Pràctica Scala: Similitud entre Documents
Programació Declarativa: Aplicacions: 2023

Aniol Molero Grau i Bernat Comas Machuca

17 de novembre de 2023

Índex

1 Introducció

Aquesta pràctica consisteix a crear una aplicació amb Scala amb l'objectiu de poder comparar documents entre sí, per saber com de similars són. Això ho farem mitjançant *Vector Space Models*, que seran vectors que contindran cada paraula d'un text i el pes associat a aquesta, que serà un valor entre 0.0 i 1.0. Aquests vectors s'utilitzaran per computar la similitud entre dos fitxers, mitjançant la similitud de cosinus.

Aquest treball consta de dues parts. En la primera compararem dos textos de llibres assignant un pes a cada paraula dins de cada text, és a dir, que emprarem els vectors de *term frequencies (tf)* (frequència de paraules), i on tots els càlculs es realitzaran de manera seqüencial. En la segona utilitzarem arxius xml de la Vikipèdia, i la similitud es podrà calcular amb molts documents a la vegada, assignant pesos a les paraules en funció de com de significatives són dins el corpus (conjunt de pàgines), de manera que farem servir els vectors *tf_idf* (*term frequency* i *inverse document frequency*) i efectuarem operacions en paral·lel mitjançant l'algorisme MapReduce. A més, en aquesta segona part no només tindrem un sol text, sinó que dividirem els fitxers en títol, contingut i referències. D'aquesta manera, a banda de detectar pàgines similars podrem dur a terme altres accions com determinar quines són les pàgines més rellevants (les que són referenciades més vegades) o calcular la mitjana d'imatges o referències en les pàgines.

L'algorisme de MapReduce mencionat anteriorment és una tècnica que normalment s'utilitza quan cal tractar amb conjunts de dades molt grans, i en casos en què la feina es pot paral·lelitzar, ja que aquest és un dels seus avantatges més importants. Una vegada s'ha definit l'estructura general del MapReduce cal crear una funció de mapping i una funció de reducing per cada tasca que vulguem dur a terme. La capacitat de paral·lelització d'aquesta tècnica rau en el fet que podem tenir múltiples *mappers*, que faran servir la funció de mapping, i *reducers*, que usaran la funció de reducing, treballant a la vegada. Generalment, cada *mapper* rebrà una part de l'entrada inicial i la tractarà mitjançant la funció de mapping. Tots els resultats obtinguts s'aniran acumulant en un mapa el qual, una vegada tots els *mappers* hagin acabat, es dividirà en parts per a ser tractat per cada *reducer*, mitjançant la funció de reducing. Quan l'últim reducer hagi acabat la seva feina, l'algorisme de MapReduce ens donarà el resultat obtingut.

2 Descripció de les classes del projecte

2.1 Primera part

Per a la primera part hem utilitzat únicament un objecte, al que hem anomenat SimilitudEntreDocuments. Hem utilitzat un únic objecte perquè simplement necessitàvem un singleton que contingués mètodes estàtics.

Aquesta classe conté mètodes per executar cada un dels apartats que podíem veure a l'enunciat de la pràctica:

- printWordOccurrence: Imprimeix la llista de paraules més freqüents. Pot rebre com a paràmetre una llista amb stopwords o sense.
- paraulaFreqFreq: Donat un string, computa les 10 freqüencies més freqüents. És a dir, ens mostra per pantalla una llista amb cada nombre d'aparicions en quantes paraules es dona dins l'string rebut per paràmetre.
- displayNGrams: Semblantment a printWordOccurrence imprimeix per pantalla el nombre de vegades que apareixen els ngrames més freqüents en l'string rebut per paràmetre. Es pot modificar la mida de l'ngrama que rep també amb un paràmetre. Si es busquen ngrames de mida 1, és idèntic a fer paraulaFreqFreq però sense mostrar la freqüència relativa.
- cosinesim: Retorna el coeficient de similitud utilitzant la tècnica del cosinesim entre els dos strings passats per paràmetre. També rep una llista de stopWords de paraules a no tenir en compte.

2.2 Segona part

Per a la segona part hem utilitzat diverses classes que ja ens venien donades i les quals hem modificat poc (o gens). A banda d'aquestes no n'hem hagut d'utilitzar més, i la feina ha consistit a instanciar-les i preparar els seus paràmetres:

- ResultViquipediaParsing: Aquesta classe es tracta d'una *case class* que té tres camps, títol, contingut i referències. Aquesta classe ens permet guardar la informació necessària de manera pràctica en un sol objecte, fent que sigui més fàcil treballar amb les dades. A banda del fet que aquesta classe serveix per obtenir les dades que s'acabaran passant a l'algorisme de MapReduce, no té cap relació amb les classes que explicarem a continuació.
- MapReduce: Aquesta és la classe principal de la nostra aplicació, i és la que s'encarrega d'utilitzar les altres classes. Per crear-ne una instància cal passar-li unes dades d'entrada, que són les que es tractaran, i també les funcions de mapping i reducing que caldrà fer servir per a dur a terme la seva feina. A més, també necessitarà saber quants mappers i reducers haurà d'usar. Una vegada una instància d'aquesta classe rebi un missatge MapReduceCompute començarà la seva feina. En primer lloc es crearan els mappers indicats, i a cada mapper se li assignarà tractar part de les dades d'entrada mitjançant missatges toMapper. Després, el MapReduce s'esperarà a rebre tants missatges fromMapper com mappers creats. Seguidament, es crearan els reducers indicats, i es repartirà la tasca de tractar els resultats obtinguts amb missatges toReducer. D'una manera semblant a la d'abans, el MapReduce s'esperarà a rebre tants missatges fromReducer com reducers creats. Finalment, s'enviarà el resultat final al client.

Cal tenir en compte que aquesta classe utilitza paràmetres polimòrfics de tipus K1,V1,K2,V2 i V3. L'input ha de ser de tipus List[(K1,List[V1])], la funció de mapping de tipus (K1,List[V1]) → List[(K2,V2)] i la funció de reducing de tipus (K2,List[V2]) → (K2,V3). També rep dos enters, que representen el número de mappers i de reducers.

- MapReduceCompute: Es tracta d'una *case class* que simplement té la funció d'engegar el procés del MapReduce. Per a fer-ho només cal enviar un MapReduceCompute al MapReduce.
- toMapper: Una altra *case class*. S'utilitza per enviar missatges als mappers. Fa servir paràmetres de tipus polimòrfics de la mateixa manera que el MapReduce. Se li ha de passar un K1 i un List[V1].
- Mapper: Aquesta classe servirà per instanciar mappers, els quals hauran de realitzar la primera part de la feina del MapReduce. També utilitza un paràmetre de tipus polimòrfic com el MapReduce, el qual es tracta de la funció de mapping que, com hem dit abans, ha de ser de tipus (K1,List[V1]) → List[(K2,V2)]. Cada mapper envia els seus resultats al MapReduce amb un missatge fromMapper.
- fromMapper: Una altra *case class*. S'utilitza per enviar missatges d'un mapper al MapReduce. Fa servir un paràmetre polimòrfic com el MapReduce. En aquest cas és el resultat obtingut amb la funció de mapping, que ha de ser de tipus List[(K2,V2)].
- toReducer: Una altra *case class*. S'utilitza per enviar missatges als reducers. Fa servir paràmetres de tipus polimòrfics de la mateixa manera que el MapReduce. Se li ha de passar un K2 i un List[V2].
- Reducer: Aquesta classe servirà per instanciar reducers, els quals hauran de realitzar la segona part de la feina del MapReduce. Utilitza un paràmetre de tipus polimòrfic com el MapReduce, el qual es tracta de la funció de reducing que, com hem dit abans, ha de ser de tipus (K2,List[V2]) → (K2,V3)). Cada reducer envia els seus resultats al MapReduce amb un missatge fromReducer.
- fromReducer: Una altra *case class*. S'utilitza per enviar missatges d'un reducer al MapReduce. Fa servir un paràmetre polimòrfic com el MapReduce. En aquest cas és el resultat obtingut amb la funció de reducing, que ha de ser una tupla de tipus (K2,V3).

També hem fet servir alguns objectes que hem utilitzat bàsicament per organitzar una mica el codi i no tenir-ho tot en un sol fitxer:

- JocsDeProves: Tal com indica el seu nom, aquest fitxer conté tot de jocs de proves, que duen a terme execucions amb paràmetres arbitraris.
- MappersAndReducers: Aquest objecte conté totes les funcions de mapping i de reducing que utilitzem en la nostra aplicació.
- SimilitudEntreDocuments: Ve a ser el mateix objecte que vam utilitzar a la primera part de la pràctica.
- ProcessListStrings: Aquest objecte ja venia inclòs en el codi inicial, però hi hem afegit la funció getFileParts que, donat el nom d'un fitxer, ens retorna una tupla amb el seu títol, contingut i llista de referències.

3 Circuits d'execució de les diferents opcions

L'objectiu d'aquesta secció és explicar seqüencialment els processos que segueixen les diferents opcions que el menú de la part 1 permet executar. D'aquesta manera veurem les diferents decisions de disseny que hem hagut de prendre al llarg de la pràctica però sense entrar en l'explicació de mappers i reducers. Les explicacions de mappers i reducers es troben al punt 5.

Per cada una de les seccions es podrà veure també una figura on es mostra l'esquema del flux d'execució d'aquella secció. L'esquema es llegeix de dalt a baix i d'esquerra a dreta. Es pot veure pintat en groc la funció d'entrada o Main, en verd les funcions que ja ens venien fetes, en blanc les funcions que nosaltres hem implementat, i en taronja cada una de les crides que s'ha fet a la nostra funció genèrica que instancia un MapReduce.

3.1 Calcular mitjana de referències

Per calcular la mitjana de referències primer es llegeixen els mappers i reducers que l'usuari vol que s'utilitzin. En aquest cas no cal indicar res més, ja que considerem totes les pàgines que tenim. Seguidament, cridem la funció timeMeasurement, passant-li la funció meanReferences, que, com indica el seu nom, calcularà la mitjana de referències, juntament amb una tupla que conté el nom de la carpeta on es troben els fitxers i el nombre de mappers i reducers que hem llegit abans, en aquest ordre. timeMeasurement simplement cridarà la funció que li hem passat amb els arguments indicats, i compta quant tarda a acabar.

Llavors, quant a la funció meanReferences, s'obté la llista de fitxers utilitzant el primer element de la tupla que es passa per paràmetre, que com hem dit, és el nom de la carpeta on tenim els fitxers, i processem tots els fitxers, quedant-nos només amb el títol de cada pàgina i la seva llista de referències. A continuació, ho passem a MRGetRef, juntament amb el nombre de mappers i reducers, que s'encarrega de cridar la funció MR indicant que l'input és aquesta llista de tuples de títols i llista de referències, fent servir les funcions mappingSD i reducingSD, juntament amb el nombre de mappers i reducers indicats. Del resultat, que és un mapa que té com a claus els títols i com a valors el nombre de referències a la pàgina, ens quedem només amb aquells que es trobin a paginesKeys, que és un Set que conté els títols de tots els fitxers que tenim, de manera que les pàgines que no formen part del nostre corpus queden excloses.

Altra vegada a meanReferences, sumem tots els valors del mapa mencionat anteriorment, i ho dividim entre el nombre de fitxers per obtenir la mitjana de referències.

Les funcions de mappingSD i reducingSD es troben explicades **aquí**. El circuit d'execució que segueix la mitjana de referències és el que es pot veure a la següent figura:

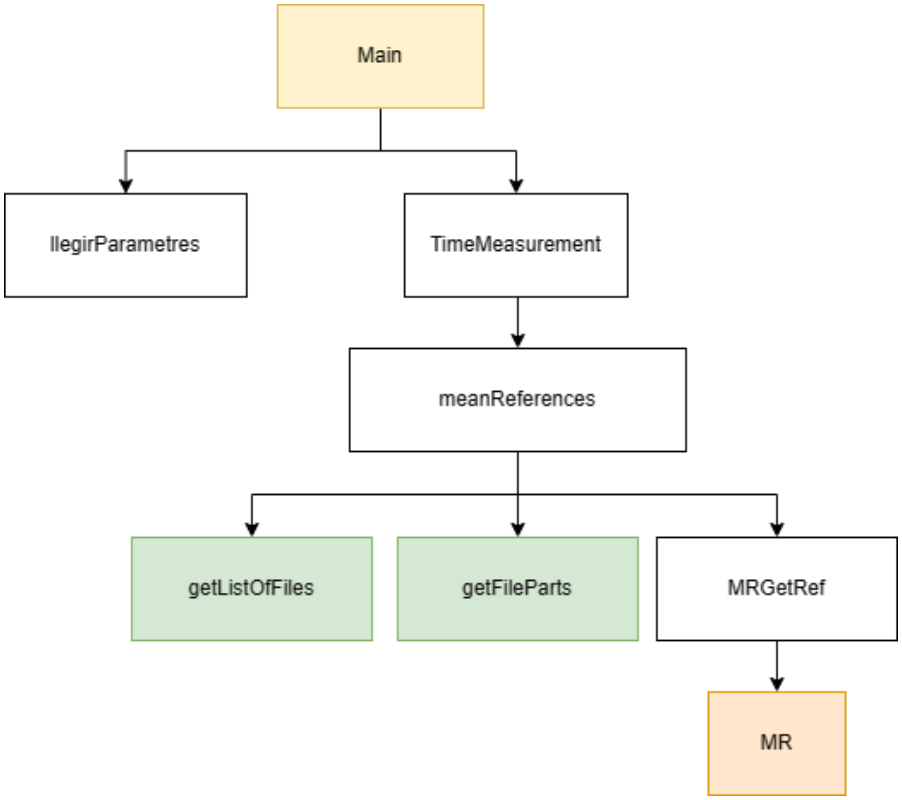

Figura 1: Flux d'execució de Calcular mitjana de referències

3.2 Calcular mitjana d'imatges

Aquesta part segueix un procediment similar a l'anterior, però aquesta vegada calculant la mitjana d'imatges. La part d'entrada de paràmetres i càlcul del temps és pràcticament el mateix, l'únic que canvia és que ara timeMeasurement crida a meanImages, que és la funció que ens interessa en aquest cas.

