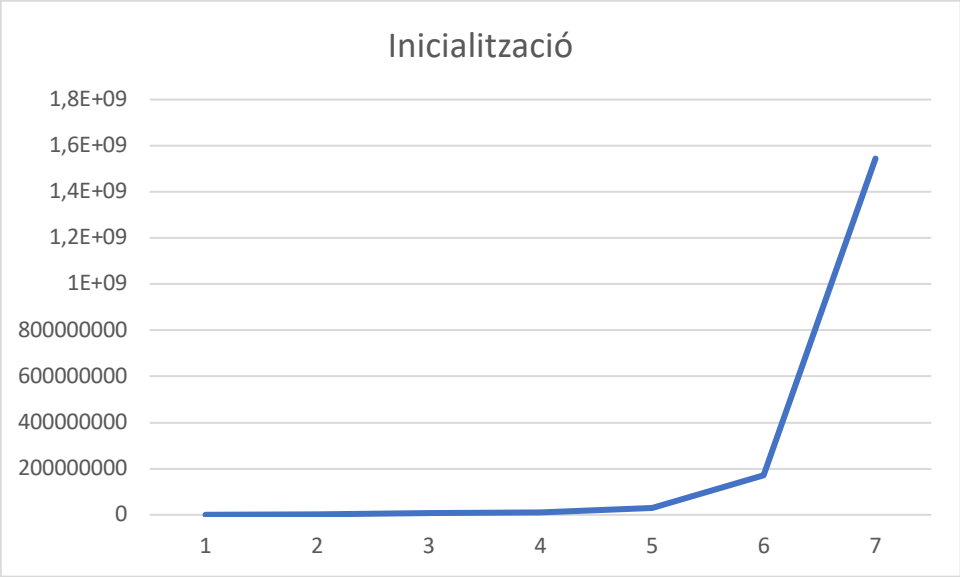
Mostra un missatge de generació d'histograma i crida al mètode contarHerejesPorProfesion() de la taula hash per a comptar el nombre d'heretges per professió i mostrar els resultats en forma de gràfic de barres.

Anàlisi de rendiment i resultats per algorisme

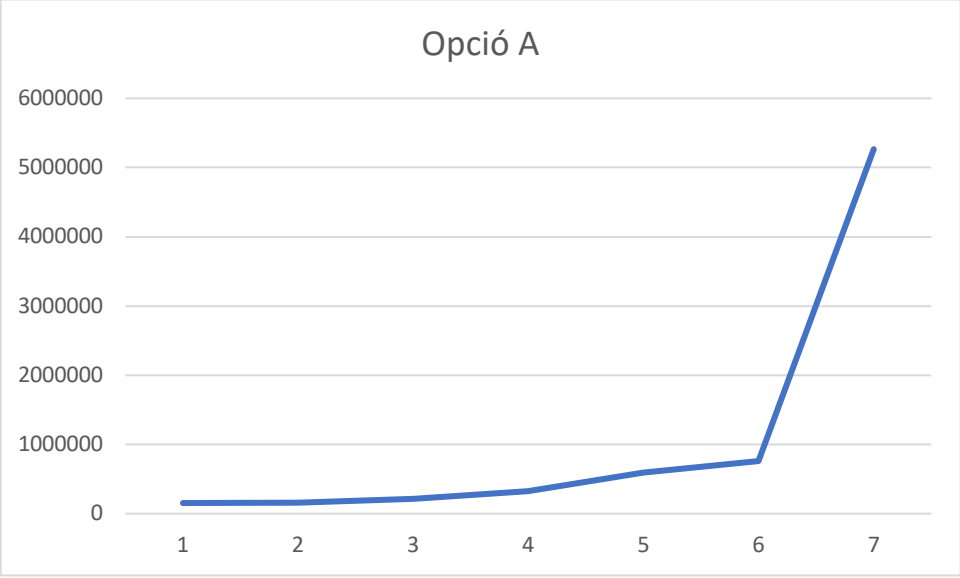
Grafs

El número 1 representa el fitxer XXS i el número 7 el fitxer XXL, i els valors estan en nanosegons:

