Utilizarea Apache Spark în generarea rețelelor de sortare

1. **Implementarea unor algoritmi distribuiți generate-and-prune**

Această secțiune descrie diferite metode de implementare a paradigmei “generate-and-prune”. Folosind acțiunile și transformările disponibile în cadrul Apache Spark au fost obținuți diverși algoritmi distribuiți de generare a rețelelor de sortare. Acești algoritmi au fost realizați într-o manieră incrementală, în sensul că fiecare nou algoritm a fost dedus din cel precedent, crescând marja superioară a numărului de fire, *n*, pentru care problema numărului optim de comparatori a fost rezolvată, și, îmbunătățind totodată performanțele și timpul de execuție pentru valori ale lui *n* pentru care problema fusese deja rezolvată de algoritmi precedenți.

**1.1 Stocarea distribuită a rețelelor de comparatori**

Pe tot parcursul execuției programului, rețelele de comparatori sunt stocate într-un *RDD* (*JavaRDD*). Acest lucru permite stocarea distribuită, pe mai multe mașini, ale rețelelor și, totodată, procesarea lor paralelă folosind acțiuni și tranformări disponibile în biblioteca Java Apache Spark. În diferiții algoritmi implementați anumite proceduri presupun transformarea rețelelor dintr-un *RDD* într-un *PairRDD* (*JavaPairRDD*), adică perechi de forma *<cheie, valoare>*. Aceste transformări permit reducerea (pruning) ulterioară a rețelelor de comparatori, tot în paralel.

**1.2 Implementarea pasului „generate”**

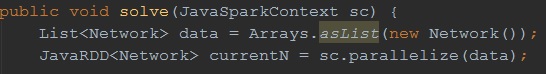
Generarea rețelelor de sortare se face într-o manieră iterativă. Se pleacă de la rețeaua vidă, fară niciun comparator și se realizează *k* pași, unde *k* reprezintă numărul optim de comparatori pentru o rețea de sortare cu *n* fire. Fiecare pas nou presupune adăugarea oricărui comparator posibil la toate rețelele obținute la pasul anterior.

Această abordare reprezintă practic un algoritm de parcurgere în lățime *BFS* (*Breadth-first search*) a unui arbore, unde nodurile de pe fiecare nivel *l* reprezintă rețele cu *l* comparatori. Nodurile de pe nivelul următor sunt obținute adăugând la fiecare rețea (fiecare nod) toți comparatorii posibili pentru *n* fire. Cum pentru o rețea cu *n* fire există comparatori posibili, vor fi obținuțe noi rețele din fiecare nod.

Pentru a mai reduce însă din numărul rețelelor ce sunt obținute la fiecare pas se evită adăugarea de comparatori redundanți. Considerăm o rețea de sortare de comparatori de forma *C;(i, j);C’*. Spunem că un comparator *(i, j)* este redundant dacă *xi*≤ *xj* pentru fiecare secvență *x1, x2, ... xn* ∈ outputs(C). Cu alte cuvinte, un comparator este redundant dacă nu aduce nicio schimbare (îmbunătățire) mulțimii *outputs* ale rețelei curente. Un exemplu simplu de comparator redundant reprezintă situația în care se încearcă adăugarea unui comparator *(i, j)* la o rețea ce are deja ca ultim comparator adăugat comparatorul *(i, j)*. Este evident că adaugându-l încă o dată nu se va produce nicio schimbare mulțimii *outputs*.

* + 1. **Paralelizarea rețelelor**

După cum a fost menționat, rețelele de comparatori sunt stocate într-un *RDD*, adică într-o structura de date distribuită. Pentru a realiza această paralelizare, se pleacă de la o listă formată dintr-o singură rețea, rețeaua vidă, fără niciun comparator, ce se paralelizează într-un *JavaRDD* folosind obiectul aparținând clasei *JavaSparkContext* și apelând metoda parallelize ce primește ca parametru de intrare, datele (lista) ce vor fi paralelizate.