A meanImages, obtenim tots els fitxers que tenim i ho passem a MROnlyImages, juntament amb el nombre de mappers i de reducers. Allà agafem el contingut de cada fitxer i ho passem a MR en forma de tupla (contingut, llista buida), ja que en aquest cas no necessitem el paràmetre V1. També indiquem que cal utilitzar les funcions mappingMeanImg i reducingMeanImage, juntament amb la quantitat de mappers i reducers que cal fer servir.

Llavors, altra vegada a meanImages, rebrem el resultat obtingut, que serà el nombre d'imatges total, és a dir, l'acumulat entre tots els fitxers, i el dividirem entre el nombre de fitxers utilitzat per saber-ne la mitjana.

Les funcions de mappingMeanImg i reducingMeanImage es troben explicades **aquí**. El circuit d'execució que segueix la mitjana d'imatges és el que es pot veure a la següent figura:

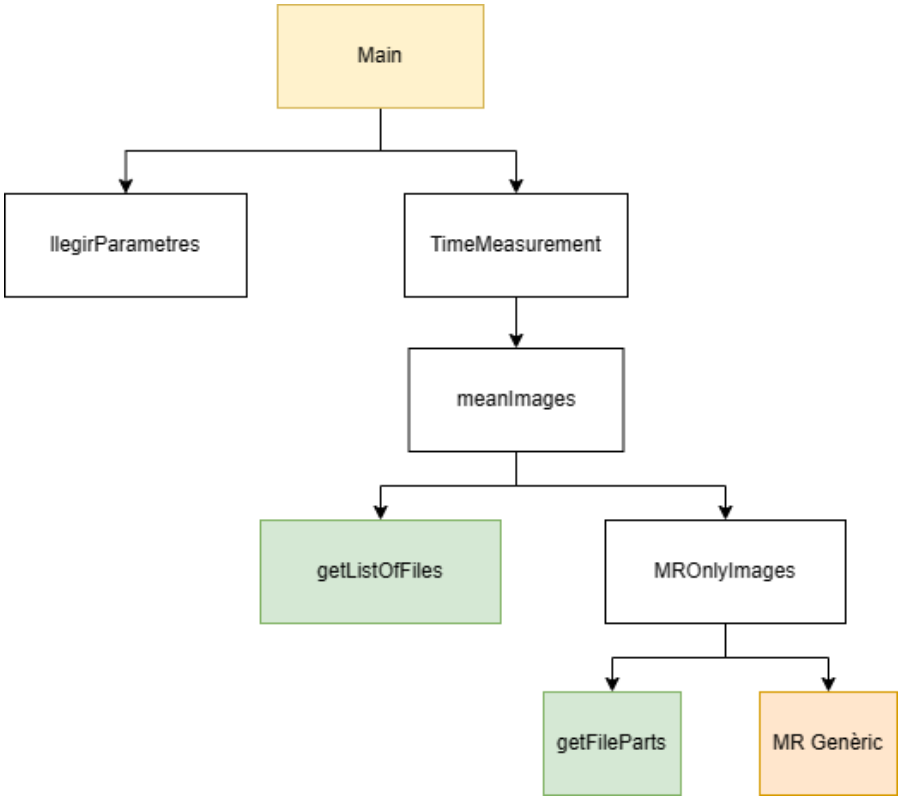

Figura 2: Flux d'execució de Calcular mitjana d'imatges

3.3 Pàgines més rellevants

En aquest cas, trobarem d'entre les N pàgines més rellevants, és a dir, les que són referenciades per altres pàgines més vegades. A l'hora de llegir els paràmetres se seguirà gairebé el mateix procediment que abans, amb l'única diferència que a banda del nombre de mappers i reducers a utilitzar, també caldrà indicar el nombre N de pàgines a mostrar.

Quan cridem paginesRellevantsGlobal, obtenim els fitxers que es troben a la carpeta indicada. Aquests fitxers i la quantitat de mappers i reducers a utilitzar els passem a MRGetRefGlobal, que ens donarà les vegades que cada pàgina ha sigut referenciada.

MRGetRefGlobal es queda amb una llista de tuples de títol i llista de referències de cada pàgina, i ho passa a MR, fent ús de mappingSD i reducingSD. Igual que al càlcul de la mitjana de referències, obtenim el Set paginesKeys, que són els títols de les pàgines presents al nostre corpus, i del resultat obtingut ens quedem només amb les pàgines en què el seu títol aparegui a paginesKeys.

Ara, a paginesRellevantsGlobal hi tenim el mapa de (títol, nombre de referències), el qual passem a llista i mostrem les N pàgines més referenciades.

Les funcions de mappingSD i reducingSD es troben explicades **aquí**. El circuit d'execució que segueix l'obtenció de les pàgines més rellevants és el que es pot veure a la següent figura:

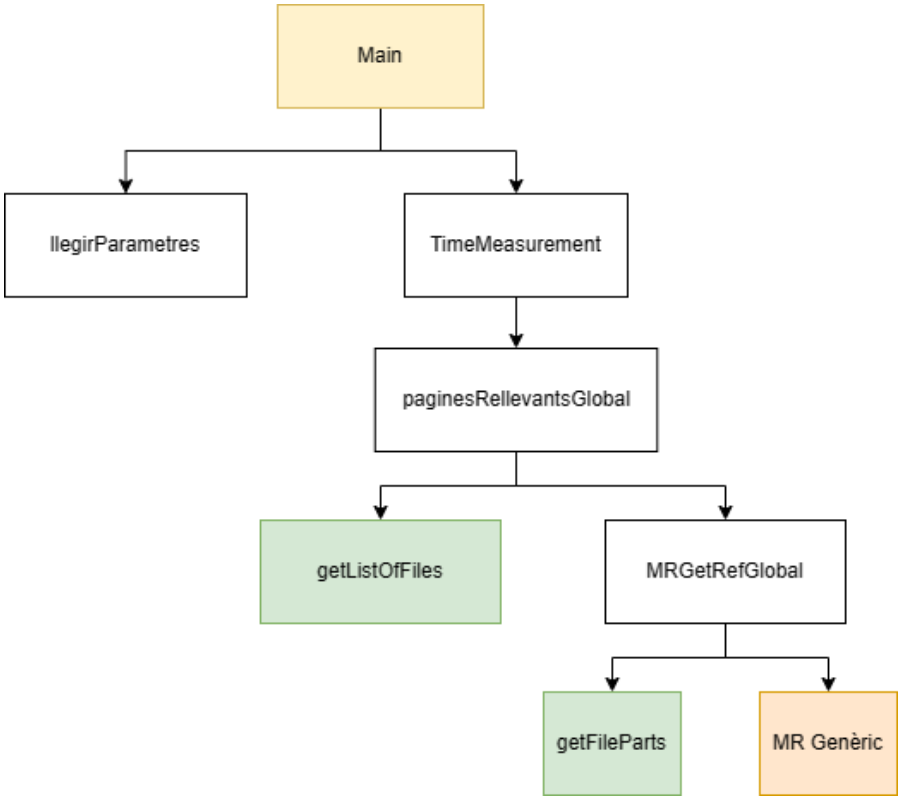

Figura 3: Flux d'execució de Pàgines més rellevants

3.4 Pàgines més similars

La quarta opció d'execució consisteix a trobar quines són les pàgines més similars entre elles i que no es referencien mútuament. Forma part del **punt 3.5**. En primer lloc, com la resta de funcions, llegim els paràmetres de teclat i cridem la funció timeMeasurement per tal de comptar el temps d'execució d'una funció. La funció executada aquesta vegada, serà similarNotRefGlobal. Com que necessitem passar-li més d'un paràmetre i el timeMeasurement polimòrfic només n'accepta un, li passarem una tupla amb tots els paràmetres dins.

SimilarNotRefGlobal extreu la llista de fitxers de la carpeta especificada per paràmetre, i escriu per pantalla els resultats d'executar TF_IDF.detectarSimilarsNoRefGlobal.

Aquesta funció llegeix els continguts dels fitxers i crea el diccionari que contindrà per cada nom de fitxer, un set amb les seves referències. Ho fa convertint la llista de paraules a mapa amb l'ajuda de groupBy i seguidament quedant-se amb només el Set de referències com a valor del mapa. D'aquesta manera tindrem un map amb clau: nom fitxer i valor: set referències. Seguidament, retorna la llista de pàgines més similars que obté de detectarPagines similars, aplicant una sèrie de funcions.

En primer lloc, converteix el mapa a llista amb ".toList". A continuació fa un filter per quedar-se únicament amb els que tenen una similitud superior a l'especificada i que no es referencien (MutuallyReferenced). Finalment els ordena de major a menor similitud.

La funció MutuallyReferenced, que serveix per comprovar si dues pagines es referencien mutuament, rep per paràmetre els noms de les dues pàgines i el diccionari de referències mutues. En primer lloc, busca dins el diccionari de referències el nom1. Si el troba en retorna el seu set de referències i, si no, un set buit. Si dins aquest set retornat hi trobem el nom2, sabem que nom1 referència nom2. Es fa la comprovació equivalent pel nom2 respecte nom1 i es retorna true només si tots dos retornen true (es referencien mutuament).

Finalment ens queda la funció detectarPaginesSimilars. En primer lloc calcula els índex IDF i TF de cada fitxer. L'índex IDF el calcula llançant el mapreduce explicat al punt **punt 5.1**. L'índex TF també el calculem llançant un MR, que es troba explicat en el **punt 5.2**.

A continuació llança el calculateVSMS, que crea els vector space models per posteriorment poder-los comparar. Seguidament, instanciem la variable comparisons, que serà les parelles de vsms que caldrà comparar. D'aquesta manera els podrem passar als mapreduces sense haver-nos de preocupar d'aparellar-los.

Per fer aquestes parelles hem utilitzat l'estructura for-yield. En aquest for, instanciem ï", que tindrà els valors de cada un dels índexs, i "j", que tindrà els valors de tots els índexs superiors a ïï menors de length. Així aconseguim no crear duplicats, ja que a efectes nostres és el mateix comparar (doc1 - doc2) que (doc2 - doc1). Per cada parella, fem un yield de les parelles de noms i les parelles de vsms, que posteriorment passarem al MR.

calculateVSMS rep una llista de tuples de títol i llista de paraules (és a dir, el contingut), juntament amb un mapa de (String, Double) que conté els pesos IDF de cada paraula en el corpus, i un altre mapa ((String, String), Int) que conté per cada fitxer i cada paraula, els pesos TF.

Per cada un dels documents de la wikipedia creem un vector de tuples (String, Double) anomenat vectorWeights on, per cada paraula dels documents, creem una tupla (String, Double) que conté la paraula i el seu pes TF_IDF. Cada tupla és afegida a vectorWeights. Una vegada hem calculat el vectorWeigths del fitxer, l'afegim a listVSMS. Retornem aquesta llista de, fitxers, amb els seus vectors de paraula - TF_IDF, que corresponen als Vector Space Models de cada fitxer.