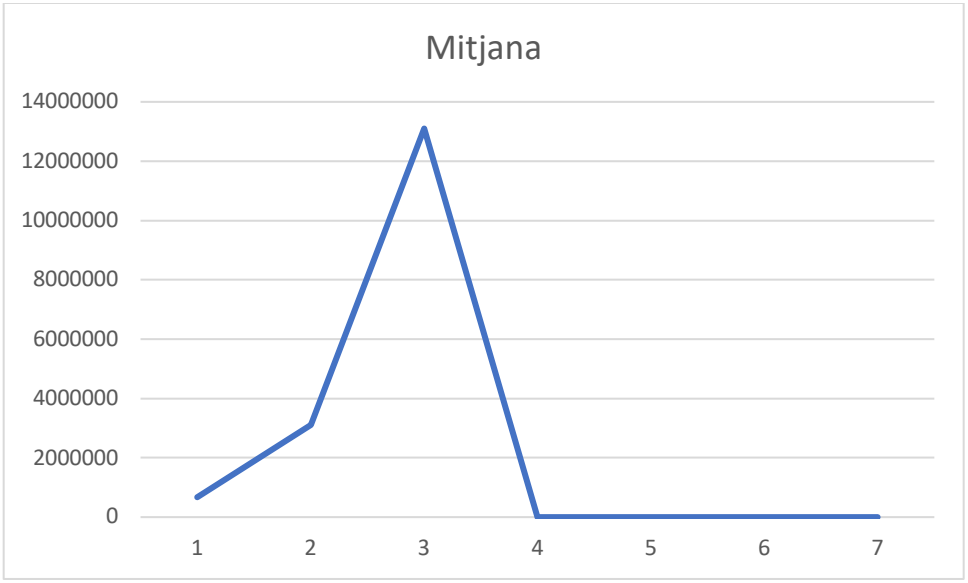
Inicialització



Opció A

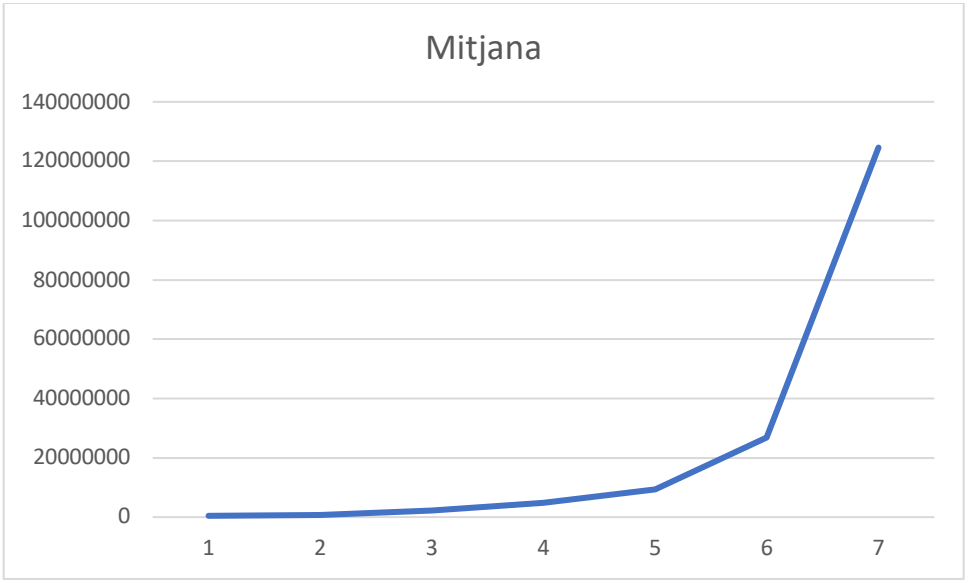


Opció B



En aquesta gràfica, hem tingut un resultat inesperat per part dels fitxers més grans del S. No havíem comprovat en aquest algorisme els casos més grans i alhora de fer les proves, hem vist que no podem calcular el rendiment en aquest cas.

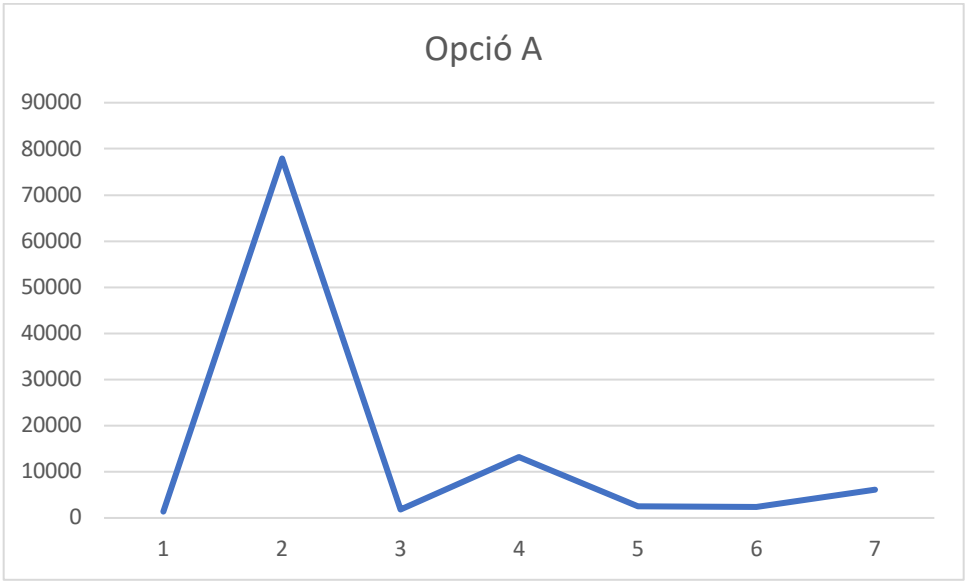
Opció C



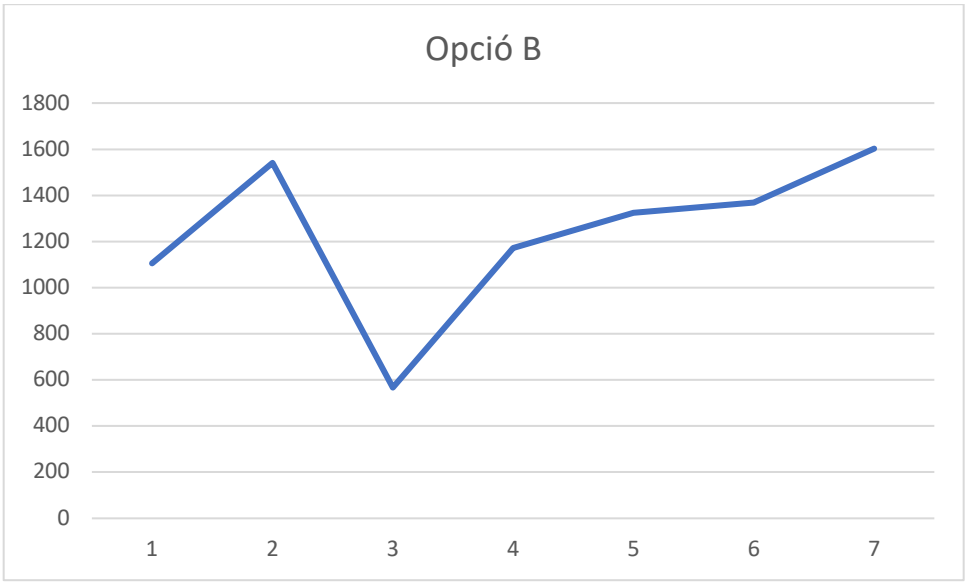
Arbres

El número 1 representa el fitxer XXS i el número 7 el fitxer XXL:

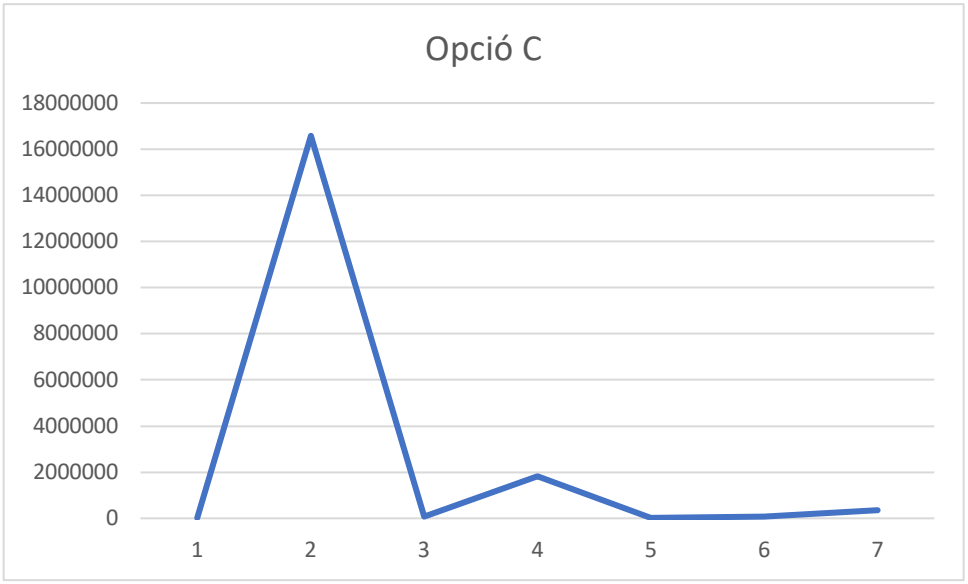
Opció A



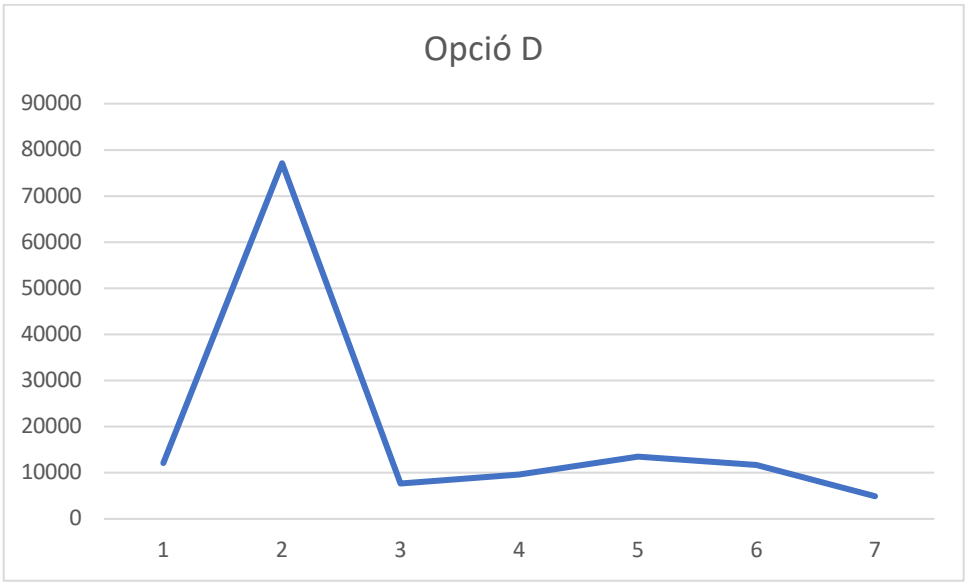
Opció B



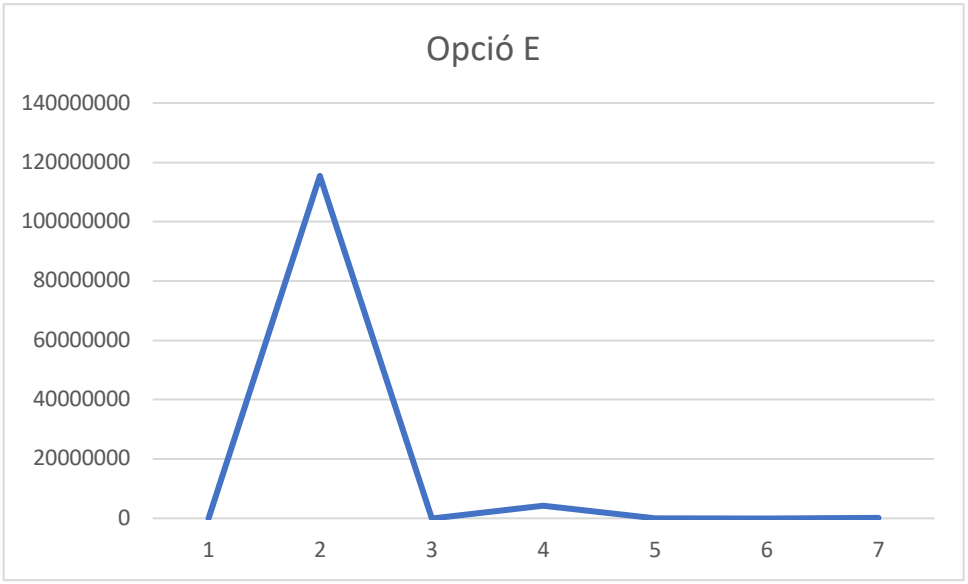
Opció C



Opció D

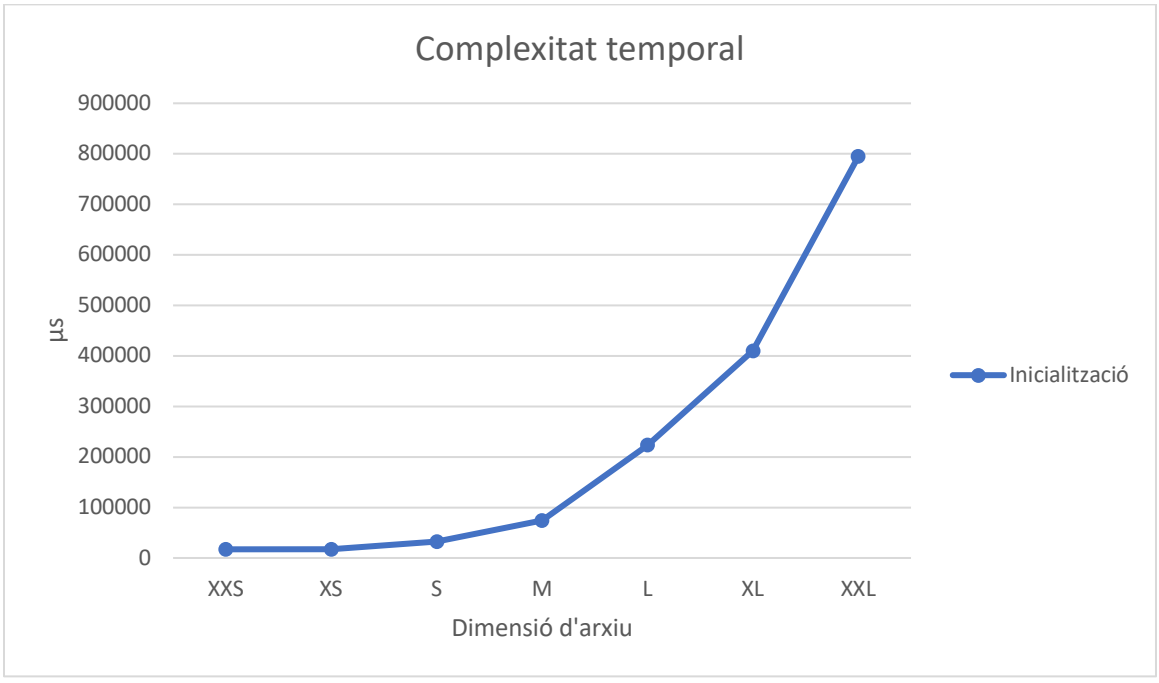


Opció E

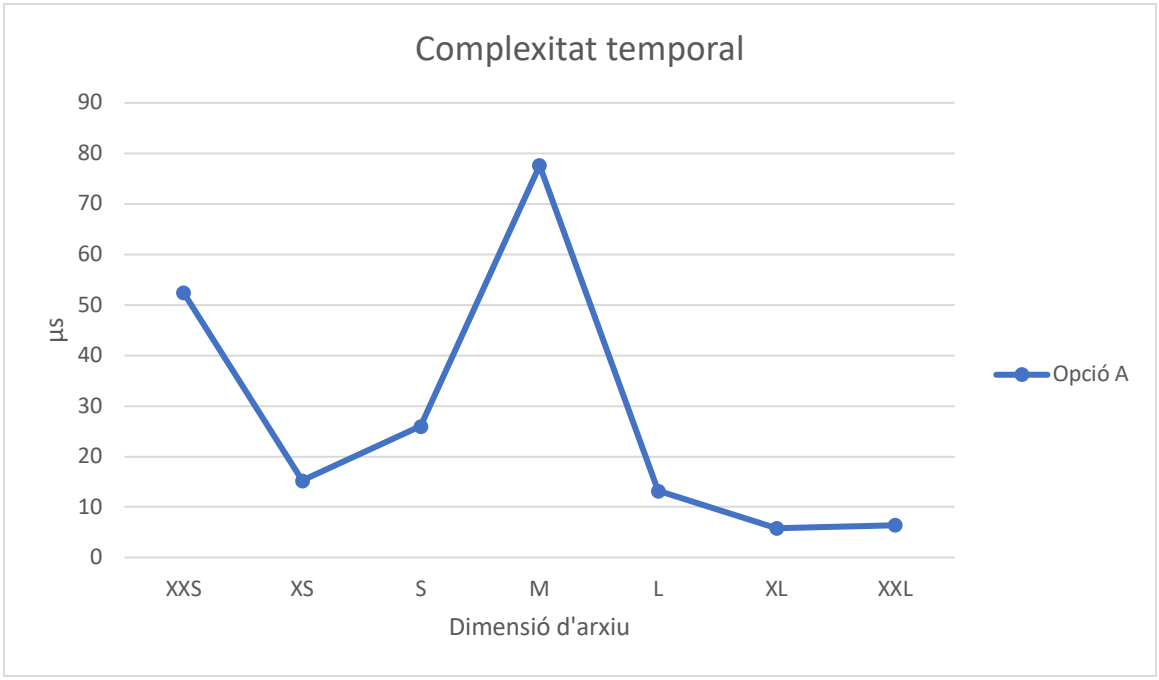


Arbres R

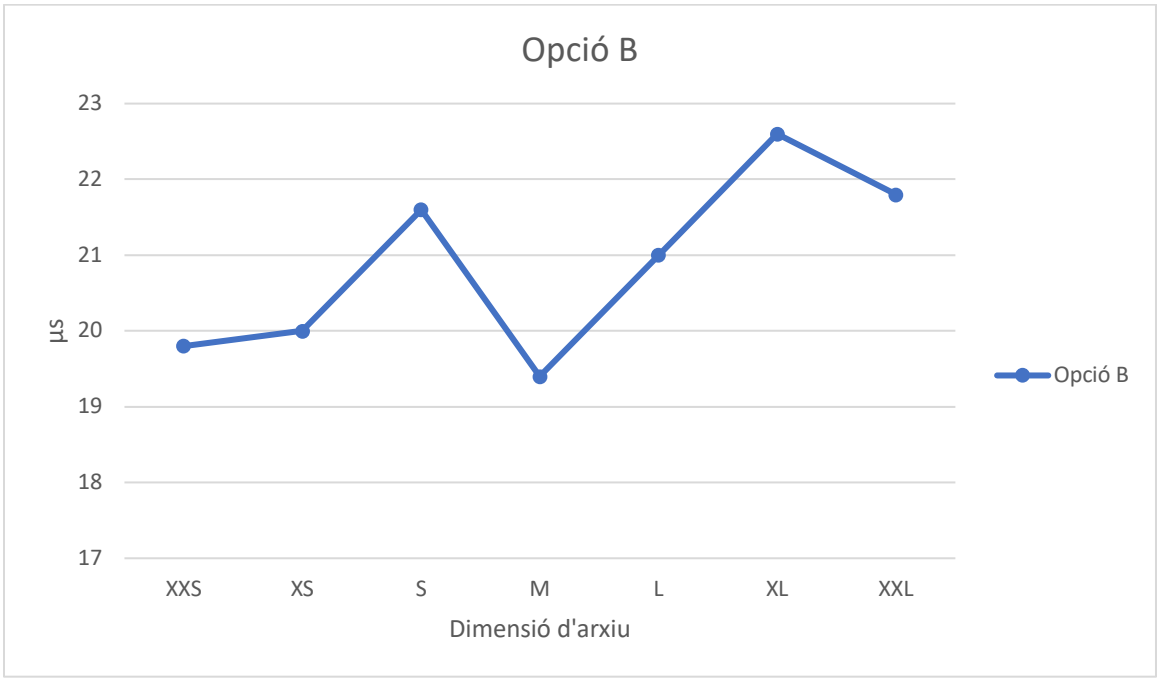
Inicialització



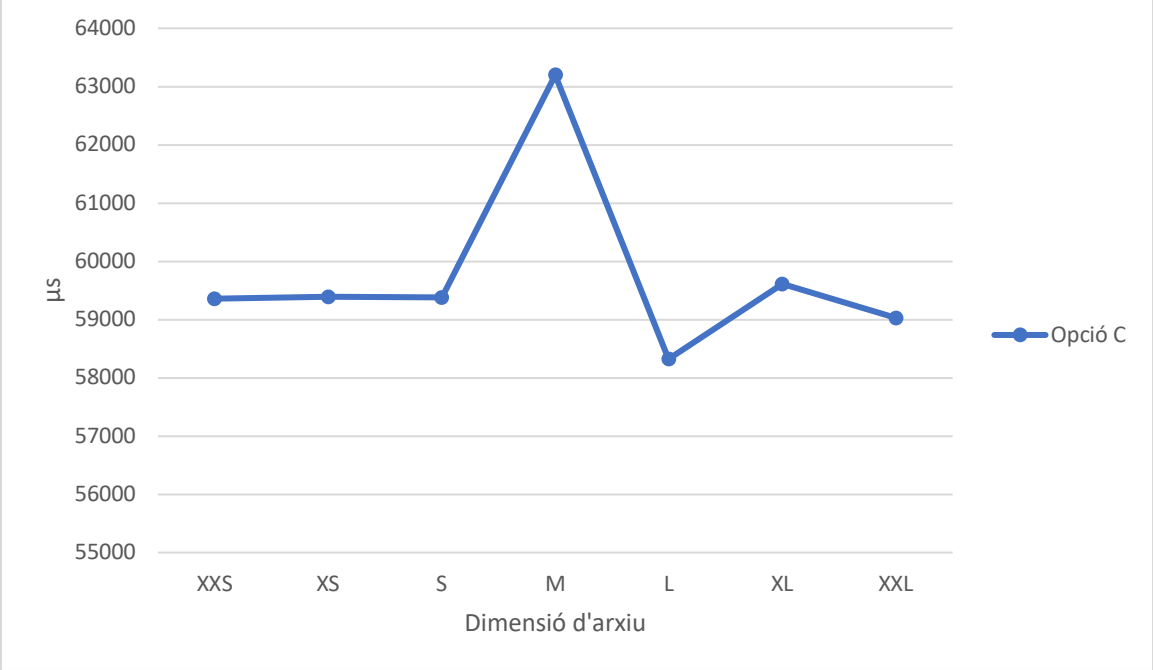
Opció A



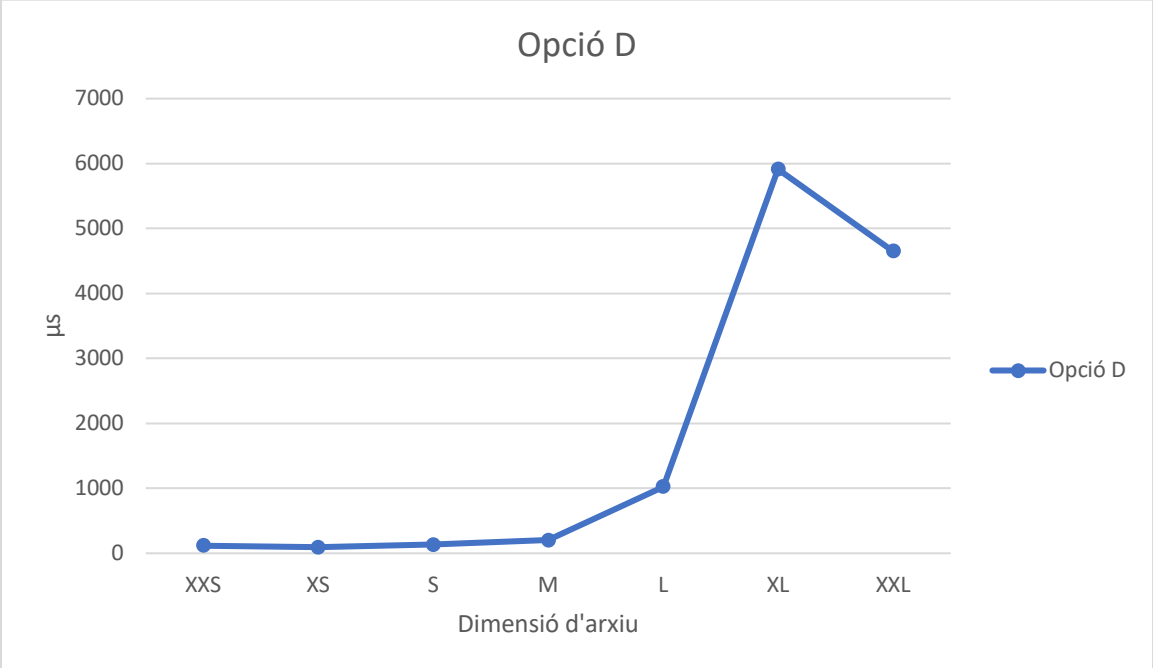
Opció B



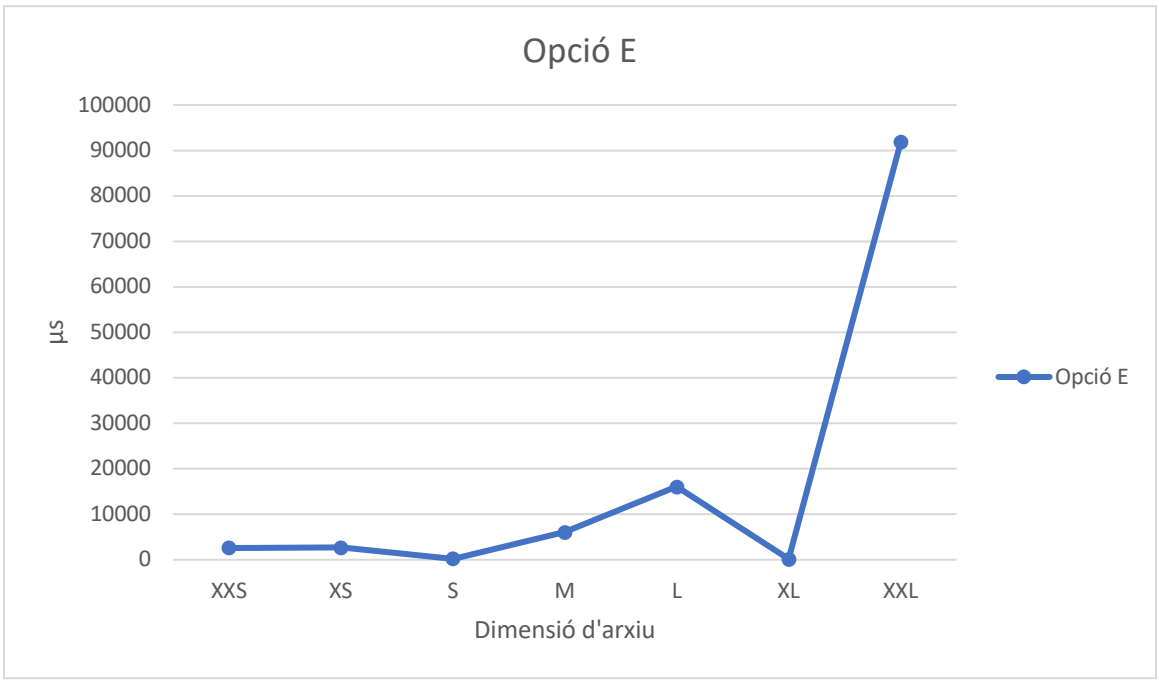
Opció C



Opció D

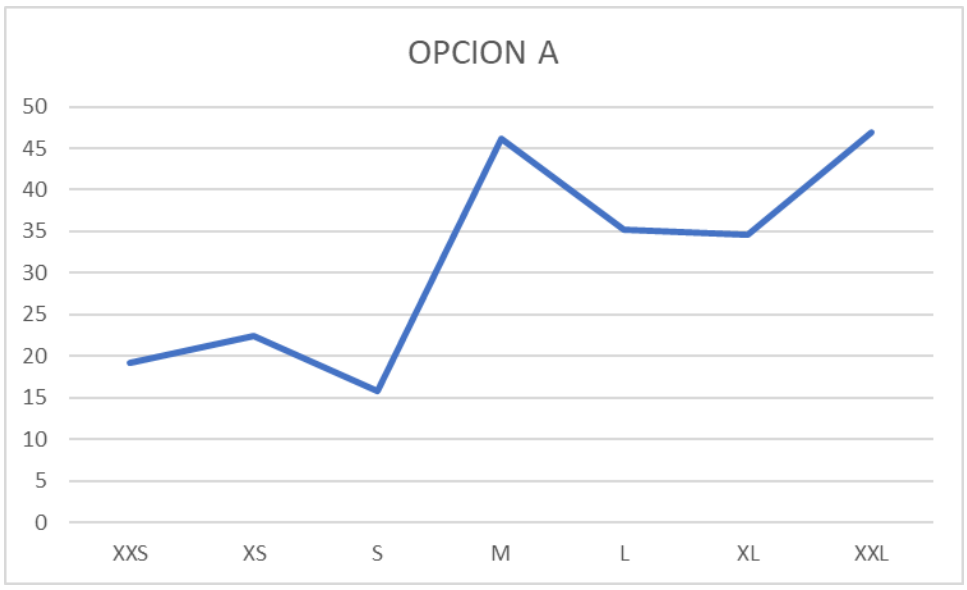


Opció E

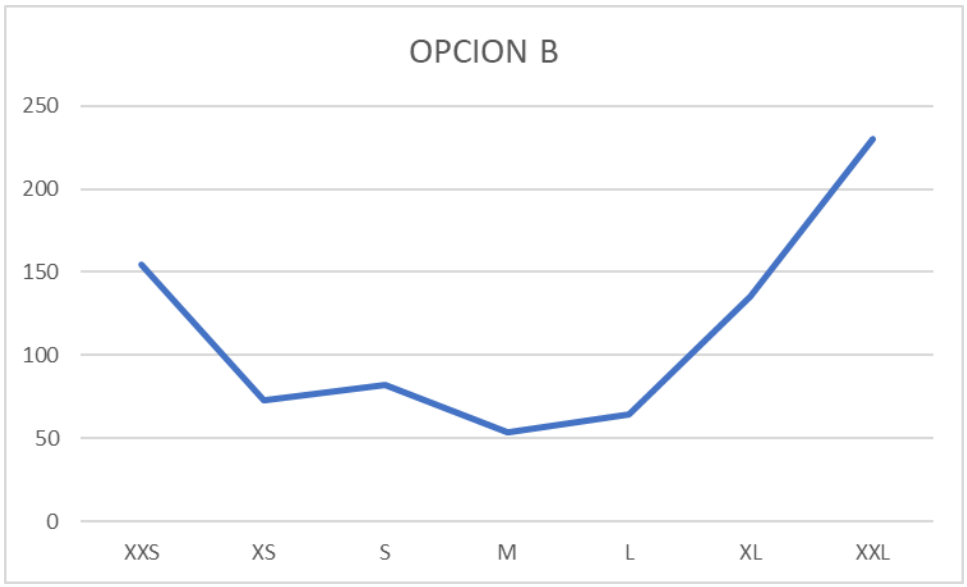


Taules

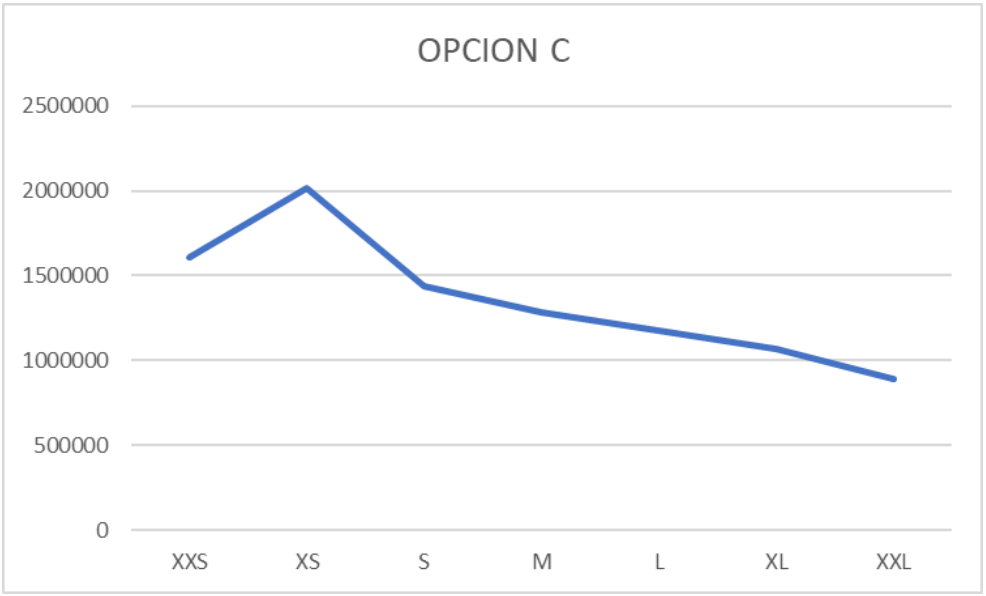
Opció A



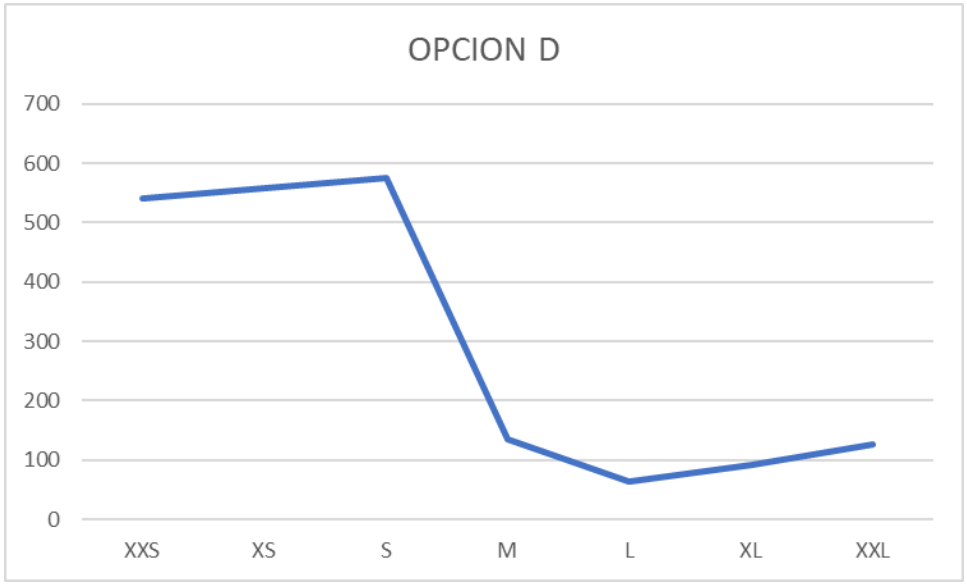
Opció B



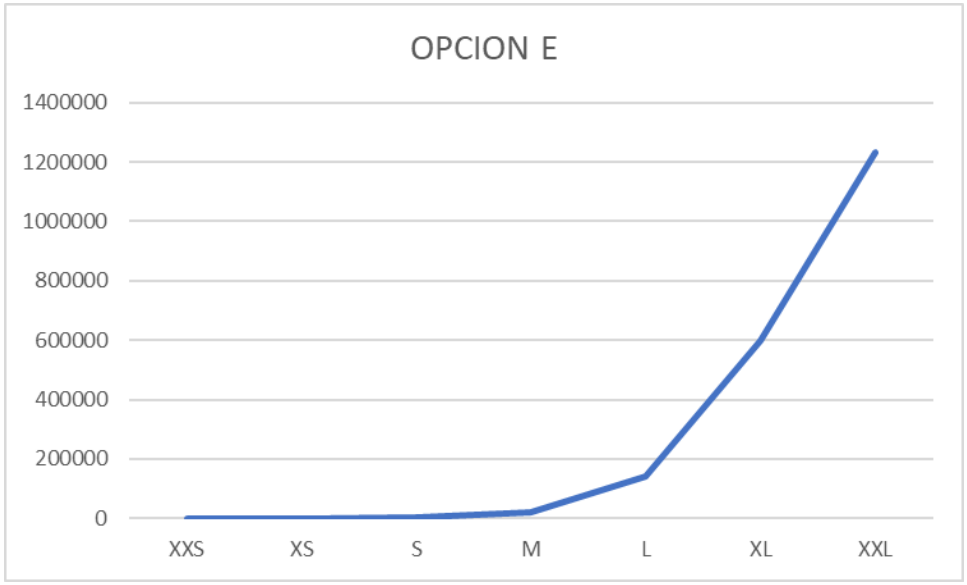
Opció C



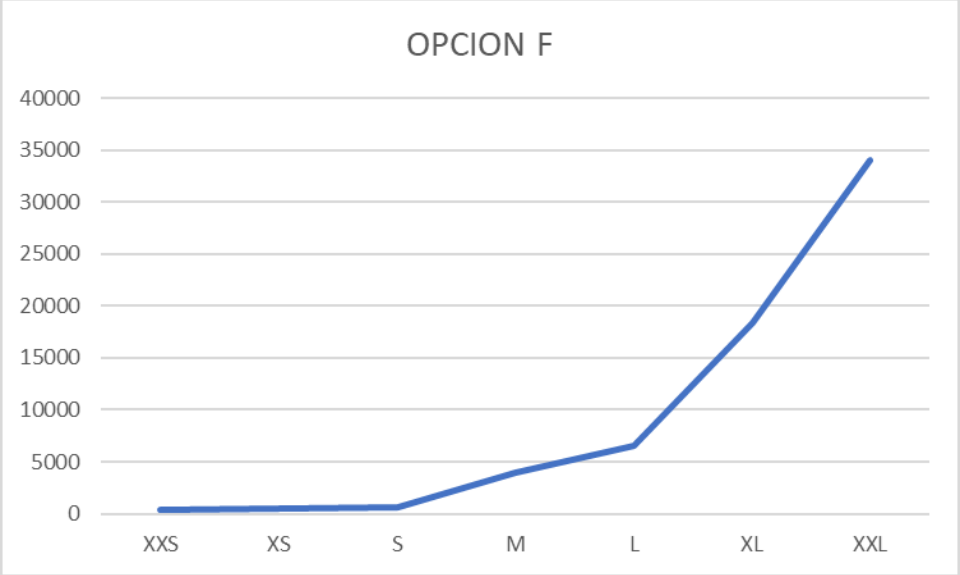
Opció D



Opció E



Opció F


OPCION F
40000
35000
30000
25000
20000
15000
10000
5000
0
