Une image contenant texte

Description générée automatiquementDimanche 29 octobre 2023

Gobigan MATHIALAHAN

Pierre CHEVILY

Groupe : 202 & 205

# **Compte Rendu Projet Java Période A**

# **Mini-projet**

Table des matières

[Introduction 3](#_Toc149482813)

[Comment fonctionne l’algorithme d’Union-Find ? 4](#_Toc149482814)

[Diagramme de Classes 6](#_Toc149482815)

[Présentation du jeu de tests 7](#_Toc149482816)

[Conclusion : 9](#_Toc149482817)

[Code de l’application 10](#_Toc149482818)

[UnionFind 10](#_Toc149482819)

[UnionFindMethodTest 12](#_Toc149482820)

[UnionFindBenchMark 14](#_Toc149482821)

# Introduction

L’objectif du projet est de réaliser une application Java illustrant un algorithme d’Union-Find.

Nous avions un code à notre disposition, celui d’un TD de l’année dernière. Le but de ce dernier était de créer une structure de données illustrant un monde parfait dans lequel des personnes qui se rencontrent doivent déterminer si elles sont déjà amies ou non. De plus, les devises « les amis de mes amis sont mes amis » et « mes amis sont mes amis pour la vie » régissent ce monde. Autrement dit, une personne A amie avec une personne B le sera à jamais, et héritera des amis de B.

L’algorithme principal consiste à interroger la base de données de notre programme afin de savoir si 2 personnes sont amies ou non. Si elles ne le sont pas, elles pourront le devenir en créant un lien d’amitié entre elles.

Le code source nous étant déjà fourni (sous forme de pseudo-algorithme), l’objectif principal était de trouver la meilleure application pour réaliser ce projet en termes de complexité temporelle et mémorielle.

# Comment fonctionne l’algorithme d’Union-Find ?

Pour expliquer l’algorithme, prenons un exemple :

Une image contenant Police, texte, noir et blanc, blanc

Description générée automatiquement

Dans ce tableau, la ligne « Node » correspond à la personne en elle-même. La ligne « Parent » symbolise le représentant du groupe d'amis de cette personne. La ligne « Rank » indique le nombre de fils que la personne possède, soit le nombre d’amis « directs » qu’elle a. Ici, comme il n'y a pas de lien d'amitié entre 2 personnes différentes, les représentants des groupes d'amis sont les personnes elles-mêmes et il n'y a aucun fils.

Une image contenant texte, Police, capture d’écran, diagramme

Description générée automatiquement

Ici, nous réalisons l’opération « Merge », qui consiste donc à rassembler 2 groupes d’amis. On voit donc que **3** pointe désormais sur **0**, et que **0** possède maintenant 1 fils. Le représentant du groupe d’amis de **3** est dorénavant **0**.

Une image contenant Police, texte, diagramme, capture d’écran

Description générée automatiquement

**2** et **3** sont désormais amis. Cela va entraîner la même opération que vu précédemment. **1** va donc avoir 1 fils, et **2** pointera vers **1**, qui est son représentant.

Une image contenant texte, diagramme, ligne, Police

Description générée automatiquementEnfin, si on réalise l’opération « Merge(3,1) », le représentant du groupe d’amis de **3**, soit **0**, deviendra aussi celui du groupe d’amis de **1**. Comme **1** possédait d’ores et déjà 1 fils, **0** aura dorénavant 2 fils.

# Diagramme de Classes

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

# Présentation du jeu de tests

Le fichier UnionFindBenchmark.java fournit un fichier nommé output.txt où sont indiquées les performances de chaque fonction. Toutes les mesures sont en millisecondes.

* La colonne « Friends » indique le nombre d'amis qui sera utilisé pour l'opération correspondante.
* La colonne « Add time » représente le temps que met l'algorithme pour ajouter Friends amis.
* « Find time » correspond au temps nécessaire au programme pour évaluer si deux personnes sont amies ou non.
* La colonne « Merge time », quant à elle, indique le nombre de millisecondes qu'a pris l'algorithme pour créer un nouveau lien d'amitié entre deux personnes.

Pour réaliser les graphiques de la complexité temporelle, nous avons utilisé le logiciel Excel. Nous avons converti les données d'output.txt au format Excel. Puis, pour chaque fonction, nous avons réalisé un histogramme, ainsi qu'une courbe, en rouge, représentant la complexité de la fonction. Pour chaque fonction, vous aurez le nombre d'amis en colonne et le nombre de millisecondes en ligne.

# Conclusion :

Ce projet a été enrichissant. Il nous a permis de progresser sur la manière de visualiser des projets. En effet, auparavant, nous ne tenions pas vraiment compte de la complexité, mais elle a été primordiale sur celui-ci.