Finalment cridem calculateCompareSpaceModels, que s'encarrega d'instanciar el mapreduce i rep per paràmetre la llista de comparacions. Tampoc explicarem aquest MapReduce ja que es troba explicat en el **punt 5.6**

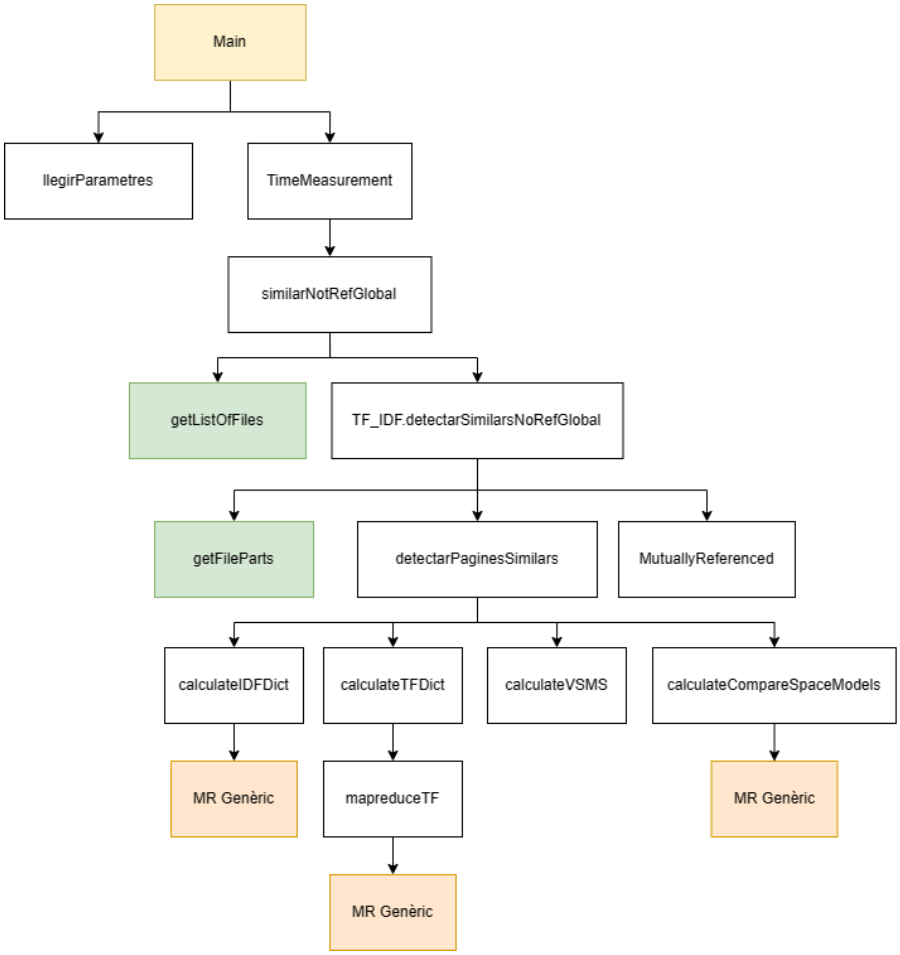

Figura 4: Flux d'execució de Pàgines més similars

3.5 Pàgines més rellevants més similars

L'execució de pàgines més rellevants més similars no correspon a un nou flux d'execució en si mateix sinó a una combinació dels dos anteriors. Vam pensar que també seria interessant d'oferir les opcions de trobar pàgines més similars i trobar pàgines més rellevants per separat.

En aquest circuit d'execució en primer lloc es llegeixen de teclat els diferents paràmetres que demana la funció, que corresponen a el nombre de pàgines, el llindar de similitud i els nombres de mappers i reducers. Seguidament crida la funció TimeMeasurement (com en totes les altres) per tal de mesurar el temps d'execució d'una funció en concret. Li passem la funció paginesMesRellevantsISimilars per tal que l'executi.

La funció paginesMesRellevantsISimilars s'encarrega en primer lloc d'obtenir les tuples que corresponen al contingut, nom i referències dels fitxers de la carpeta rebuda per paràmetre. En segon lloc fa una crida a la funcio paginesRellevants, passant el nom i llista de referències de cada pàgina, a més dels paràmetres que ens ha entrat l'usuari a l'inici. D'aquesta crida rep les pàgines més rellevants i n'obté el contingut (en aquest cas listWords), i el passa a la funcio similarNotRef, que s'encarregarà de trobar les

pàgines més similars que no es referencien. Abans de passar-lo, però, converteix el contingut a una llista de paraules i n'extreu els stopWords.

Per tal de convertir el contingut d'un text a llista de paraules el que hem fet és cridar la funció normalitza i seguidament la funció strToNonStopList que hem explicat en el punt anterior.

La funció paginesRellevants crida MRGetRef, que instancia el MapReduce que calcula el nombre de referències que té cada pàgina, i mostra les pàgines que han rebut més referències. Retorna únicament el seu nom. La funció MRGetRef està explicada en el **punt 3.1** ja que és exactament igual.

La funció similarNotRef simplement mostra els resultats de cridar la funció detectarSimilarsNotRef. Aquesta funció al seu torn, genera un mapa de pagina - set de referències, que permetrà trobar si dues pàgines es referencien mutuament o no. Això ho fa fent un groupBy (ja que ens retorna un diccionari), i posteriorment quedant-nos només amb la llista der referències d'entre els valors i convertint-la a set per fer cerques més eficients.

Finalment retorna el contingut d'executar la funció detectarPaginesSimilars amb els següents paràmetres: la llista de noms de fitxers amb el seu contingut per comprovar similitud, el nombre de mappers i el nombre de reducers. Abans de retornar aquesta funció, es filtra les pagines més similars quedant-nos únicament amb les que tenen un llindar superior a l'establert i amb les que no es referencien mutuament (**Explicació MutuallyReferenced**).

La funció detectarPaginesSimilars es troba explicada en el **punt 3.4**.

El circuit d'execució d'aquest punt és el que podem veure a la figura a continuació. Només s'han marcat les funcions del flux principal (per exemple getFileParts i similars no s'han posat). El flux és de dalt a baix i d'esquerra a dreta.

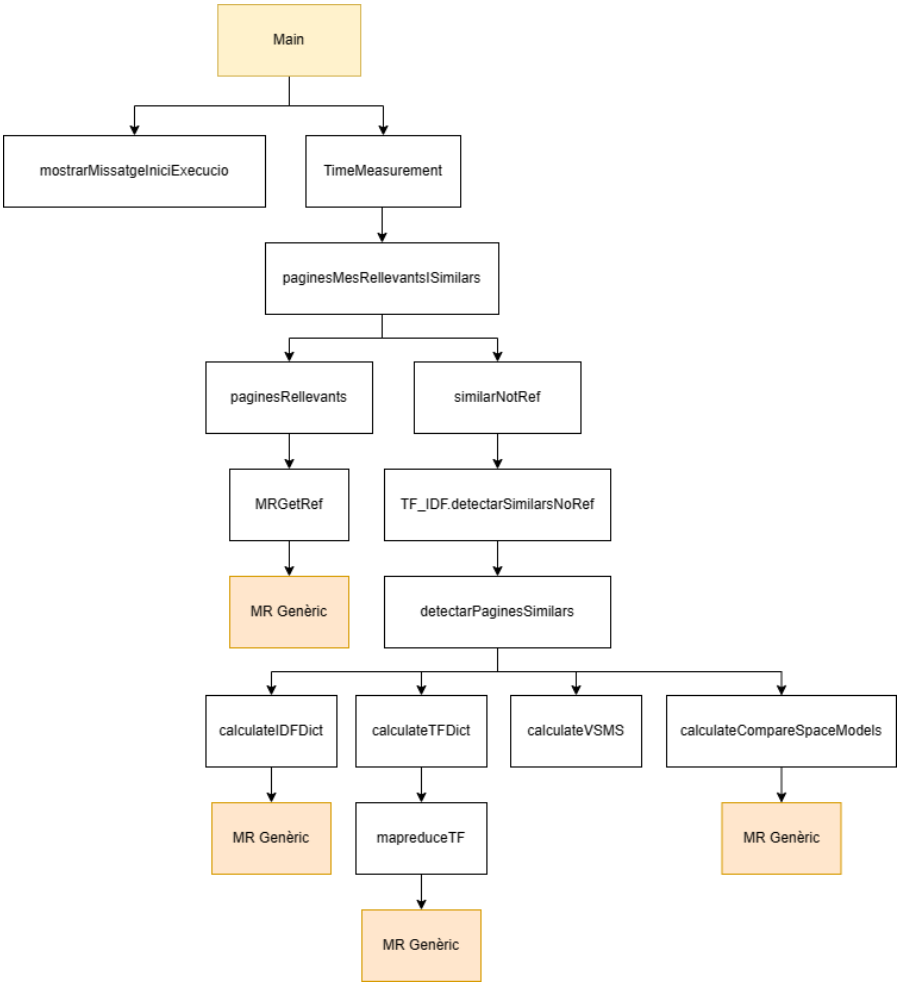

Figura 5: Flux d'execució de Pàgines més rellevants més similars

4 Principals funcions d'ordre superior usades

En aquest apartat ens centrarem a mostrar alguns apartats del nostre treball on utilitzem funcions d'ordre superior i creiem que són rellevants i/o interessants de mostrar.

4.1 Map, Filter i Foreach

Per comparar dos VSMs inicialment havíem pres una aproximació, però l'hem canviat quan ens hem adonat a la segona part que el temps quedava reduït a la meitat.

El primer codi que havíem fet és el següent:

```
def compareVectorSpaceModel(vsm1: Vector[(String, Double)], vsm2: Vector
    [(String, Double)]): Double = {
    val ngramIn1 = vsm1.map { case (ngram, _) => ngram }.toSet
    val notIn1 = vsm2.filter(x => !(ngramIn1.contains(x._1)))

    val ngramIn2 = vsm2.map { case (ngram, _) => ngram }.toSet
    val notIn2 = vsm1.filter(x => !(ngramIn2.contains(x._1)))

    var mutable_vsm1: Vector[(String, Double)] = vsm1
    var mutable_vsm2: Vector[(String, Double)] = vsm2

    notIn1.foreach(ngram => {
      val tuple: (String, Double) = (ngram._1, 0)
      mutable_vsm1 :+= tuple
    })
    notIn2.foreach(ngram => {
      val tuple: (String, Double) = (ngram._1, 0)
      mutable_vsm2 :+= tuple
    })

    // Sort both vectors by n-gram to ensure they are in the same order
    mutable_vsm1 = mutable_vsm1.sortBy(_._1)
    mutable_vsm2 = mutable_vsm2.sortBy(_._1)

    val component1 = Math.sqrt(mutable_vsm1.map(x => x._2 * x._2).sum)
    val component2 = Math.sqrt(mutable_vsm2.map(x => x._2 * x._2).sum)

    val sum = mutable_vsm1.zip(mutable_vsm2).map { case (ngram1, ngram2)
    =>
      ngram1._2 * ngram2._2
    }.sum

    sum / (component1 * component2)
}
```

Aquest codi compara els dos vectors que rep per paràmetre i en retorna el seu índex de similitud. En primer lloc crea dos nous vsms mutables i extreu de cada un dels vsms les paraules que no es troben en l'altre per afegir-los amb pes 0. Finalment, ordena la llista per assegurar que en fer el zip cada paraula anirà amb el seu equivalent en l'altre vsm i en fa la multiplicació i posterior suma de totes les multiplicacions.

A part també calcula l'arrel quadrada del sumatori dels quadrats de tots els elements de cada vsm. Això ens permet obtenir l'índex de similitud dividint el producte escalar, per la multiplicació de l'arrel quadrada de la suma dels quadrats de tots els elements dels dos vsms.

Les funcions d'ordre superior que utilitzem, en ordre d'aparició, s'expliquen a continuació.

La funció map ens permet aplicar a cada element d'una llista una funció. Nosaltres la utilitzem per fer diverses funcions. En la primera ens permet que el vsm deixi de ser de parelles i passi a contenir únicament el primer element de la parella. Ho fem emprant case. També el fem servir a la línia 24 i 25 per convertir el mutable_vsm2 de vector de parelles: paraula - pes a vector d'enters: pes al quadrat. Ho fem usant una funció anònima que rep una tupla i en retorna el quadrat del segon element.

També en fem un ús molt similar a la línia 27 on ens permet aplicar al zip dels dos mutable vectors una funció que retorna la multiplicació dels segons elements dels dos integrants de la tupla que contindrà a cada posició.

La funció filter permet fer un filtratge dels elements d'una llista en funció de si compleixen una funció booleana. L'utilitzem a les línies 3 i 6 per quedar-nos únicament amb els elements d'un vector que no apareixen en l'altre.

La funció foreach permet iterar a través d'una llista i fer una operació o executar una funció a cada element d'aquesta llista. Nosaltres l'hem utilitzat per anar afegint als respectius mutable vsms els elements que no hi pertanyien amb pes 0.

Finalment, el codi amb què ens hem quedat després de comprovar com de dolent era és el que podrem veure en el següent apartat.

Altres llocs on hem utilitzat aquestes funcions a la segona part són:

```
1  /**
2  * Returns the MapReduce results of finding the number of references for
       each page file in the input.
3  * @param nfitxers
4  * @return
5  */
6  def MRGetRefGlobal(lf: List[File], nmappers: Int, nreducers: Int): Map[
       String, Int] = {
7      val folder="viqui_files"
8
9      val pagines = lf.map(x => ProcessListStrings.getFileParts(folder + "/
       " + x.getName)).map(x => (x._1,x._3));
10     val result = MR(pagines, MappersAndReducers.mappingSD,
       MappersAndReducers.reducingSD, nmappers, nreducers)
11     val paginesKeys: Set[String] = pagines.map(_._1).toSet
12     result.filter(x => paginesKeys.contains(x._1))
13 }
```

Aquesta funció és la que executa el MapReduce per trobar el nombre de referències a cada pàgina a la llista que rep com a input. Primer fem un map sobre la llista de fitxers, en què, per cada fitxer x, apliquem la funció de getFileParts amb el nom del fitxer en qüestió (x.getName). Això ens retorna una llista de la mateixa mida que la llista de fitxers, on cada posició és la tripleta de títol, contingut i llista de referències de cada fitxer. A aquesta llista resultant, li apliquem un altre map que ens retorna una llista de tuples corresponents a només el títol i la llista de referències de cada fitxer. Aquest resultat és el que passem al MapReducer com a input.