XXS
XS
S
M
L
XL
XXL

Explicació del mètode de proves utilitzat

Grafs

El mètode de proves que hem utilitzant ha estat l'ús de fitxers de menor mida per tal de comprovar els resultats que obteníem de cadascun dels algorismes implementats. Amb els fitxers XXS i XS, podíem mirar si tenia coherència el resultat amb l'ordre que havia de prendre mirant els fitxers en si.

Per als errors dels algorismes, hem fet ús del *debugger* que implementa IntelIJ, utilitzant breakpoints als punts crítics per veure les variables en memòria i veure en quin moment i el lloc estava l'error, i si les variables en tot moment eren correctes o no. També hem fet ús de printf per saber en tot moment per la terminal el contingut del graf.

Arbres

Per als arbres, també hem utilitzat el mètode de proves del graf. Hem utilitzat en tot moment els fitxers de les diferents mides per comprovar els resultats que obteníem en cadascun dels algorismes. Sobretot hem fet molt ús dels fitxers XXS i XS, ja que ens permetien veure millor els resultats i comprovar si eren coherents.

Per a la comprovació de l'estructura de l'arbre en memòria hem fet servir molt el debugger d'IntelliJ, ja que ens permetia veure com estava en cada moment l'arbre estructurat. Aquest ens ha permès veure si les accions que fèiem a l'arbre eren correctes o no.

Finalment, per als errors dels algorismes, també hem utilitzat el Debugger, utilitzant breakpoints als punts crítics per veure les variables en memòria per veure en tot moment el lloc on estava l'error i si les variables en tot moment eren correctes o no.

Arbres R

S'ha utilitzat la funció System.nanoTime() per calcular cada algorisme. Primer es guarda el temps d'abans de l'algoritme i després es torna a guardar. Amb això, restant-los i dividint-los per 1000 obtenim el temps en microsegons que triga l'algoritme a fer la seva feina.