L’étape la plus longue pour nous a été le choix de la structure de données. Nous hésitions avec une ***LinkedList***, une ***ArrayList*** et une ***HashMap***. C’est finalement cette dernière que nous avons retenue. Effectivement, la ***LinkedList*** ne nous a pas semblé très avantageuse, car elle n’offrait pas des performances à la hauteur de la difficulté qu’elle requiert pour l’utiliser. Pour ce qui est de ***l’ArrayList***, nous avons préféré la ***HashMap***, car nous avons jugé plus compréhensible l’utilisation d’un dictionnaire. En effet, nous nous sentions plus à l’aise avec ce type de données, outre le fait que la ***HashMap*** soit plus flexible qu’une ***ArrayList***.

En dehors de ce choix, nous n’avons pas éprouvé de difficultés particulières. En revanche, nous avons appris à rediriger la sortie standard vers un fichier .txt, en Java ainsi qu’à utiliser des types génériques.

Pour ce qui est des tests, nous avons fait le choix de fournir un grand jeu de tests, afin de bien mesurer les performances. C'est pourquoi le programme sera long à exécuter. Sur nos machines, le test a duré environ 15 minutes.

# Code de l’application

## UnionFind

*package* UnionFind;  
  
*import* java.util.HashMap;  
*import* java.util.Map;  
  
*public class* UnionFind<T>{  
 *private final* Map<T,T> **parent**;  
 *private final* Map<T,Integer> **rank**;  
  
 *public* UnionFind() {  
 **parent** = *new* HashMap<>();  
 **rank** = *new* HashMap<>();  
 }  
 *public* Map<T,T> getParent() {  
 *return this*.**parent**;  
 }  
 *public* Map<T,Integer> getRank() {  
 *return this*.**rank**;  
 }  
  
 */\*\*  
 \* Permet de créer un Set  
 \** ***@param element*** *: l'élément à ajouter  
 \* Complexité temporelle : O(1) → On ajoute simplement un élément donc la complexité est constante  
 \* Complexité en espace : O(1) → On ajoute simplement un élément donc la complexité est constante  
 \*/  
 public void* add(T element) {  
 *if* (!**parent**.containsKey(element)) {  
 **parent**.put(element,element);  
 **rank**.put(element,0);  
 }  
 }  
  
 */\*\*  
 \* Permet de trouver le représentant d'un groupe d'amis  
 \** ***@param element*** *: la personne dont on souhaite trouver le représentant  
 \** ***@return*** *le représentant du groupe d'amis  
 \* Complexité temporelle : O(log(n)) → On parcourt l'arbre jusqu'à trouver le représentant  
 \* Complexité en espace : O(1) → On ajoute simplement un élément donc la complexité est constante  
 \*/  
 public* T find(T element) {  
 *if* (!**parent**.get(element).equals(element)) { *// Si ce n'est pas le représentant* **parent**.put(element,find(**parent**.get(element)));  
 }  
 *return* **parent**.get(element);  
 }  
  
 */\*\*  
 \* Permet de fusionner deux groupes d'amis  
 \** ***@param element1*** *: le premier groupe  
 \** ***@param*** *element2 : le second groupe  
 \* Complexité temporelle : O(log(n)) → On parcourt l'arbre jusqu'à trouver le représentant  
 \* puis on compare les rangs  
 \* Complexité en espace : O(1) → On ajoute simplement un élément donc la complexité est constante  
 \*/  
 public void* mergeSet(T element1, T element2) {  
 T parent = find(element1);  
 T parent2 = find(element2);  
 *if* (!parent.equals(parent2)) {  
 *if* (**rank**.get(parent) < **rank**.get(parent2)) {  
 *this*.**parent**.put(parent,parent2);  
 }  
 *else if* (**rank**.get(parent) > **rank**.get(parent2)) {  
 *this*.**parent**.put(parent2,parent);  
 }  
 *else* {  
 *this*.**parent**.put(parent2,parent);  
 *this*.**rank**.put(parent,**rank**.get(parent)+1);  
 }  
 }  
 }  
}

## UnionFindMethodTest

*package* Test;  
  
*import* java.util.HashMap;  
*import* java.util.Map;  
*import* UnionFind.UnionFind;  
*import static* org.junit.jupiter.api.Assertions.\*;  
  
*class* UnionFindMethodTest {  
 *private static* UnionFind<Integer> *uf* = *new* UnionFind<Integer>();  
 *private static void* addUf() {  
 *for* (*int* i = 0; i<=5; i++) *uf*.add(i);  
 }  
  