També fem un altre map per generar paginesKeys, que és un Set que conté només els títols de les pàgines. Això ho fem amb un map molt senzill que simplement es queda amb el primer component de cada tupla de la llista pagines, i al resultat li apliquem un toSet.

Finalment, a result, que és un Map[String, Int] li apliquem un filter on, per cada entrada, mirem si aquesta es troba continguda a paginesKeys. Per tant, al resultat només hi haurà els elements de result que formin part de la llista de fitxers inicials.

Un altre tros de codi on podem veure'ls utilitzats:

```
/**
* Returns the pages more similar than similaritat that do not reference each other
* @param lf          List of files that contains the pages to consider
* @param quantes     How many pages of lf do we want to use
* @param stopWords   List of stopwords to not consider when doing the similarity assessment
* @param similaritat Similarity Threshold to consider pages similar enough
*/
def detectarSimilarsNoRefGlobal(lf: List[File], quantes: Int, stopWords: List[String], similaritat: Double,
                                nmappers: Int, nreducers: Int): List[((String, String), Double)] = {
    /** Folder to get files from */
    val folder = "viqui_files"

    /** Contains triplets of (Name, Words in content, References) */
    val listWords = lf.take(quantes).map(x => ProcessListStrings.getFileParts(folder + "/" + x.getName)).map(x => (x._1, SimilitudEntreDocuments.strToNonStopList(SimilitudEntreDocuments.normalitza(x._2), stopWords), x._3))
    /** Contains a dictionary of (PageName => Set of references) */
    val dictRefs: Map[String, Set[String]] = listWords.groupBy(_._1).view.mapValues(_.head._3.toSet).toMap

    /** Contains a map of (PageName1, PageName2) => Similarity */
    detectarPaginesSimilars(listWords.map(x => (x._1, x._2)), nmappers, nreducers).toList.filter(x => x._2 >= similaritat && !MutuallyReferenced(x._1._1, x._1._2, dictRefs)).toList.sortBy(_._2)(Ordering.Double.IeeeOrdering.reverse)
}
```

En aquest codi tornem a fer un map amb getFileParts com abans, però després, al resultat li apliquem un map diferent. Retornem una llista amb tantes tripletes com la llista on apliquem el map, on el primer i tercer element de la tripleta es mantenen, però el segon és la llista de paraules del contingut, després d'haver-ne tret referències i altres elements que no hi hagin de ser (SimilitudEntreDocuments.normalitza(x._2)), i d'aquest resultat, traient-ne els stopwords (SimilitudEntreDocuments.strToNonStopList(SimilitudEntreDocuments.normalitza(x._2), stopWords)). El resultat el guardem a listWords, que és una List[(String, List[String], List[String])].

Després, volíem un diccionari que ens digués, per cada pàgina, les referències que té. Per tant, generem un mapa fent un groupBy agrupant pel primer element de cada tupla, és a dir, el títol. En acabat, amb el mapValues, que s'aplica només sobre els valors de les claus, ens quedem només amb la tercera part de la tripleta, que com hem dit abans és la llista de referències, convertida a Set per eliminar duplicats.

Per retornar el resultat, cridem detectarPaginesSimilars(listWords.map(x =¿ (x._1, x._2)), nmappers, nreducers) on, com podem veure, mitjançant un altre map li passem una llista que conté tuples amb només el primer i segon element de les tripletes de listWords. El que obtenim ho passem a llista, i li apliquem un filter en què ens quedem només amb els elements que siguin més similars que el llindar de similitud i que no es referenciïn mútuament. Finalment, només cal ordenar el que hem generat.

Només per acabar d'explicar unes últimes funcions que són rellevants en la nostra execució:

```
/**
```

```
* Prints the mean references: The mean number of references each page in
    the folder has.
* @param folder
*/
def meanReferences(t : (String, Int, Int)): Unit = {
    val lf: List[File] = ProcessListStrings.getListOfFiles(t._1);

    // Change if you want less files to be used.
    val nFitxers = lf.length
    val pagines = lf.take(nFitxers).map(x => ProcessListStrings.
    getFileParts(t._1 + "/" + x.getName)).map(x => (x._1, x._3));
    val resultat = MRGetRef(pagines, t._2, t._3);

    println("La mitjana de referencies a cada pagina es " + resultat.
    values.sum.toDouble / nFitxers)
}
```

Aquesta funció és prou simple. Només s'encarrega d'escriure per pantalla la mitjana de referències entre totes les pàgines. Primer obtenim el títol, contingut i llista de referències de cada fitxer com abans, i fent un map ens quedem amb només els títols i les llistes de referències.

Llavors, només cal passar el que hem generat al mapper i, un cop tenim el resultat, sumem les referències que hem trobat a cada fitxer i ho dividim entre nFitxers per saber quina és la mitjana.

```
/**
* Executes a MapReduce that computes the number of images every file in
    lf has.
* @param lf
* @return
*/
def MROnlyImages(lf: List[File], nmappers: Int, nreducers: Int): Map[
    String, Int] = {
    val folder = "viqui_files"
    val nFitxers = lf.length
    val pagines = lf.take(nFitxers).map(x => ProcessListStrings.
    getFileParts(folder + "/" + x.getName));

    MR(pagines.map(x => (x, List())), MappersAndReducers.mappingMeanImg,
    MappersAndReducers.reducingMeanImage, nmappers, nreducers)
}
```

Per calcular el nombre d'imatges de cada pàgina hem assumit que qualsevol part del contingut del fitxer que sigui del tipus "[[Fitxer:"és una imatge.

A pagines hi guardem el resultat d'aplicar un map a la llista de fitxers tal com hem fet abans, generant una llista de tripletes amb títol, contingut i llista de referències.

Llavors, simplement cridem el MapReduce generant l'input amb un map aplicat sobre pagines, que genera una tupla que conté com a primer element una llista amb totes les tripletes de pagines, i com a segon element una llista buida que actua com a paràmetre "Dummy". El resultat d'executar aquest MapReduce és el que retornem.

4.2 FoldI

```
/**
* Compares two vectorspacemodels
* @param vsm1 Vector of [(Word, Weight)]
* @param vsm2 Vector of [(Word, Weight)]
* @return Double: Similarity
```

```
*/
def compareVectorSpaceModel(vsm1: Vector[(String, Double)], vsm2: Vector
    [(String, Double)]): Double = {

    val component1 = Math.sqrt(vsm1.map(x => x._2 * x._2).sum)
    val component2 = Math.sqrt(vsm2.map(x => x._2 * x._2).sum)

    val map1 = vsm1.toMap

    /**
     * We will do a fold left so that we iterate over vsm2 and we
    accumulate the multiplication of each element of vsm2 that we can
    find in vsm1.
     * If we cannot find it as the multiplication would return 0 we
    simply return the accumulated value.
     */
    val scalarProduct = vsm2.foldLeft(0.0) { (acm, word) =>
      map1.get(word._1) match {
        case Some(weight) => acm + (weight * word._2)
        case None => acm
      }
    }

    //val scalarProduct = vsm2.map(x => x._2*map1.getOrElse(x._1,0.0)).
    sum

    scalarProduct / (component1 * component2)
}
```

Aquesta implementació pren una aproximació diferent del problema que l'altre. En comptes d'intentar que tots els elements es trobin a les dues llistes el que fa és simplement ignorar els elements que no es troben en una de les dues llistes. Sabem que podem fer això perquè en fer el producte escalar totes les paraules que no es trobin en un dels dos vectors tindran pes 0 en el vector on no es troben, i per tant no tindran cap impacte en el càlcul. Aquest codi és molt més senzill i no utilitza tantes funcions d'ordre superior, però si que n'utilitza una de nova, el foldLeft.

El foldLeft és una funció que permet reduir una llista utilitzant un acumulador i una funció, que s'aplicarà a cada element de la llista d'esquerra a dreta. La funció pren el valor de l'acumulador en cada pas. En aquest cas el que fem és recórrer el vector 2. Inicialment l'acumulador valdrà 0, ja que és l'element neutre de la suma. A cada pas comprovarem si la paraula que el vsm2 té en aquella posició es troba també a vsm1. Si no és així l'ignorarem, retornant l'acumulador que ens venia del pas anterior. Si trobem l'element a vsm1, afegim a l'acumulador la multiplicació del pes que té la paraula a vsm2 i el que té a vsm1. D'aquesta manera quan arribem al final, scalarProduct contindrà la suma de tots els productes dels elements que es troben a les dues llistes, és a dir, el producte escalar.

4.3 GroupBy

```
def freqNGrams(str: String, n: Int): List[(String, Int)] = {

    val listNGrams = strToList(str).sliding(n).toList
    val ngrams = listNGrams.map(ngram => ngram.mkString(" "))

    val listAppearances = ngrams.groupBy(identity).view.mapValues(_.size)
    .toList.sortBy(_._2)(Ordering.Int.reverse)

    listAppearances
```

```
}
```

$def groupBy[K](f : A => K) : Map[K, List[A]]$

La funció groupBy serveix per agrupar els elements d'una col·lecció segons una funció, i obtenir un Mapa clau-valor, on les claus seran els valors únics determinats per la funció i els valors la llista d'elements de la col·lecció que comparteixen la mateixa clau. En aquest cas utilitzem la funció groupBy per crear una nova llista de parelles paraula - nombre d'aparences. Donant a groupBy la funció identity ens retorna un mapa on el primer element serà cada una de les paraules i el segon serà una llista que contindrà tantes vegades la paraula com es trobés a la llista inicial.

La funció mapValues llavors ens permet aplicar una funció a tots els valors del mapa resultant, el que utilitzem per substituir totes les llistes que ocupaven la posició valor del mapa per la mida d'aquestes llistes, és a dir el nombre d'ocurrences de la paraula clau en aquella posició.

5 Quins Map Reduce hem implementat?

Al llarg d'aquesta pràctica hem implementat nombrosos MapReduce, que ens han servit per paral·lelitzar gran part del codi i reduir el temps el màxim que hem pogut. Tot i això, hi ha hagut un cas on hem implementat un MapReduce i no només no ens ha millorat el temps sinó que ens l'ha empitjorat considerablement.

5.1 Càlcul de l'índex IDF

En el nostre procés per aconseguir el resultat del tf_idf, en primer lloc necessitàvem fer el càlcul de l'índex IDF, que s'expressa com a $idf = \log \frac{|C|}{df}$. Per aconseguir-ho vam veure molt clar que ens seria útil un mapReduce.

L'objectiu del mapReduce serà fer el càlcul de df, que és el nombre de pàgines on apareix el terme sobre el que estem fent el mapReduce. Llavors els mappers i reducers construïts són els següents:

```
/**
* Mapper to calculate the IDF.
* @param pagina Name of page
* @param words  List of strings in page
* @return List of (Word, 1)
*/
def mappingIDF(pagina: String, words: List[String]): List[(String, Int)]
    = {
    (for (w: String <- words.toSet) yield (w, 1)).toList
}

/**
* Reducer to calculate the IDF
* @param word Name of page
* @param nums List of 1s
* @return Returns the pair of (Word, Sum of nums)
*/
def reducingIDF(word: String, nums: List[Int]): (String, Int) = {
    (word, nums.sum)
}
```

L'objectiu d'aquest mapper serà retornar una llista de parelles paraula - 1, per tal que el reducer després només hagi de sumar el nombre de vegades que apareix cada paraula. Llavors el diccionari convertirà les parelles (paraula, 1) a parelles (Paraula List[1s]), i el reducer només haurà de comptar quants elements té la llista per saber el nombre de pàgines que referencien aquella paraula.

Un detall important d'aquest mapper és el words.toSet, que assegura que les paraules només apareixeran una vegada dins words, és a dir que només comptarem una vegada l'aparició d'una paraula en una pàgina. Així doncs, els tipus seran els següents:

- K1: String: Nom d'una pàgina.
- V1: String: Paraula.
- K2: String: Paraula.
- V2: Int: 1.
- V3: Int: Nombre de vegades que apareix una paraula

5.2 Càlcul de l'índex TF

En el nostre procés per aconseguir el resultat del tf_idf, en segon lloc necessitàvem fer el càlcul de l'índex TF, que consisteix en el nombre de vegades que apareix un terme, és a dir la seva freqüència absoluta.