Per cada algoritme i per cada arxiu s'ha fet 5 tests i s'ha fet la mitja d'aquests per tindre un resultat més acurat (a la bibliografia hi ha l'enllaç a l'Excel amb totes les dades usades). S'ha representat en un gràfic de línies per veure com el temps varia depenent de l'arxiu.

Taules

Per a cada algoritme s'ha utilitzat la funció System.nanoTime() per a calcular el temps que ha trigat a executar-se, per a assegurar-nos que no hi hagués cap cas extraordinari s'ha fet diversos intents per a cada algoritme i s'ha fet una mitjana de temps, amb els resultats s'han fet gràfics on es poden veure com funciona cada algoritme d'acord amb la quantitat de dades i el temps que ha trigat de mitja.

Problemes observats

Grafs

En la primera part de la nostra pràctica, ens centrem en l'estructura de dades del Graf. Trobem que la majoria dels problemes no estaven relacionats amb el codi en si, sinó més aviat amb com volíem implementar els diferents algorismes.

Inicialment, vam tenir un problema en decidir com volíem que fos el nostre graf per tal de treballar de manera eficient i optimitzar l'ús de memòria. Per tant, des d'un inici considerem que la millor manera és implementar el graf en una llista d'adjacències. Això significava que optimitzaríem l'ús de memòria des d'un inici.

Un altre problema que vam tenir, va ser a la part de la implementació del dijkstra, ja que la manera en la qual vam decidir fer el graf va donar problemes a l'hora de poder recórrer el graf amb aquest algorisme, però es va solucionar creant unes matrius de suport on es guardaven les distàncies entre els diferents nodes, i gràcies a aquestes matrius vam poder fer que funciones l'algorisme.