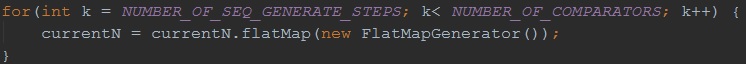
**1.2.2 FlatMap Generator**

Fiecare nivel de rețele din arbore este reprezentat printr-un *JavaRDD<Network>* denumit *currentN* (defapt, la fiecare pas este nevoie doar de un singur nivel din arbore, și anume nivelul curent, din care va fi generat nivelul următor). Pentru a obține următorul nivel al arborelui, adică pentru a adăuga toți comparatorii posibili fiecărei rețele, este folosită transformarea *flatmap*.

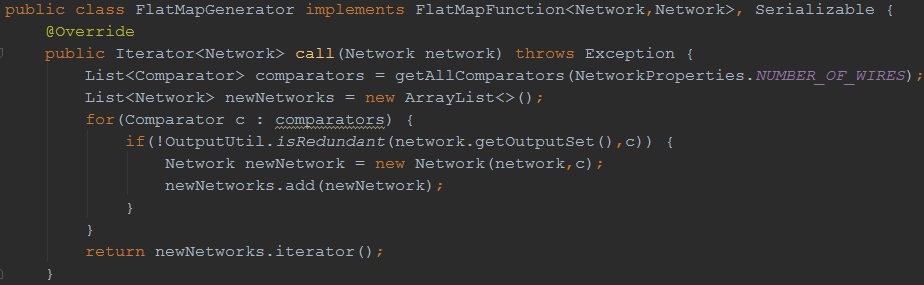
Pentru a explica utilitatea transformării flatmap, va fi explicate mai întâi transformarea din care aceasta este derivată, și anume *map*. Transformarea map aplicată unui *RDD* pasează fiecare element din *RDD* printr-o funcție și îl procesează/transformă/extinde după scopul dorit. În cazul rețelelor de comparatori, funcția map primește la intrare fiecare rețea din *RDD-ul currentN* și o transformă într-o nouă rețea, adăugând un nou comparator, neredundant.

Însă, deoarece fiecare rețea este transformată în rețele (excluzând rețelele cu comparatori redundanți) adăugând fiecare comparator neredundant, a fost folosită transformarea *flatmap,* ce este similară lui *map*, însă fiecare rețea poate fi tranformată în 0 sau mai multe noi elemente. Așadar, fiecare rețea din *RDD-ul currentN* este extinsă în alte noi rețele, folosind o funcție de tip *flatmap*, ce returnează un iterator la o listă de rețele.

Fluxul principal al algoritmului de generare este următorul:



*FlatMapGenerator* este o clasă ce implementează *FlatMapFunction*, și suprascrie metoda *call* prin care se extinde rețeaua curentă din *RDD* într-o listă de noi rețele, adăugând toți comparatorii posibili pentru *n* fire, ce nu sunt redundanți. În final se returnează un *iterator* la lista obținută.



* + 1. **Rezultate**

Folosind această abordare, au fost determinate rețelele de sortare cu număr optim de comparatori pentru un număr de fire până în 5, executând programul pe o singură mașină cu 4 nuclee si 8GB memorie RAM.

Pentru *n = 4*, au fost obținute în total *708* de rețele cu 5 comparatori, din care *12* sunt rețele de sortare (aproximativ *1,70%*).

Pentru *n = 5*, au fost obținute în total *3.269.040* de rețele cu 9 comparatori, din care *149.040* sunt rețele de sortare (aproximativ *4,55%*).

Pentru valori ale lui *n* mai mari ca 5, numărul de rețele obținute devine mult prea mare pe măsura ce sunt adăugați comparatori, și folosirea doar a unei abordări bazată pe generare este nefezabilă. Se folosește în schimb paradigma *generate-and-prune*, ce presupune excluderea anumitor rețele ce se reduc la altele pe baza unor similarități.

**1.3 Implementarea pasului “prune”**