L'objectiu del mapReduce serà fer el càlcul complet del TF. Llavors els mappers i reducers construïts són els següents:

```
/**
* Mapper to calculate TF of every word in page
* @param pagina page of words
* @param words  list of words in pagina
* @return List of ((PageName, Word), 1)
*/
def mappingTF(pagina: String, words: List[String]): List[((String, String), Int)] = {
    for (w: String <- words) yield ((pagina, w), 1)
}

/**
* Gets a list of (page, word),List(1,1,1,1,1...)
* @param t
* @param nums
* @return for each page and word the number of occurrences
*/
def reducingTF(t: (String, String), nums: List[Int]): ((String, String), Int) = {
    (t, nums.sum)
}
```

El mapper rebrà una llista de paraules d'una de les pàgines i retornarà una llista de tuples (pàgina, paraula), 1. Posteriorment, totes les parelles es posaran en un diccionari on les claus seran (pàgina, paraula) i els valors una llista de 1s que serà de la mida del nombre de vegades que apareix paraula a pàgina. El reducer només haurà de sumar les paraules. L'output del MapReduce, doncs, seran parelles ((pàgina, paraula), freqüència_absoluta). Així, els tipus seran els següents:

- K1: String: Nom d'una pàgina.
- V1: String: Paraula.
- K2: (String,String): (Pagina, Paraula).
- V2: Int: 1.
- V3: Int: Nombre de vegades que apareix una paraula en aquella pàgina

5.3 Càlcul de la mitjana d'imatges per pàgina

Per obtenir la mitjana d'imatges per pàgina hem fet un MapReduce més complex que els casos que hem vist fins ara. Ens interessava que el tipus de retorn d'aquest MapReduce fos un mapa on tinguéssim el nombre d'imatges a cada pàgina, i que, per tant, per trobar la mitjana haguéssim de fer una simple suma i divisió.

Per fer aquesta feina al principi havíem fet una aproximació on preparàvem les dades pel MapReduce, però posteriorment ens vam adonar que deixar que fos cada Mapper el que es preparés les seves dades és molt més eficient i redueix enormement el temps.

El codi que hem utilitzat és el següent:

```
/**
* @param pagina Title, Content and Referenes of a page
* @param dummy_content Dummy parameter
* @return
*/
def mappingMeanImg(pagina: (String, String, List[String]), dummy_content:
     List[String]): List[(String, Int)] = {
    for (pag <- pagina._2.split(" ").toList; if pag.contains("[[Fitxer:")
    ) yield (pagina._1, 1)
}

/**
*
* @param pagina
* @param nums
* @return
*/
def reducingMeanImage(pagina: String, nums: List[Int]): (String, Int) = {
    (pagina, nums.sum)
}
```

En comptes de rebre directament la llista de paraules, el mapper rebrà una tripleta i una llista dummy. Aquesta tripleta contindrà tot el que fa referència a una pàgina, és a dir el nom, el seu contingut com a String i la llista de les seves referències. El for del mapping està lleugerament modificat respecte a un mapper clàssic. En primer lloc, separa l'String pels seus espais i ho converteix a llista. Això permetrà que totes les referències a fitxers, es puguin comptar simplement comptant el nombre de paraules que contenen "[[Fitxer", que és la marca d'inici de Fitxer.

El que fa el mapper és separar el text que rep per paràmetre, i retorna una llista de tuples on el primer element és el nom de la pàgina en qüestió i el segon és un 1. Després de convertir-se en diccionari, el primer element de la llista serà el nom d'una pàgina i el segon una llista d'1s amb tants elements com imatges tingui una pàgina.

El reducer només haurà de sumar els segons elements de la tupla i el mapReduce retornarà un Mapa on la clau serà el nom de la pàgina i el valor el nombre d'imatges que té aquesta pàgina.

Així doncs, els tipus seran els següents:

- K1: (String, String, List[String]): (Nom, Text, List Referencies).
- V1: String: Dummy, inùtil.
- K2: String: Pàgina.
- V2: Int: 1.
- V3: Int: Nombre d'imatges que té una pàgina

5.4 Càlcul del nombre de referències que té cada pàgina

El primer lloc on vam veure clar que calia aplicar un mapReduce, probablement no necessàriament perquè fos el més evident sinó perquè sense aplicar-lo tardava molt temps és en el càlcul del nombre de referències que té cada pàgina. Ens interessava poder enviar una llista amb totes les pàgines i les pàgines a qui feien referència i recuperar el nombre de vegades que es feia referència a cada una de les pàgines.

Podríem haver passat directament una referència a la llista de pàgines que volem tenir en compte i comprovar des de dins del MapReduce si cada pàgina hi pertany, però vam comprovar que el temps era molt similar a filtrar la llista una vegada acabat el MapReduce, així que ho vam deixar com ho teníem. El resultat del MapReduce, doncs, és un map amb totes les pàgines a què es fa referència des d'alguna de les pàgines que tenim en compte, i el nombre de vegades que s'hi fa referència.

Hem decidit comptar el nombre de vegades que s'hi fa referència, no el nombre de pàgines que hi fan referència per dues raons. La primera és que no podem tractar igual una pàgina que és referenciada per dues pàgines una sola vegada que una que és referenciada 20 cops per cada una. La segona és que ens facilita enormement el càlcul i ens redueix el temps de computació no haver de comprovar si hem comptat o no cada referència.

El codi que hem utilitzat és el següent:

```
/**
*
* @param pagina Title of the page
* @param referencies List of references in the page
* @return
*/
def mappingSD(pagina: String, referencies: List[String]): List[(String,
    Int)] = {
    // We clean all references at once because we have to delete some of
    them.
    for (ref <- referencies; if !references.shouldDeleteReference(ref))
    yield (references.cleanReference(ref), 1)
}

/**
*
* @param pagina
* @param nums
* @return
*/
def reducingSD(pagina: String, nums: List[Int]): (String, Int) = {
    (pagina, nums.sum)
}
```

Aquest cas és molt simple: El mapper rep el nom d'una pàgina i la llista de referències d'aquesta pàgina. Per motius d'eficiència és el mateix mapper el que neteja les referències. D'aquesta manera només hem de recórrer una vegada la llista de referències. En primer lloc comprova si una referència ha de ser comptada (pot no haver ser comptada degut a 3 raons: Que contingui "#", que vol dir que és una referència a una secció d'una pàgina, que comenci amb "[[MG", que voldrà dir que és una pàgina no definida o bé que comenci per "[[Fitxer", que voldrà dir que el que es referencia és un Fitxer. Seguidament neteja la referència, eliminant els possibles espais al principi i final, traient "[[]]"de principi i final i eliminant el text de les referències (Que apareix després de ").

La llista retornada es posarà dins un diccionari i el mapper només haurà de comptar per cada pàgina quantes vegades hem fet yield 1 per saber el nombre de referències que es fa a una pàgina en concret.

Així doncs, els tipus seran els següents:

- K1: String: Nom de pàgina.
- V1: String: Referència a una pàgina.

- K2: String: Pàgina a la que referenciem.
- V2: Int: 1.
- V3: Int: Nombre de referències que té una pàgina

5.5 Càlcul de la mitjana de referències que es fa a cada pàgina

Aquest és el primer dels casos on hem vist que no sortia a compte fer un MapReduce a mida sinó que utilitzant un MapReduce que havíem implementat anteriorment ens anava millor. El MapReduce que hem fet a mida funcionava similarment a l'anterior, però en comptes de fer yield de cada pàgina feia un yield amb una única clau, el que feia que s'ajuntés tot en un únic diccionari i que fos un únic reducer l'encarregat de sumar tots els elements de la llista per trobar el nombre global de referències.

Segons aquesta primera interpretació del problema, calculàvem la mitja de referències que feia cada pàgina a altres pàgines entre les pàgines que tenim en compte. Després, però vam decidir que la interpretació que tenia més sentit era calcular el nombre mitjà de referències que es feien a cada una de les pàgines que tenim en compte. Per aquesta raó el que hem acabat fent és, en primer lloc, calcular el nombre de referències que es fan a cada pàgina i posteriorment sumar-ho i dividir-ho entre el nombre total de pàgines.

El codi que hem utilitzat per a la primera interpretació és el següent:

```
/**
* @param pagina Tuple containing the name of the current page and the
    list of all pages to be able to distinguish if the reference is in
    our set of pages.
* @param referencies
* @return
*/
def mappingMR(pagina: (String, List[String]), referencies: List[String]):
     List[(String, Int)] = {
// We clean all references at once because we have to delete some of them
    .
    for (ref <- referencies; if !references.shouldDeleteReference(ref) &&
     pagina._2.contains(references.cleanReference(ref))) yield ("total",
    1)
}

/**
*
* @param pagina
* @param nums
* @return
*/
def reducingMR(pagina: String, nums: List[Int]): (String, Int) = {
    (pagina, nums.sum)
}
```

Com es pot veure és molt semblant al punt anterior amb dues petites diferències. La primera és que el mapper rep la llista de totes les pàgines que tenim en compte, per poder descartar les referències a pàgines que no pertanyen al conjunt amb què estem treballant. La segona diferència és que en comptes de fer output del nom de la pàgina a què fem referència fem output d'una única clau, que és "total". Això farà que la funció reducing s'executi una única vegada i que el Map resultant d'aquest MR només contingui un element, amb clau "totalï valor el nombre total de referències que fan les

pàgines que hem tingut en compte a altres pàgines del mateix set. Llavors només hem de dividir pel total de pàgines i ja tenim el resultat.

Així doncs, els tipus seran els següents:

- K1: (String, List[String]): (Nom de pàgina, List Pagines a tenir en compte).
- V1: String: Referència a una pàgina.
- K2: String: 'total'.
- V2: Int: 1.
- V3: Int: Nombre de referències que tenen totes les pàgines a pàgines dins el conjunt actual.

5.6 Comparació de Vector Space Models

Des del moment en què vam fer la comparació de la manera que es pot veure en el primer apartat del punt 3.1 ens vam adonar que el MapReduce ens aniria molt bé per comprar Vector Space Models. La nostra aproximació intenta paral·lelitzar la comparació de VSMS fent que siguin diferents mappers els que s'encarreguin de comparar parelles de VSMS.

Per tant, abans de cridar el MapReduce creem totes les parelles que necessitem comparar, i les enviem perquè sigui l'estructura MapReduce la que decideixi com repartir-les per reduir el temps de còmput.

El codi que hem utilitzat és el següent:

```
/**
*
* @param names Titles of each page
* @param vsms Vector Space Models for each word in each page
* @return
*/
def mappingCVSM(names: (String, String), vsms:  List[(Vector[(String,
    Double)], Vector[(String, Double)])]): List[((String,String), Double)
    ] = {
    for (vsm <- vsms) yield (names, SimilitudEntreDocuments.
    compareVectorSpaceModel(vsm._1,vsm._2))
}

/**
* There will only be one element in weights as each pair is unique.
* @param pages
* @param weights
*/
def reducingCVSM(pages: (String,String), weights: List[Double]): ((String
    ,String), Double) = {
    (pages, weights.sum)
}
```

El mapper rep una parella de vsms, per un costat els noms i per l'altre els vsms, i per cada parella que rep (en cada cas rebrà només una única parella) farà yield del nom de la parella i el seu índex de similitud. No és exactament un MapReduce clàssic, però era molt important paral·lelitzar aquesta part i hem vist que ens millora molt el temps utilitzant aquest MapReduce. Per aquesta raó, les parelles seran úniques i el reducer contindrà només un únic element dins weights, que en fer sum quedarà igual.

Obtindrem d'aquest MapReduce un map que contindrà totes les parelles que hem passat i el seu pes calculat.

Així doncs, els tipus seran els següents:

- K1: (String, String): (Nom pàgina 1, Nom pàgina 2)
- V1: (Vector[(String, Double)], Vector[(String, Double)]): (VSM1, VSM2)
- K2: (String,String): (Nom pàgina 1, Nom pàgina 2).
- V2: Double: Índex de similitud del parell de pàgines.
- V3: Double: Índex de similitud de cada parell de pàgines.

5.7 Producte Escalar de dos VSMS

L'últim MapReduce implementat que ens falta per explicar és el del Producte Escalar. Aquest és un molt bon exemple de com la paral·lelització d'operacions massa simples pot resultar en un augment del temps a causa dels *delays* de comunicació entre Actors. Precisament és això el que ens va passar.

Quan fèiem el còmput del producte escalar dels dos vectors en la comparació d'aquests, se'ns va acudir que podria ser molt bona idea paral·lelitzar el MapReduce, ja que sembla una aplicació bastant natural d'aquesta estructura, pensant que ens podria reduir el temps de còmput.

Els mappers i reducers construïts van ser els següents:

```
/**
* The mapping makes the multiplications and returns always the same word.
* That will make the dictionary put all the multiplications in the same
    list.
* @param word
* @param vsm2
* @return
*/
def mappingScalarProduct(word: (String, Double), vsm2: List[(String,
    Double)]): List[(String, Double)] = {
    val result = vsm2.find(_._1 == word._1)
    result match {
      case Some(value) =>
        List(("total", word._2*value._2));
      case None =>
        List();
    }
}

/**
* As all multiplications will be in vsm we just make the addition part of
    the scalar product.
* @param word
* @param vsm
* @returns
*/
def reducingScalarProduct(word: String, vsm: List[Double]): (String,
    Double) = {
    (word,vsm.sum)
}
```

El mapper rebrà una dupla de paraula, pes, i una llista del vsm amb el que comparar la paraula actual. En primer lloc buscarà la paraula dins l'altre vsm, i si la troba retornarà una tupla amb, en la primera posició, la paraula "total", ja que d'aquesta manera acumularem els resultats i, en segona posició, la multiplicació del pes de la paraula rebuda amb el seu pes dins el vector space model rebut. En cas que no hi sigui, retornarà una llista buida. Es guardarà en un diccionari de tuples ('total', pes) on pes és la multiplicació de cada una de les paraules.

