IUT de Paris – Rives de Seine

R3.02 – Développement efficace

Mini-Projet

L'objectif du projet est de développer et d'évaluer une classe Java correspondant à un union-find. Durant un des exercices de l'année passée, vous avez étudié le pseudo-code des algorithmes fondamentaux de cette structure. Le corrigé de cet exercice vous est fourni en annexe.

En plus des opérations fondamentales du type (union et find), il vous est demandé d'introduire une nouvelle opération permettant d'isoler un élément. Pour reprendre l'analogie du TD de l'année passée, une personne du monde parfait peut être retirer du groupe d'amis auquel elle appartient pour redevenir une personne sans ami.

De plus, votre structure devra supporter l'ajout dynamique d'un nouvel élément (toujours avec la même analogie, les personnes sont toujours immortelles mais des naissances doivent être possibles). Un élément nouvellement ajouté est considéré comme étant initialement isolé (une personne n'a pas d'ami à sa naissance).

Enfin, le TD faisait l'hypothèse que les éléments de l'union-find étaient désignés par un entier unique. Dans la mesure du possible, vous ferez en sorte que cela puisse être par des donnée d'un type choisi à la déclaration de l'union-find. Si vous ne savez pas comment faire, restez en aux entiers.

Le projet doit être fait en binôme et à son issu, vous devrez rendre un rapport (un fichier pdf) contenant :
- Une brève explication de vos choix de conception (en les justifiant par la complexité des algorithmes mis en œuvre).
- Une évaluation chiffrée des performances de votre solution accompagnée d'explications (indiquant ce que vous avez mesuré, comment vous l'avez fait et commentant les résultats obtenus).
- Le code source de votre projet.

Votre projet devra être déposé sur Moodle avant le **vendredi 27 octobre à 23h59**.

Annexe – Sujet et corrigé du TD de l'année passée (S1 - R1.01)

Thèmes

- Choix des structures de donnée
- Conception d'algorithmes
- Complexité des algorithmes

Nous faisons l'hypothèse que nous sommes dans un monde parfait dans lequel les devises "les amis de mes amis sont mes amis" et "mes amis sont mes amis pour la vie" ont été adoptées par chacun. De plus, seuls des immortels infertiles peuplent ce monde (presque) parfait. Le nombre d'habitants est connu et il n'évolue plus.

Lorsque deux habitants de ce monde se rencontrent, ils ont souvent à décider s'ils sont déjà amis ou pas. Pour répondre à cette question, chacun doit énumérer ses amis actuels afin de détecter s'ils ont des amis communs. Cette tâche (laborieuse et source d'erreur) est si fréquente qu'il a été décidé de proposer une solution informatique.

Le système que vous développerez pourra être interrogé par chacun pour savoir si deux personnes sont connues comme étant amies. Dans le cas contraire, il devra être possible d'enregistrer leur amitié nouvelle auprès du système.

Chaque habitant sera désigné dans le système par un numéro qui lui est propre (allant de 0 à $n-1$ avec n le nombre d'habitants). Le système devra être initialisé de façon à ce qu'aucun habitant n'ait d'ami.

Tous les algorithmes demandés devront être exprimés en pseudo-code.

1. **Première analyse**

   Proposez une structuration de donnée permettant d'encoder la relation d'amitié liant les habitants. Indiquez dans la mesure du possible le coût (la complexité temporelle et spatiale) des deux fonctionnalités devant être assurées par le système. Pour ce faire vous devrez imaginer et exprimer (en pseudo-code) les algorithmes correspondants.

   Solution: Les deux fonctionnalités attendues du système sont :

   1. Répondre à la question de savoir si deux habitants sont amis.
   2. Enregistrer une nouvelle amitié entre deux habitants.

   La liste des amis d'un habitant contient au plus $n-1$ habitants et peut donc être stockée dans un tableau.

   Si nous devons stocker la liste de chacun des n habitants, la structure de données va avoir une taille proportionnelle à $n \times (n-1)$. De plus, nous stockons deux à deux endroits distincts la même information. Si l'habitant h_1 apparaît dans la liste d'amis de l'habitant h_2 alors nécessairement h_2 est dans la liste d'amis de h_1.

