Range Series

youneselachqar

May 2021

1 Introduction

Pour des ranges labélisés de type closed-open , on cherche à implanter la méthode matchinglabels(Item) qui retourne un array de Label où chaque label correspond à un range où se trouve l'item. Les ranges considérés sont des ranges d'entiers et les labels sont des chaînes de caractères. On propose à travers ce projet 2 versions : une version optimisée(RangeSeriesDividedImpl.java) et une version naive (RangeSeriesNaiveImpl.java).

2 Modélisation du problème

Le problème revient à retrouver tout les ranges auxquels appartient chaque entier le plus rapidement et efficacement possible.
On a choisi de modéliser les ranges labélisés sous forme de dictionnaire avec les labels comme key et les ranges comme value : $Map < String, Integer[] >$ **mapRanges**.
Dans le cas où l'on cherche à determiner un seul entier et sans avoir de connaissances au préalable concernant le dictionnaire de range labélisés , la meilleure méthode en termes de perfomances est la méthode dite Naive où l'on va parcourir les ranges un à un pour voir s'ils contiennent l'entier.
Pour pouvoir optimiser notre parcours , un traitement effectué sur le dictionnaire de ranges labélisés s'impose. Or, pour que cela soit effectivement rentable il faut que la méthode s'applique sur un nombre élévé d'entiers et c'est pour cette raison que l'on prend en entrée de la fonction matchinglabels une liste d'entiers qui correspond à tout les entiers sur lesquels on va appliquer cette fonction : $ArrayList < Integer >$ **listInts**.

3 Algorithme Naif

La première version implantée est une version plutôt simple et qui consiste à parcourir les ranges un par un pour chaque entier pour trouver toutes les occurences de l'entier.

Algorithm 1 matchingLabels(x, mapRanges)

for all range in mapRanges **do**
  if $x < upperbound(range)$ $x >= lowerbound(range)$ **then**
    $Lables[] \leftarrow Lables[] + keyRange$
  else
    skip
  end if
end for
return listLables

Cet algorithme peut être généralisé au cas où l'on prend une liste d'entiers en entrée à travers une boucle parcourant les entiers de la liste pour retourner une liste qui contient tout les Label[] correspondant aux entiers de la liste.
La correspondance entier-labels[] peut être retrouvée à travers les index : l'index de l'entier dans mapRanges est le même que l'index du label[] qui lui est associé.

Algorithm 2 matchingLabels(listInts, mapRanges)

for all int in listInts **do**
  $list(Lables[]) \leftarrow list(Labels[]) + matchingLabels(int, mapRanges)$
end for
return listLables

4 Algorithme optimisé

Afin d'optimiser l'algorithme , on choisit d'effectuer un traitement sur mapRanges afin de simplifier et d'accélerer le parcours en utilisant la méthode Map.get de Java.
Le traitement choisi est le suivant : on effectue un parcours de chaque range pour associer à chaque entier le composant le label correspondant au range. Ainsi la fin du traitement , on retrouve un dictionnaire $Map < Integer, ArrayList < String >>$ divRanges où les clés sont les entiers et les values sont les labels des ranges auxquels ils appartiennent.
Pour un x en entrée , il suffit donc d'appeler la méthode map.get sur divRanges pour retrouver les labels.

Algorithm 3 Divide(mapRanges)

```
for all range in mapRanges do
  for all int in range do
    intLabels[] ← intLabels[] + rangeLabel
  end for
end for
return listLables
```

Algorithm 4 matchingLabels(listInts, mapRanges)

```
for all i in listInts do
  list(Labels[]) ← list(Labels[]) + divRanges.get(i)
end for
return listLables
```

5 Utilisation des algorithmes

Pour tester les performances du programme , le choix s'est porté sur des sets de ranges et de ints générés aléatoirement.
Le lancement des algorithmes se fait à partir de la classe RunRangeSeries.java. Il suffit d'éxecuter la classe , le programme vous invite à rentrer le nombre de ranges labelisés et d'ints que vous souhaitez générer et à choisir l'algorithme executer puis retourne le temps d'execution.

6 Comparaison

En analysant les deux algorithmes , on peut voir que l'algorithme naîf a une complexité de $O(N * M)$ avec N le nombre d'entiers et M le nombre de ranges , tandis que l'algorithme optimisé a une opération de complexité $O(M * R)$, avec R la longueur maximale des intervalles considérés , qu'on effectue une seule fois et puis une boucle de complexité O(N).
On peut voir cela à travers les temps d'execution des deux algorithmes et on retient les points suivants :

- **Pour un faible nombre d'entiers et de ranges** , le traitement de la map n'est pas rentable : à titre d'exemple , pour 10 ranges et 50 entiers on obtient un temps d'execution de l'algorithme optimisé 10 fois plus grand que celui de l'agorithme naif.
- **Le temps d'execution de l'algorithme optimisé évolue très peu en fonction du nombre d'entiers** : pour 10 000 ranges et 100 000 entiers , le temps d'execution est de 4987ms alors qu'avec 1 000 000 d'entiers considérés le temps d'execution est de 5610ms.
- **L'algorithme naif au delà des 100000 entiers est très peu performant** , le temps d'execution dépasse 2min.

- **L'agorithme optimisé reste peu performant pour un nombre de ranges qui dépasse 300 000 ranges**

7 Améliorations possibles

Pour répondre au dernier point , plusieurs améliorations sont possibles mais toutes ont pour but de réduire la taille de la map divRanges et pour cela on pourrait implanter une subdivision basée sur les intersections de la mapRanges initiale où chaque x se trouverait dans un unique range de la subdivision et on pourrait donc le récupérer par un simple appel à Map.get .Par manque de temps je n'ai pas pu aller au bout de cet algorithme.