El reducer doncs només cal que sumi tots els pesos, que tindran la mateixa clau, i ja haurem fet el producte escalar. Vam veure que utilitzar aquest MapReducer per fer el càlcul del producte escalar ens multiplicava el temps per aproximadament 1,5.

Els tipus del MapReduce, seran els següents:

- K1: (String, Double): (Paraula, Pes en el VSM1)
- V1: (String, Double): (Paraula, Pes en el VSM2)
- K2: String: 'total'.
- V2: Double: Multiplicació dels pesos de la paraula en els dos vsms.
- V3: Double: Suma de totes les multiplicacions dels pesos de les paraules.

6 Jocs de Proves

6.1 Proves de la primera part

Per fer proves de la primera part hem creat un menú, que permet provar les diferents opcions. La opció 0 permet fer tots els càlculs amb el fitxer precarregat. La resta d'opcions serveixen per modificar els paràmetres (fitxers a utilitzar, fitxer de stopwords, mida dels n-grames) i per acabar opcions que permeten directament executar les proves que s'expliquen a continuació.

6.1.1 Prova 1

La prova 1 coincideix amb els exemples que trobavem a l'enunciat per comprovar que tot funcionés correctament. Carrega el fitxer pg11 i en calcula les similituds amb/sense stopwords i les freqüències d'n-grams amb n=3. Finalment calcula el coeficient de similitud utilitzant el cosinèssim entre pg11 i pg12.

```
Entra la opció desitjada: 5
Num de Paraules:    30419     Diferents:     3007
Paraules       ocurrencies   frequencia
--------------------------------------
the              1818   5.98
and               940   3.09
to                809   2.66
a                 690   2.27
of                631   2.07
it                610   2.01
she               553   1.82
i                 545   1.79
you               481   1.58
said              462   1.52
```

Figura 6: Exemple Referències amb StopWords p1

```
Num de Paraules:    10038     Diferents:     2623
Paraules       ocurrencies   frequencia
--------------------------------------
alice             403   4.01
gutenberg          93   0.93
project            87   0.87
queen              75   0.75
thought            74   0.74
time               71   0.71
king               63   0.63
turtle             59   0.59
began              58   0.58
tm                 57   0.57
```

Figura 7: Exemple Referències sense StopWords p1

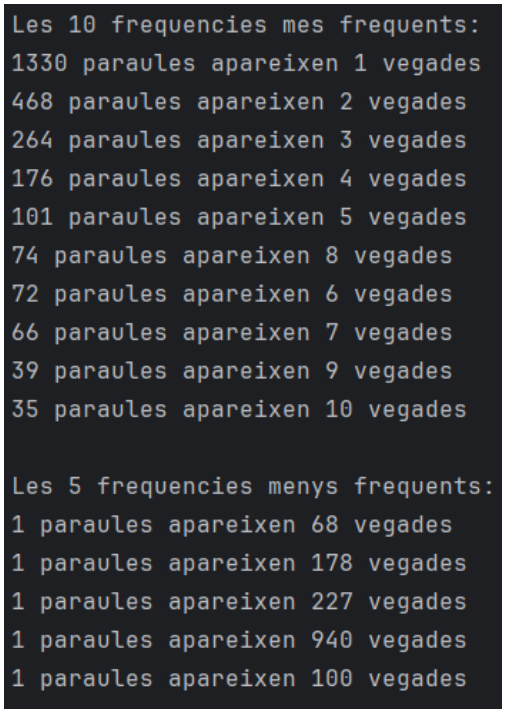

Figura 8: Exemple Referències Freqüents p1

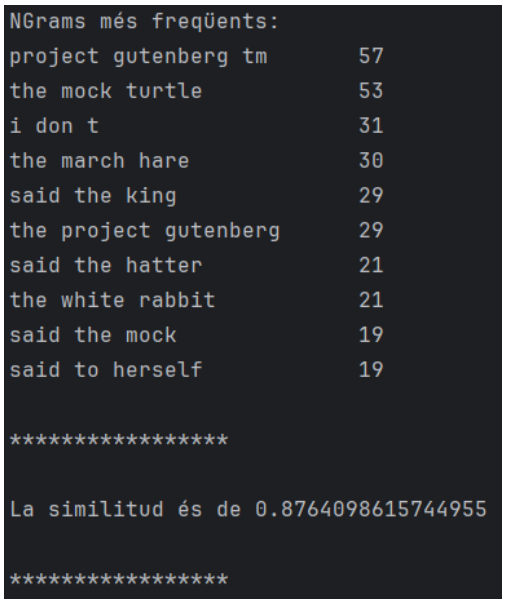

Figura 9: Exemple NGrams més freqüents i cosinèsim p1

6.1.2 Prova 2

La segona prova té com a objectiu comprovar que tots els càlculs son instantanis fins i tot utilitzant un fitxer molt més gran. El que fa és prendre dos fitxers que no s'havien utilitzat en l'altra prova, i fer tots els càlculs sobre referències, n-grames i freqüencies de referències sobre el fitxer més gran que tenim. En acabat en mesura el cosinèsim amb el segon fitxer més gran.

També aprofitem per comprovar que funciona correctament les proves amb diferents nombres d'n-grames, provant amb n=5.

```
Entra la opció desitjada: 6
Num de Paraules:    77488     Diferents:     7625
Paraules       ocurrencies   frequencia
---------------------------------------
the              3973   5.13
and              3193   4.12
a                1955   2.52
to               1807   2.33
of               1585   2.05
it               1332   1.72
he               1256   1.62
was              1170   1.51
that             1044   1.35
i                1018   1.31
```

Figura 10: Exemple Referències amb StopWords p2

```
Num de Paraules:    28559     Diferents:     7192
Paraules       ocurrencies   frequencia
---------------------------------------
tom               824   2.89
huck              258   0.90
time              191   0.67
joe               170   0.60
boys              158   0.55
boy               136   0.48
back              121   0.42
becky             115   0.40
began             110   0.39
good              108   0.38
```

Figura 11: Exemple Referències sense StopWords p2

```
Les 10 frequencies mes frequents:
3640 paraules apareixen 1 vegades
1269 paraules apareixen 2 vegades
642 paraules apareixen 3 vegades
423 paraules apareixen 4 vegades
240 paraules apareixen 5 vegades
189 paraules apareixen 6 vegades
169 paraules apareixen 7 vegades
134 paraules apareixen 8 vegades
96 paraules apareixen 10 vegades
85 paraules apareixen 9 vegades

Les 5 frequencies menys frequents:
1 paraules apareixen 131 vegades
1 paraules apareixen 309 vegades
1 paraules apareixen 232 vegades
1 paraules apareixen 882 vegades
1 paraules apareixen 100 vegades
```

Figura 12: Exemple Referències Freqüents p2

```
NGrams més freqüents:
the project gutenberg literary archive  13
project gutenberg literary archive foundation  13
project gutenberg tm electronic works  12
i don t want to            7
the terms of this agreement   7
project gutenberg tm electronic work   6
full project gutenberg tm license   6
to the project gutenberg literary   6
i don t care for           6
of the project gutenberg tm   5

*****************

La similitud és de 0.29932983555980053

*****************
```

Figura 13: Exemple NGrams més freqüents i cosinèsim p2

6.1.3 Prova 3

L'objectiu d'aquesta prova és comprovar què passa quan executem un fitxer contra ell mateix i en busquem la similitud. Com podem veure en el càlcul, la similitud és de 1.0000000000000002, el que podem atribuïr a un error d'arrodoniment de la màquina.

```
Entra la opció desitjada: 7
Num de Paraules:    30419     Diferents:     3007
Paraules       ocurrencies   frequencia
---------------------------------------
the              1818   5.98
and               940   3.09
to                809   2.66
a                 690   2.27
of                631   2.07
it                610   2.01
she               553   1.82
i                 545   1.79
you               481   1.58
said              462   1.52
```

Figura 14: Exemple Referències amb StopWords p3

```
Num de Paraules:    10038     Diferents:     2623
Paraules       ocurrencies   frequencia
---------------------------------------
alice             403   4.01
gutenberg          93   0.93
project            87   0.87
queen              75   0.75
thought            74   0.74
time               71   0.71
king               63   0.63
turtle             59   0.59
began              58   0.58
tm                 57   0.57
```

Figura 15: Exemple Referències sense StopWords p3

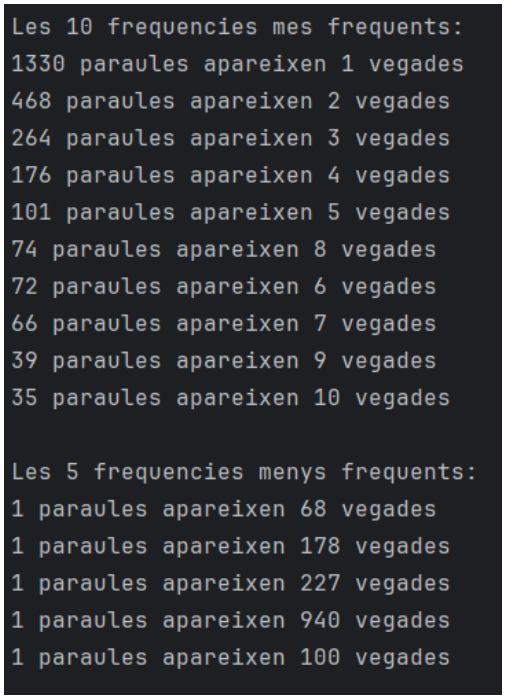

Figura 16: Exemple Referències Freqüents p3

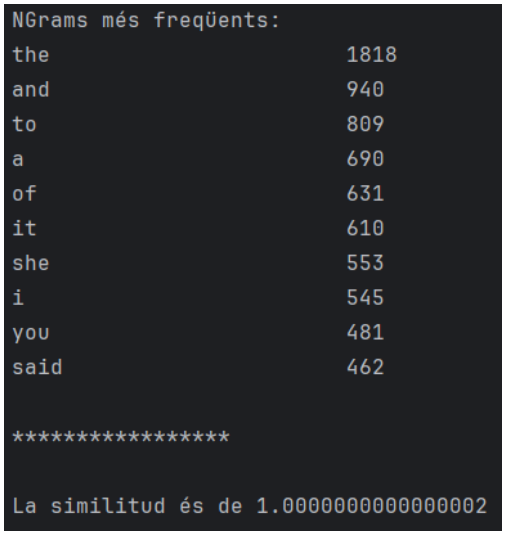

Figura 17: Exemple NGrams més freqüents i cosinèsim p3

6.2 Proves de la segona part

En la segona part de la pràctica també hem elaborat un menú que permet executar els jocs de proves directament sense posar paràmetres ni redirigir fitxers. Les opcions de 0-4, però, en comptes de executar-se amb paràmetres definits, cada una executa una funció diferent de les explicades a continuació, demanant cada una d'elles els paràmetres que es volen utilitzar per dur a terme la seva tasca.

Les opcions que hi ha son: calcular la mitjana de referències, calcular la mitjana d'imatges, calcular les pàgines més rellevants, amb paràmetres personalitzats, calcular les pàgines més similars i calcular les pàgines més rellevants entre les més similars.

Nota: Totes les proves s'han fet executant una prova rere l'altra en un sol fil d'execució sense reinciar-la.

6.2.1 Prova 1

Aquesta prova és molt simple. Es busquen les 4 pàgines més rellevants i que s'assemblin més o igual que un llindar de similitud de 0.0. S'utilitzen 16 mappers i 16 reducers per agilitzar el procés.