   Étudions la complexité des deux opérations attendues. Décider si un habitant donné est ami avec un autre est simple. Il suffit de regarder dans la liste d'amis du premier si le second apparaît. Cela nécessite au plus $n-1$ comparaisons si le liste n'est pas triée et $\log_2(n-1)$ si elle l'est.

   Mémoriser une nouvelle amitié est une tâche bien plus compliquée car les amis des amis sont des amis. Si h_1 et h_2 deviennent amis, il faut ajouter la liste de $h1$ à celle de $h2$ et ajouter la liste de $h2$ à celle de h_1 mais aussi mettre à jour les listes de tous les habitants référencés dans ces deux listes. Ajouter un habitant dans une liste a un coût linéaire ou constant selon que la liste est maintenu triée ou pas. Dans le pire des cas, le nombre total d'amitiés à ajouter est quadratique par rapport à n.

   Une solution alternative consiste à ne garder que des couples d'indices (h_1, h_2). On peut éviter l'information redondante en ne gardant que les couples où $h_1 < h_2$. Toutefois, le nombre maximal de couples est $n \times (n-1)/2$, rechercher un élément est linéaire sur cette taille et insérer une nouvelle amitié reste toujours aussi difficile.

   Les exercices suivants vont nous conduire progressivement à une solution bien plus économe en mémoire et en temps de calcul.

2. **Une structuration alternative**

   Une structuration des données bien connue pour ce problème permet d'obtenir des algorithmes de très faible complexité (quasi constante).

   Le principe général consiste à désigner au sein de chaque groupe d'amis, un représentant (unique) du groupe. Comme initialement personne n'a d'ami, chaque habitant est le représentant de son propre groupe.

   La structure de données est constituée d'un simple tableau (nommé *amis*) contenant une case par habitant. L'indice d'une case correspond au numéro de l'habitant représenté par la case.

   Dans chaque case du tableau est stocké l'indice d'une personne appartenant au même groupe d'amis (le représentant du groupe par exemple).

   Initialement, chaque habitant est seul dans son groupe et en est le représentant. Donnez le tableau dans sa configuration initiale pour un monde composé de 4 habitants. Donnez le pseudo-code de cette initialisation pour une population de n habitants.

   Solution: Le tableau initial pour une population de 4 habitants est le suivant

	0	1	2	3
amis	0	1	2	3

L'algorithme de l'initialisation est simple.

pour *i variant de 0 à* $n-1$ **faire**
  $amis[i] \leftarrow i$;
fin

3. **Premier algorithme**

   Un algorithme essentiel de cette structuration est savoir déterminer le représentant du groupe d'amis d'une personne donnée. Les représentants de groupe sont les seuls pour lesquels la valeur de la case est égale à son indice. Pour une personne donnée, trouver le représentant du groupe auquel elle appartient revient à suivre la suite d'indices formée par les cases jusqu'à trouver un représentant. Par exemple, considérons la situation suivante (avec une population de 6 habitants) :

	0	1	2	3	4	5
amis	0	0	1	3	0	3

   Déterminer le représentant du groupe auquel appartient 2 revient à parcourir les cases d'indice 2, 1 puis 0.

   De ce tableau, nous pouvons déduire que 0 est le représentant du groupe auquel appartiennent 1 et 4, que 2 a le même représentant que 1 (et donc que c'est 0 qui représente ce groupe) et que 3 est le représentant du groupe auquel appartient 5. On peut en conclure que $\{0, 1, 2, 4\}$ ainsi que $\{3, 5\}$ sont les deux groupes d'amis.

   Nous verrons par la suite comment nous assurer que le tableau ne contienne pas de chaîne d'indices formant un circuit non élémentaire (*i.e.* ne passant jamais par un représentant de groupe).

   Donnez le pseudo-code de l'algorithme déterminant le représentant d'un groupe auquel appartient une personne donnée.

Solution: Nous faisons l'hypothèse que le tableau d'amitiés se nomme *amis*, que l'habitant porte le numéro h et que le représentant de h doit être stocké dans r.