Arbres

En la implementació dels arbres, vam tenir alguns problemes. L'inconvenient va sorgir perquè inicialment implementem un arbre de cerca binària (BST) en lloc d'un arbre AVL. La solució per a aquest problema va ser clara: vam haver de reemplaçar l'arbre BST per un arbre AVL, que garanteix un equilibri adequat i un rendiment òptim en les operacions de cerca i modificació.

A més, enfrontem un altre problema en la part de la visualització dels arbres. Vam intentar desenvolupar el nostre propi algorisme per a aquesta tasca, però només aconseguíem representar correctament els arxius XXS. Com a resultat, ens vam veure obligats a buscar informació a internet per a trobar una solució que ens permetés visualitzar adequadament qualsevol grandària dels fitxers proporcionats. Això ens va permetre resoldre el problema de visualització i garantir una representació precisa dels arbres en tots els casos.

Arbres R

Durant la implementació, ens enfrontem a diversos desafiaments que requerien solucions adequades per a garantir el funcionament correcte del sistema.

Un dels problemes identificats estava relacionat amb els conjunts de dades més grans, on la càrrega d'informació no es realitzava correctament. Després d'investigar a fons, descobrim que l'error residia en el procés de fer un split dels nodes no fulla. Específicament, en realitzar el split, es requeria passar el rectangle que superava el límit màxim a la funció corresponent. No obstant això, notem que sempre apuntava a un rectangle nul, la qual cosa causava la càrrega incorrecta de la informació. Per a solucionar aquest problema, realitzem modificacions en l'algorisme perquè aquest busqués el rectangle que superava el límit de manera interna, la qual cosa va simplificar el codi i va permetre que la càrrega d'informació es realitzés adequadament fins i tot en conjunts de dades més grans.

Un altre problema significatiu que trobem estava relacionat amb el maneig de les mesures dels rectangles que no eren nodes fulla. Observem que les actualitzacions de les mesures no es realitzaven correctament, la qual cosa ens va portar a investigar més a fons la causa subjacent. Descobrim que el codi no tenia en compte un aspecte important: un rectangle pot ocupar tots dos límits del mateix eix simultàniament. Aquesta falta de consideració en el càlcul dels límits afectava l'actualització precisa de les mesures dels rectangles. Per a solucionar aquest problema, realitzem modificacions en el codi perquè tingués en compte aquesta situació i, finalment, les mesures dels rectangles es van actualitzar correctament.

A més, ens trobem amb un desafiament relacionat amb el split de nodes que no eren fulla. Descobrim que aquests nodes no s'inicialitzaven correctament, la qual cosa generava resultats inesperats. En investigar més a fons, trobem que la distància entre els rectangles es calculava utilitzant el punt central de cada rectangle. No obstant això, aquest enfocament no proporcionava els resultats desitjats. Per a solucionar aquest problema, canviem la manera de calcular la distància. En lloc de basar-nos en el punt central, calculem l'àrea del rectangle suposat que es formaria en prendre l'àrea més gran entre els dos rectangles més allunyats. Aquesta modificació en el càlcul de la distància va permetre una divisió adequada dels nodes i va evitar els errors previs.

Finalment, en la part de la visualització del sistema, utilitzant la biblioteca Swing, ens enfrontem a un problema en la ubicació correcta dels rectangles i les bardisses en la interfície gràfica. Quan intentàvem mostrar les coordenades dels rectangles i les bardisses, notem que es situaven en punts no desitjats de la finestra de visualització. Aquest problema es devia a la necessitat d'adaptar les coordenades dins de la finestra de Swing. Per a resoldre-ho, creem una funció que escalava els valors de totes les coordenades perquè s'ajustessin correctament dins de la finestra de visualització de Swing.

Taules

Quant a les taules no ens hem trobat amb cap problema a l'hora de realitzar aquesta part de la pràctica, tot s'ha realitzat i implementat tal i com tenim previst.

Conclusions

Les estructures de dades són fonamentals en la programació moderna, ja que ens permeten organitzar i representar eficientment conjunts de dades en la memòria d'un ordinador. En aquesta pràctica, hem tingut l'oportunitat d'explorar i comprendre el funcionament de tres de les estructures de dades més importants en l'actualitat, a més d'una estructura més específica coneguda com l'Arbre R.

Durant la nostra implementació, ens hem enfrontat a diversos desafiaments i problemes que han requerit solucions adequades. Hem après a identificar i solucionar errors en la càrrega de dades, el càlcul de mesures i la divisió de nodes en les estructures d'arbres. A més, ens hem endinsat en la visualització de les dades utilitzant biblioteques com a Swing.

Aquesta experiència ens ha permès apreciar la importància de triar l'estructura de dades adequada per a cada situació i comprendre com implementar-les internament per a optimitzar el codi i maximitzar l'eficiència. El coneixement adquirit sobre aquestes estructures ens proporciona una base sòlida per a abordar futurs desafiaments en el desenvolupament de solucions pràctiques en el món real.

En conclusió, les estructures de dades són eines essencials en el desenvolupament de programari eficient. A través d'aquesta pràctica, hem enfortit la nostra comprensió de les estructures d'arbres, i en particular, hem adquirit experiència en l'ús de l'Arbre R. Estem preparats per a aplicar aquest coneixement en situacions reals i aprofitar al màxim els avantatges que ofereixen aquestes estructures en la resolució de problemes i la manipulació de dades.

Finalment, podem veure que la teoria donada a classe ens ha servit per aplicar i enfortir els nostres coneixements i augmentar positivament la nostra experiència en aquest nou àmbit.

Bibliografia

Hem consultat les següents fonts d'informació per poder fer de la millor manera la pràctica:

- PAED. Date: 2023, Març 10. Apunts 2n Semestre – Available In: https://estudy2223.salle.url.edu/course/view.php?id=1324&sectionid=1451
- *AVL Tree (Data Structures) - javatpoint*. (s. f.). www.javatpoint.com. https://www.javatpoint.com/avl-tree
- *AVL Tree*. (s. f.). https://www.programiz.com/dsa/avl-tree
- *Data Structure Visualization*. (s. f.). https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
- dbislab. (2021, 25 enero). *034 r tree* [Vídeo]. YouTube. https://www.youtube.com/watch?v=hUIHtPLL940
- dbislab. (2021, 25 enero). *034 r tree* [Vídeo]. YouTube. https://www.youtube.com/watch?v=hUIHtPLL940
- Excel de proves grafs: provesgraf.xlsx
- Excel de proves arbre: provesArbre.xlsx
- Excel de proves arbres R: pruebasPAED.xlsx
- Excel de proves taules hash:pruebasTaulesHash.xlsx
- *How to print binary tree diagram in Java?* (s. f.). Stack Overflow. https://stackoverflow.com/questions/4965335/how-to-print-binary-tree-diagram-in-java