La finalitat d'aquesta prova és poder comprovar fàcilment que totes les parelles de pàgines s'estan comparant entre si, ja que podem veure a la sortida que s'han generat totes les possibles parelles. Per això en aquesta prova s'utilitza un llindar de 0.0, perquè es mostrin totes.

```
Entra la opció desitjada: 5
Executant Prova1: 4 Pàgines més rellevants amb llindar de similitud 0.0:
Començem a comptar el temps d'execució.
Hem rebut lencarrec
Going to create MAPPERS!!
All sent to Mappers, now start listening...
All sent to Reducers
All Done from Reducers!
Pàgines més rellevants:
(Segona Guerra Mundial,5237)
(França,2394)
(Alemanya,2049)
(Regne Unit,1592)
```

Figura 18: Execució Prova 1 Part 2 1/2

```
Les pàgines més similars de 0.0 són:
((Segona Guerra Mundial,Alemanya),0.08749918158363128)
((França,Regne Unit),0.04556424631965856)
((França,Alemanya),0.02707815014145392)
((Regne Unit,Alemanya),0.026260034830822447)
((Regne Unit,Segona Guerra Mundial),0.013165868160921558)
((França,Segona Guerra Mundial),0.008682476569534475)
La funció ha tardat: 3735 milisegonds
```

Figura 19: Execució Prova 1 Part 2 2/2

6.2.2 Prova 2

En aquesta prova es busquen les 100 pàgines més rellevants i que s'assemblin més o igual que un llindar de similitud de 0.5, que és el mínim que es demana a la pràctica. En aquest cas es fa amb 1 mapper i 1 reducer per obtenir una aproximació del que es tarda a fer-ho de manera seqüencial, sense aprofitar els avantatges d'utiltizar MapReduce.

```
Entra la opció desitjada: 6
Executant Prova2: 100 pàgines més rellevants amb llindar de similitud 0.5 amb 1 mapper i 1 reducer (seqüencial):
Començem a comptar el temps d'execució.
Hem rebut lencarrec
Going to create MAPPERS!!
All sent to Mappers, now start listening...
All sent to Reducers
All Done from Reducers!
Pàgines més rellevants:
(Segona Guerra Mundial,5237)
(França,2394)
(Alemanya,2049)
(Regne Unit,1592)
(1945,1530)
(Primera Guerra Mundial,1444)
(Itàlia,1324)
```

Figura 20: Execució Prova 2 Part 2 1/2

```
Les pàgines més similars de 0.5 són:
((Creu de Cavaller de la Creu de Ferro,Creu de Ferro),0.693990942778031)
((Iran,Àsia),0.557903455417625)
((Segona Guerra Mundial,Front Oriental de la Segona Guerra Mundial),0.5276380316955174)
((Orde de l'Imperi Britànic,Orde del Bany),0.5129338771279432)
La funció ha tardat: 6035 milisegonds
```

Figura 21: Execució Prova 2 Part 2 2/2

6.2.3 Prova 3

Aquesta prova és similar a l'anterior. Altra vegada es busquen les 100 pàgines més rellevants i que s'assemblin més o igual que un llindar de similitud de 0.5, però ara amb 16 mappers i 16 reducers. Podem veure que hi ha una millora en el temps d'execució, encara que a causa del limitat nombre de pàgines utilitzades la millora és petita.

```
Entra la opció desitjada: 7
Executant Prova3: 100 pàgines més rellevants amb llindar de similitud 0.5 amb 16 mappers i 16 reducers:
Començem a comptar el temps d'execució.
Hem rebut lencarrec
Going to create MAPPERS!!
All sent to Mappers, now start listening...
All sent to Reducers
All Done from Reducers!
Pàgines més rellevants:
(Segona Guerra Mundial,5237)
(França,2394)
(Alemanya,2049)
(Regne Unit,1592)
(1945,1530)
(Primera Guerra Mundial,1444)
(Itàlia,1324)
(1944,1238)
(1941,1118)
(París,1116)
(Londres,1020)
(Europa,1007)
(1939,989)
```

Figura 22: Execució Prova 3 Part 2 1/2

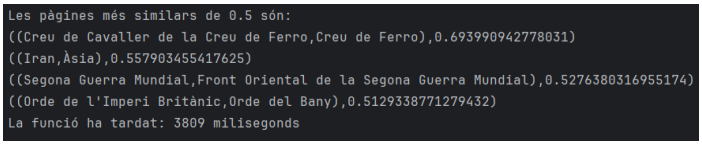

Figura 23: Execució Prova 3 Part 2 2/2

6.2.4 Prova 4

Semblant a la prova 2. Es busquen les 500 pàgines més rellevants i que s'assemblin més o igual que un llindar de similitud de 0.5, altra vegada fent servir només 1 mapper i un reducer, per poder-ho comparar amb el cas en què paral·lelitzem la feina.

Figura 24: Execució Prova 4 Part 2 1/3

Figura 25: Execució Prova 4 Part 2 2/3

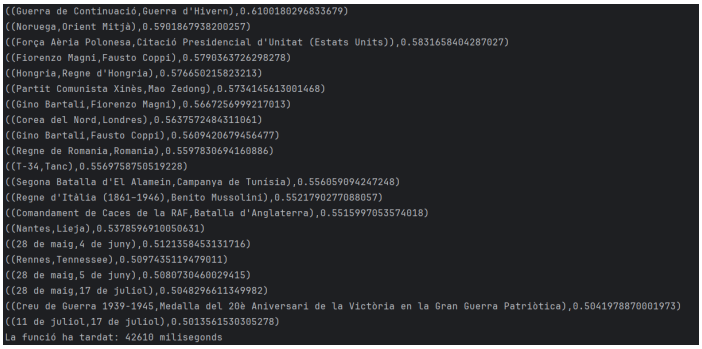

Figura 26: Execució Prova 4 Part 2 3/3

6.2.5 Prova 5

Altra vegada, aquesta prova és semblat a l'anterior. Es busquen les 500 pàgines més rellevants i que s'assemblin més o igual que un llindar de similitud de 0.5, però ara tornant a fer servir 16 mappers i 16 reducers. Aquesta vegada la millora és molt més evident, i realment es nota com el MapReduce ens ajuda molt amb la seva capacitat de paral·lelització.

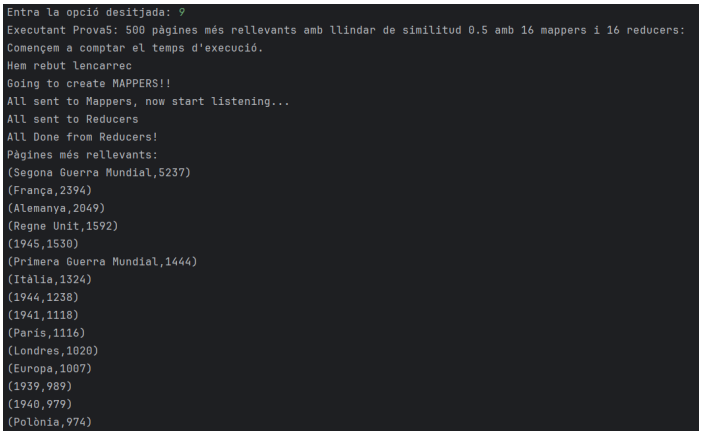

Figura 27: Execució Prova 5 Part 2 1/3

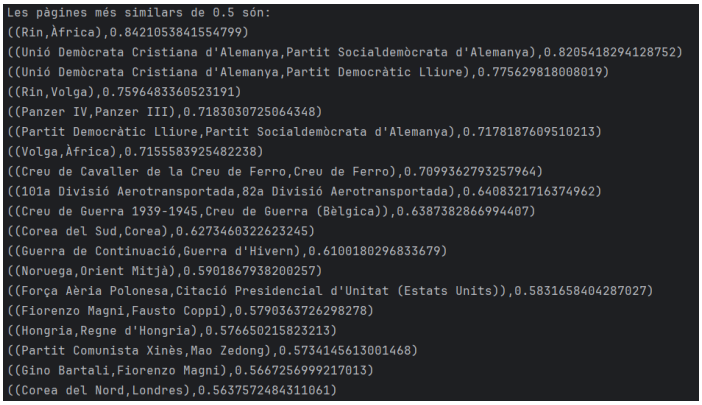

Figura 28: Execució Prova 5 Part 2 2/3

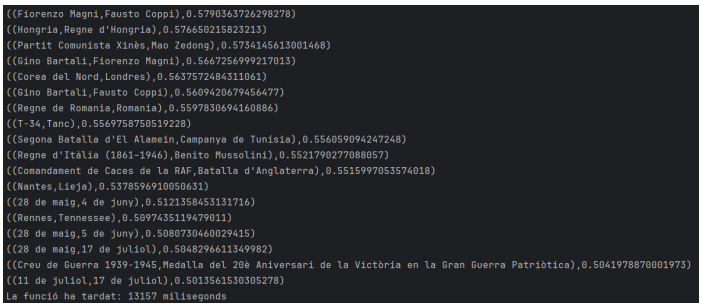

Figura 29: Execució Prova 5 Part 2 3/3

6.2.6 Prova 6

En aquesta prova busquem les pàgines que s'assemblin més o igual que un llindar de similitud de 0.5, però aquesta vegada utilitzant les 100 primeres pàgines, sense buscar les més rellevants. En aquest cas la prova es fa amb 16 mappers i 16 reducers. Així, podem veure més o menys quant tarda aquest procés per si sol.

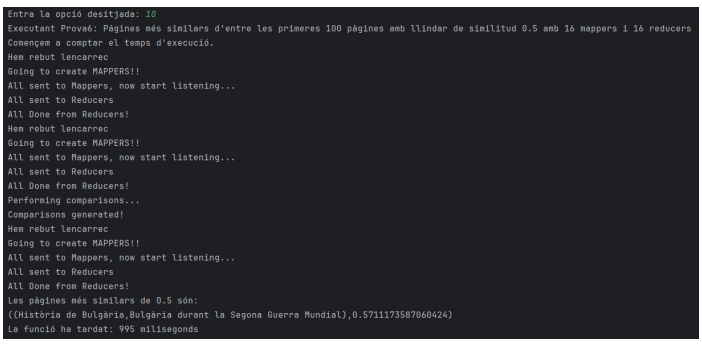

Figura 30: Execució Prova 6 Part 2

6.2.7 Prova 7

El mateix que abans, però ara amb les primeres 500 pàgines. Altra vegada utilitzem un llindar de similitud de 0.5 i 16 mappers i 16 reducers.

Figura 31: Execució Prova 7 Part 2 1/2

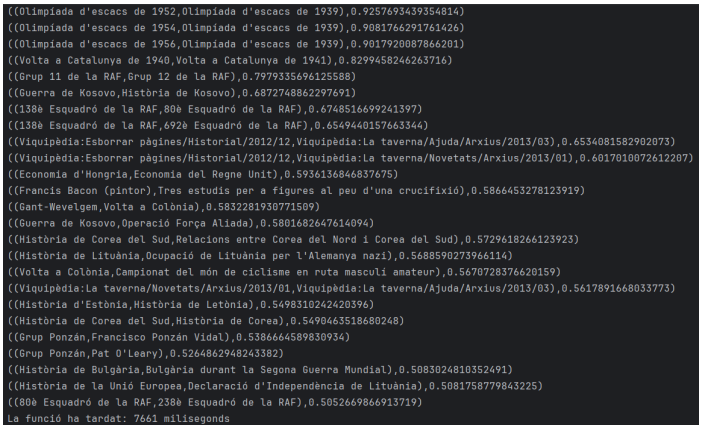

Figura 32: Execució Prova 7 Part 2 2/2

6.2.8 Prova 8

Aquesta prova calcula la mitjana de referències tenint en compte totes les pàgines i utilitzant 16 mappers i 16 reducers.

```
Entra la opció desitjada: 12
Executant Prova8: Promig de referències amb 16 mappers i 16 reducers
Començem a comptar el temps d'execució.
Hem rebut lencarrec
Going to create MAPPERS!!
All sent to Mappers, now start listening...
All sent to Reducers
All Done from Reducers!
La mitjana de referències a cada pàgina és 21.105263157894736
La funció ha tardat: 1957 milisegonds
```

Figura 33: Execució Prova 8 Part 2

6.2.9 Prova 9

Aquesta prova calcula la mitjana d'imatges tenint en compte totes les pàgines i utilitzant 16 mappers i 16 reducers.

```
Entra la opció desitjada: 13
Executant Prova8: Promig d'imatges amb 16 mappers i 16 reducers
Començem a comptar el temps d'execució.
Hem rebut lencarrec
Going to create MAPPERS!!
All sent to Mappers, now start listening...
All sent to Reducers
All Done from Reducers!
3417
La mitjana de imatges a cada pàgina és 6.045386746217771
La funció ha tardat: 1668 milisegonds
```

Figura 34: Execució Prova 9 Part 2

7 Taula de rendiment segons el nombre d'actors

L'objectiu d'aquest apartat és mostrar com evoluciona el temps d'execució si canviem el nombre de mappers i reducers en funció del nombre de documents que vulguem tenir en compte. L'execució triada és la més costosa, que consisteix en, primer calcular d'entre les n pàgines (n escollit per l'usuari) quines són les més rellevants, i posteriorment calcular d'entre aquestes n pàgines quines són les que tenen un índex de similitud, utilitzant la similitud de VSMS i l'índex tf_idf, superior a un nombre entre 0 i 1 també entrat per l'usuari.

Veiem doncs que clarament tenim 3 variables, però com que la menys rellevant és el llindar a partir del que mostrem les pàgines el mantindrem estàtic i variarem, el nombre d'actors i el nombre de pàgines a utilitzar. Hem triat un mínim de 10 fitxers perquè fer l'execució amb menys hem vist que no tenia gaire sentit perquè és pràcticament instantani i tenim molta variabilitat en el temps en nombres de fitxers inferiors. El màxim és 4693 perquè era el màxim nombre de pàgines de la Viquipedia que teniem. Els nombres entremig s'han triat per intentar veure mostres representatives i que aportessin informació de diferents escales per poder comparar els nombres d'actors.

Els nombres d'actors estaven preescollits per l'enunciat (1, 4, 10 i 20) i hem decidit afegir 16 actors i 50. Els 16 actors els hem afegit perquè durant l'elaboració d'aquesta pràctica hem desenvolupat una teoria. Com que vèiem que 10 i 20 actors ens donaven resultats similars i petits en temps, hem pensat que 16, que és una potència de 2, podria ser el que ens donés el temps mínim. Finalment hem afegit 50 perquè ens interessava un nombre gran d'actors per comprovar què passava si l'augmentavem massa.