$r \leftarrow h$;
tant que $amis[r] \neq r$ **faire**
  $r \leftarrow amis[r]$;
fin

4. **Algorithme complémentaire**

   Deux personnes sont des amis si leurs groupes respectifs ont le même représentant. L'algorithme est immédiat en se basant sur la solution de l'exercice précédent.

   Lorsque deux personnes sont déclarées comme étant liées d'amitié alors que ce n'est pas encore le cas, le principe consiste à indiquer que le représentant du second groupe est à présent le représentant du premier groupe (ceci nous assure que le tableau ne contiendra jamais de circuit d'indice). Pour ce faire, il est suffisant d'affecter une seule case du tableau. Déterminez laquelle et donnez le pseudo-code de l'algorithme correspondant.

Solution: La seule affectation requise dans le tableau est la dernière instruction du pseudo-code qui suit.

```
r1 ← h1;
tant que amis[r1] ≠ r1 faire
    r1 ← amis[r1];
fin
r2 ← h2;
tant que amis[r2] ≠ r2 faire
    r2 ← amis[r2];
fin
amis[r1] ← r2;
```

5. **Complexité**

   Imaginons un monde de 10 personnes. À partir de la situation initiale, les personnes 0 et 1, puis 1 et 2, ..., jusqu'à 8 et 9 sont successivement déclarées amis. Donnez la situation finale du tableau et déduisez en la complexité dans le pire des cas des deux précédents algorithmes (cf. exercices 3 et 4).

   Solution:

	0	1	2	3	4	5	6	7	8	9
amis	1	2	3	4	5	6	7	8	9	9

   Le calcul du représentant de 0 conduit à parcourir toutes les cases du tableau. Comme les deux algorithmes reposent sur cette fonctionnalité, on peut en déduire que leur complexité dans le pire des cas est linéaire par rapport à la taille du tableau (la valeur de n).

6. **Première optimisation**

   L'idée est de profiter du calcul du représentant du groupe auquel appartient un habitant pour compresser le chemin d'indices parcouru. Cette compression permettra d'accélérer les futures recherches impliquant les habitants correspondants à ces indices.

   Le principe consiste à mettre à jour chaque case visitée avec le contenu de la future case visitée. Par exemple, dans le tableau illustrant l'exercice 3, le calcul du représentant de 2 conduit à visiter les cases 2, 1 et 0. Lors de ce parcours, il est facile d'affecter à *amis*[2] la valeur de *amis*[1] puis à *amis*[1] celle de *amis*[0]. Ainsi les recherches devant visiter la case d'indice 2 sont à présent plus rapides.

   Donnez la valeur du tableau de l'exercice 3 après la compression due au calcul du représentant de 2 ainsi que le pseudo-code de l'algorithme le réalisant.

   Solution: Avant compression, le tableau vaut :

	0	1	2	3	4	5
amis	0	0	1	3	0	3

   Après compression, nous obtenons :

	0	1	2	3	4	5
amis	0	0	0	3	0	3

   Le pseudo-code de l'algorithme intégrant la compression reste linéaire et n'emploie qu'une case mémoire de plus.

   ```
   r ← h;
   tant que amis[r] ≠ r faire
       tmp ← r;
       r ← amis[r];
       amis[tmp] ← amis[r];
   fin
   ```

7. **Une optimisation plus radicale**

Une façon plus coûteuse mais plus efficace en termes de compression consiste à calculer la valeur du représentant puis à affecter cette valeur à toutes les cases impliquées dans la recherche. Donnez le pseudo-code de l'algorithme intégrant cette nouvelle compression.

Solution:

$r \leftarrow h$;
tant que $amis\,[r] \neq r$ **faire**
  $r \leftarrow amis\,[r]$;
fin
$a \leftarrow h$;
tant que $amis\,[a] \neq a$ **faire**
  $tmp \leftarrow a$;
  $a \leftarrow amis\,[a]$;
  $amis\,[tmp] \leftarrow r$;
fin

Le problème plus général traité lors de cette séance est celui des partitions d'un ensemble. Une structure de données et des algorithmes très proches de ceux que nous avons étudiés permettent d'obtenir des très bons résultats de complexité. Les curieux pourront consulter la page suivante : `https://fr.wikipedia.org/wiki/Union-find`.