 @org.junit.jupiter.api.Test  
 *void* add() {  
 addUf();  
 Map<Integer,Integer> expectedParent = *new* HashMap<>();  
 expectedParent.put(0,0);  
 expectedParent.put(1,1);  
 expectedParent.put(2,2);  
 expectedParent.put(3,3);  
 expectedParent.put(4,4);  
 expectedParent.put(5,5);  
  
 assertEquals(*uf*.getParent(), expectedParent);  
 }  
  
 @org.junit.jupiter.api.Test  
 *void* find() {  
 addUf();  
 *uf*.mergeSet(0,2);  
 assertEquals(0,*uf*.find(0));  
 assertEquals(1,*uf*.find(1));  
 assertEquals(0,*uf*.find(2));  
 assertEquals(3,*uf*.find(3));  
 assertEquals(4,*uf*.find(4));  
 assertEquals(5,*uf*.find(5));  
 }  
  
 @org.junit.jupiter.api.Test  
 *void* merge(){  
 addUf();  
 *uf*.mergeSet(0,2);  
 *uf*.mergeSet(3,5);  
 *uf*.mergeSet(0,3);  
 Map<Integer,Integer> expectedRank = *new* HashMap<>();  
 expectedRank.put(0,2);  
 expectedRank.put(1,0);  
 expectedRank.put(2,0);  
 expectedRank.put(3,1);  
 expectedRank.put(4,0);  
 expectedRank.put(5,0);  
  
 assertEquals(*uf*.getRank(),expectedRank);  
 }  
  
 @org.junit.jupiter.api.Test  
 *void* remove(){  
 addUf();  
 *uf*.mergeSet(0,2);  
 *uf*.mergeSet(3,5);  
 *uf*.mergeSet(0,3);  
  
 Map<Integer,Integer> expectedRank = *new* HashMap<>();  
 expectedRank.put(0,1);  
 expectedRank.put(1,0);  
 expectedRank.put(2,0);  
 expectedRank.put(3,0);  
 expectedRank.put(4,0);  
 expectedRank.put(5,0);  
  
 assertEquals(*uf*.getRank(),expectedRank);  
 }  
}

## UnionFindBenchMark

*package* UnionFind;  
  
*import* java.io.FileOutputStream;  
*import* java.io.PrintStream;  
*import* java.util.Arrays;  
*import* java.util.List;  
*import* java.util.Random;  
*import* java.util.function.Consumer;  
  
*public class* UnionFindBenchMark {  
 *private final static int* ***MIN*** = 100\_000, ***MAX*** = 1\_000\_000, ***PAS*** = 1000;  
 *private static* Random *rand* = *new* Random();  
 *public static double* getTime(Consumer<UnionFind<Integer>> function, UnionFind<Integer> uf) {  
 *long* d = System.nanoTime();  
 function.accept(uf);  
 *return* (System.nanoTime() - d) / 1E6;  
 }  
 *public static void* main(String[] args) {  
 *try* {  
 *// On redirige la sortie standard vers un fichier* PrintStream out = *new* PrintStream(*new* FileOutputStream("output.txt"));  
 System.setOut(out);  
  
 List<Consumer<UnionFind<Integer>>> functions = Arrays.asList (  
 uf -> {  
 *for* (*int* i = 0; i < ***MAX***; i++) {  
 uf.add(i);  
 }  
 },  
 uf -> {  
 *for* (*int* i = 0; i < ***MAX***; i++) {  
 uf.find(*rand*.nextInt(***MAX***)); *// Random find* }  
 },  
 uf -> {  
 *for* (*int* i = 1; i < ***MAX***; i++) {  
 *int* randomElement1 = *rand*.nextInt(***MAX***);  
 *int* randomElement2 = *rand*.nextInt(***MAX***);  
 uf.mergeSet(randomElement1, randomElement2); *// Random merge* }  
 }  
 );  
  
 System.***out***.print("Friends;Add time;Find time;Merge time\n");  
 *long* startTime = System.nanoTime();  
 *for* (*int* n = ***MIN***; n <= ***MAX***; n += ***PAS***) {  
 System.***out***.print(n);  
 UnionFind<Integer> unionFind = *new* UnionFind<Integer>();  
 *for* (*int* i = 0; i < functions.size(); i++) {  
 Consumer<UnionFind<Integer>> f = functions.get(i);  
 *double* time = getTime(f, unionFind);  
 System.***out***.print(";" + time);  
 }  
 System.***out***.println();  
 }  
  
 out.close(); *// On arrête la redirection vers le fichier* } *catch* (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}