Totes les proves s'han dut a terme amb una màquina amb les següents característiques:

- Processador: 12th Gen Intel Core i5-12400F
- Memòria: 16.0 GB DDR4
- Targeta Gràfica: NVIDIA GeForce RTX 3060
- Memòria Física: SSD M.2

Totes les mostres s'han fet amb un llindar que no ha variat de 0.5.

Tots els elements de la taula son mitjanes de 5 execucions.

Les dades de la taula son en segons.

Llindar = 0.5		**NFitxers**				
		10	**100**	**1000**	**2000**	**4693 (Màx)**
	1	3,714	6,188	136,576	473,129	
	4	2,181	3,805	41,825	143,874	473,916
Nmappers & Reducers	**10**	2,165	3,545	30,347	107,237	320,689
	16	2,283	3,604	31,43	118,6	355,772
	20	2,13	3,64	31,449	110,254	349,19
	50	2,213	5,833	34,711	114,581	345,548

Figura 35: Taula de rendiment

A continuació a partir de les dades que podem veure en la taula de rendiment elaborarem gràfics, un per cada nombre de fitxers diferent, per veure com evoluciona el temps segons el nombre d'actors. Primer es mostraràn tots els gràfics i seguidament es comentaràn els resultats.

Tots els gràfics tenen en l'eix X nombre d'actors concurrents (és a dir nombre de mappers i nombre de reducers) i en l'eix y segons que tarda l'execució en la màquina indicada.

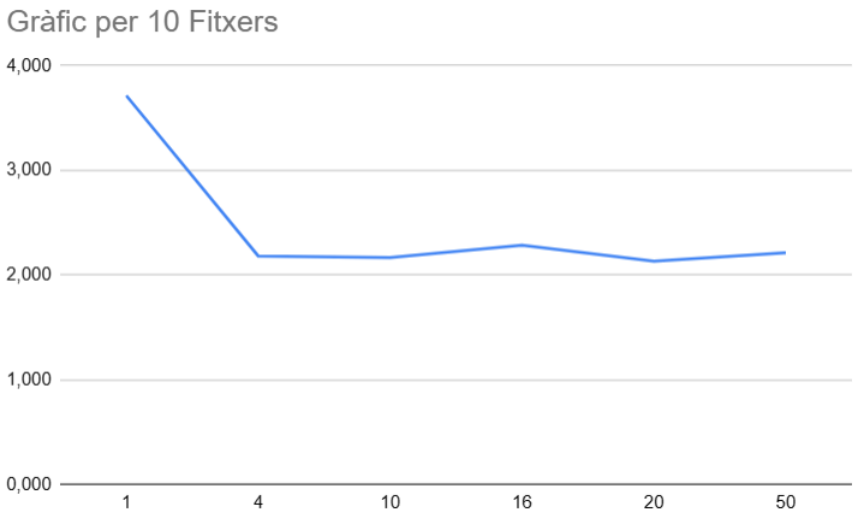

Figura 36: Gràfic per 10 fitxers segons nombre d'actors

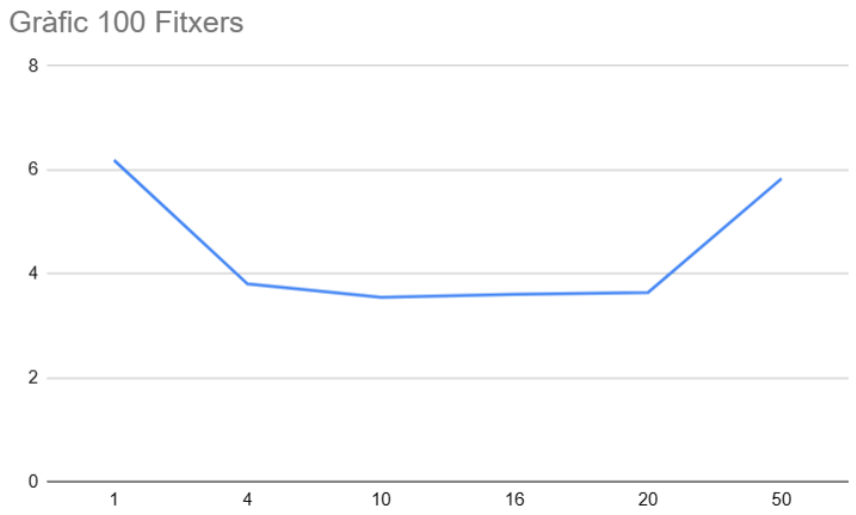

Figura 37: Gràfic per 100 fitxers segons nombre d'actors

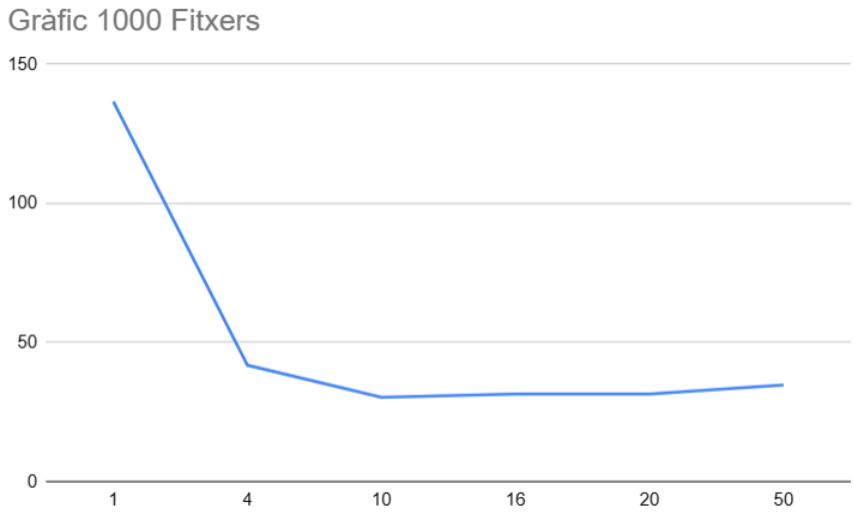

Figura 38: Gràfic per 1000 fitxers segons nombre d'actors

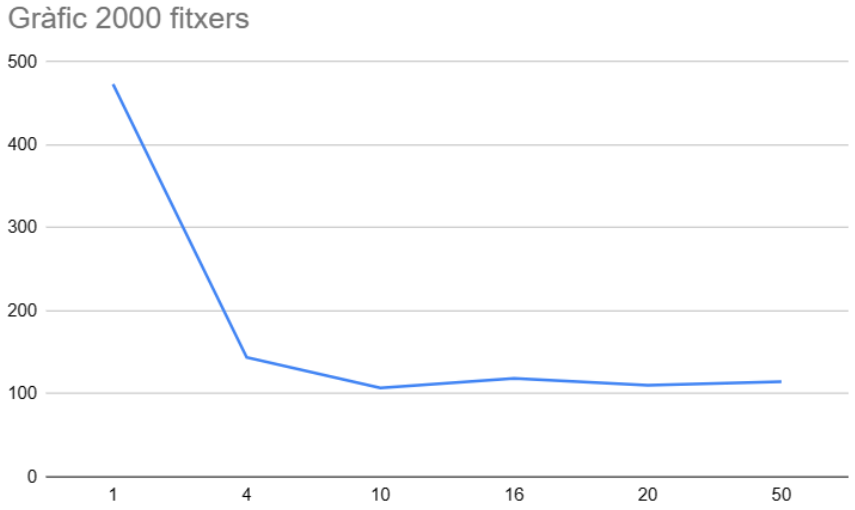

Figura 39: Gràfic per 2000 fitxers segons nombre d'actors

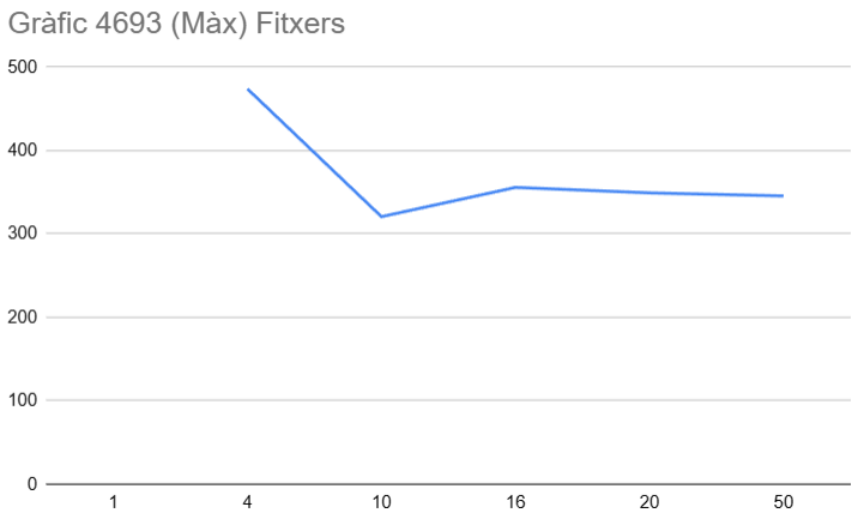

Figura 40: Gràfic per 5000 fitxers segons nombre d'actors

Tal i com podem veure en els nostres gràfics, el nombre d'actors que ens dona els millors resultats generalment és 10. En alguns casos, però 20 actors s'hi acosta, fins i tot arribant-lo a superar en 10 fitxers. Les execucions amb 1 sol actor (que podriem entendre com a seqüencials) sempre ens donen els pitjors resultats tal i com era d'esperar.

Un altre cas a destacar és el de 50 actors, on podem veure que degut a que estem creant massa actors el temps ens augmenta considerablement respecte els 20 actors. L'exemple de 10 fitxers i 50 actors no seria representatiu degut al poc temps que ens tarda a fer aquesta execució. Amb 100 fitxers veiem que el temps s'ens dispara pràcticament duplicant el de 20 actors, mentre que com més augmentem el nombre de fitxers més s'assembla el temps de 50 actors al de 20 fins i tot superant-lo amb el màxim de fitxers.

El punt més curiós és el de l'execució amb 16 actors, que pràcticament amb qualsevol nombre de fitxers dona resultats pitjors que 10 o 20 actors. No hem trobat explicació a aquest fenòmen però sería interessant fer un anàlisis en profunditat.

8 Fins a quants documents hem pogut tractar?

Hem pogut arribar a tractar els 4693 fitxers i, com podem veure a la taula de rendiment, ens ha tardat 5.8 minuts. La nostra aplicació agafa tots els fitxers si posem un nombre de pàgines a utilitzar massa gran, de manera que s'utilitzen els 4693 documents que hem dit.

```
Entra la opció desitjada: 4
Entra el nombre de pàgines a utilitzar: 5000
Entra el llindar de similitud a utilitzar: 0.5
Entra el nombre de mappers a utilitzar: 10
Entra el nombre de reducers a utilitzar: 10

**********************
INICIANT EXECUCIÓ AMB:
5000 Pàgines
0.5 de llindar de similitud
10 Mappers
10 Reducers
**********************
```

Figura 41: Paràmetres utilitzats

```
((Medalla de la Distinció Laboral,Medalla de Nakhimov),0.5029083437951415)
((Selecció de futbol d'Alemanya,Arsenal Football Club),0.5026837063138015)
((Cens dels Estats Units del 2010,Força Aèria Polonesa),0.5025887061390492)
((Dunkerque,Tailàndia),0.5024948547772771)
((Erwin Rommel,Operació Crusader),0.5024442330673597)
((Ernst Heinkel,Heinkel He 112),0.5022073258760027)
((Aeroport de Frankfurt,Aeroport Internacional de Washington-Dulles),0.5022055893305968)
((1 de juny,21 de juny),0.5019675000640629)
((Campionat del món de ciclisme en ruta masculí amateur,Tour de França de 1939),0.5017216428155785)
((Campionat del món de ciclisme en pista,Forces Armades de Turquia),0.5016912092069058)
((Medalla del combatent militar 1940-1945,Medalla per la Conquesta de Budapest),0.5013410359098054)
((Campionat del Món d'hoquei gel masculi,Llista d'asos de l'aviació de la Segona Guerra Mundial amb 49-20 victò
((Commonwealth,Campionat de Bèlgica de ciclisme en ruta masculí),0.5011110397750058)
((París-Roubaix,Llista d'asos de l'aviació de la Segona Guerra Mundial amb 19-10 victòries),0.5009501749131674)
((Consell d'Europa,París-Tours),0.5007366017491124)
((Middlesbrough Football Club,Fußball-Club Bayern München),0.5006319650512089)
((Fiorenzo Magni,Fausto Coppi),0.5005851798093369)
((Història de la Unió Soviètica (1953-1985),Desglaç de Khrusxov),0.5003933723359927)
La funció ha tardat: 320689 milisegonds
```

Figura 42: Resultats obtinguts

9 Bibliografia

Referències

[1] Varis, *Scala Basics* (Enllaç a la pàgina)

[2] Dwi Setyo Aji, Tomer Shetah. *MapView(<not computed>) in Scala* (Enllaç a la pàgina)

[3] Mateu Villaret et al., *Privat. Apunts de classe: Programació Declarativa. Aplicacions.*

[4] Misael Taveras, *Usar map, filter y reduce para olvidarnos de los bucles for* (Enllaç a la pàgina)