Algorithmique & Programmation (INF431)
Contrôle des connaissances CC1

22 avril 2013

La première partie concerne une implémentation de la structure de données Union-Find. La seconde partie utilise cette structure de données, mais est indépendante des détails de son implémentation ; elle peut donc en principe être traitée même si l'on n'a pas répondu aux questions de la première partie.

Dans tout le sujet, on demande d'écrire **non pas du pseudo-code**, mais **du code Java**.

Première partie

La structure de données Union-Find

La structure de données Union-Find permet de représenter des objets que nous appellerons *points* munis d'une *relation d'équivalence*. Nous emploierons le mot *groupe* pour désigner une classe d'équivalence. Les groupes forment donc une partition des points. La structure de données Union-Find distingue dans chaque groupe un point particulier, que nous appellerons le *représentant* de ce groupe. La structure de données Union-Find doit fournir quatre opérations :

1. Create() : créer un nouveau point, isolé dans un nouveau groupe ;
2. Find(x) : trouver le représentant du groupe auquel appartient le point x ;
3. Equivalent(x, y) : déterminer si les points x et y appartiennent ou non à un même groupe ;
4. Union(x, y) : sous l'hypothèse que les points x et y appartiennent à deux groupes distincts, fusionner ces groupes.

On souhaite implémenter en Java cette structure de données, sous forme d'une classe `UFPoint` dotée d'un constructeur et de trois méthodes `find`, `equivalent`, et `union`. Ce constructeur et ces méthodes correspondent respectivement aux opérations Create, Find, Equivalent, et Union. Le squelette de la classe `UFPoint` est donné dans la figure 1.

```
public class UFPoint {
  private UFPoint parent;
  public UFPoint () { parent = null; }
  public static UFPoint find (UFPoint x) { ... }
  public static boolean equivalent (UFPoint x, UFPoint y) { ... }
  public static void union (UFPoint x, UFPoint y) { ... }
}
```

Figure 1 – Squelette de la classe `UFPoint`

Chaque objet de classe `UFPoint` est doté d'un champ `parent` de type `UFPoint`. Ce champ contient donc soit `null`, soit l'adresse d'un point. Les points forment ainsi un graphe orienté $\mathcal{G}$, où il existe une arête du point `x` vers le point `y` si et seulement si `x.parent == y`. Un point a donc zéro ou une arête sortante. Le graphe $\mathcal{G}$ est consulté et potentiellement modifié par chacune des quatre opérations ci-dessus.

De plus, on adopte *l'invariant* suivant : « le graphe $\mathcal{G}$ est acyclique ». Chaque opération peut donc supposer cette propriété satisfaite, et doit la préserver. Parce que tout point a zéro ou une arête sortante, la propriété d'acyclicité peut s'exprimer sous la forme équivalente : « dans $\mathcal{G}$, à partir de tout point x, il est possible d'atteindre un point y dénué de successeur ». De plus, ce point y est déterminé de façon unique par le choix de x.

On considère que y est un représentant si et seulement si y n'a pas de successeur. On considère que y est le représentant de x si et seulement si y est un représentant et il existe un chemin de x à y. Tout point a donc un et un seul représentant. On considère que deux points appartiennent à un même groupe si et seulement si ils ont le même représentant.

Question 1 Implémentez la méthode `find`. Cette version de `find` ne doit pas modifier le graphe. Vous pouvez l'écrire sous forme itérative (c'est-à-dire à l'aide d'une boucle `while`) ou sous forme récursive. En prévision de la question 7, la seconde forme est conseillée. ◇

Question 2 Implémentez la méthode `equivalent`. On rappelle que les identités de deux objets peuvent être comparées à l'aide de l'opérateur `==`. ◇

Question 3 Implémentez la méthode `union`, de la façon la plus simple possible. ◇

Question 4 Montrez que votre implémentation de la méthode `union` préserve l'invariant : « le graphe $\mathcal{G}$ est acyclique ». ◇

Question 5 Montrez que la méthode `find` termine. Quelle est, dans le cas le pire, sa complexité en temps ? Exprimez-la en fonction du nombre total de points n. ◇

Cette implémentation de Union-Find est, dans le cas le pire, très inefficace. La question suivante vise à vérifier que sa complexité est quadratique.

Question 6 Étant donné un entier n arbitraire, exhibez une suite de $O(n)$ opérations Create et/ou Find et/ou Union qui, à partir d'un graphe $\mathcal{G}$ vide, ont une complexité en temps $\Omega(n^2)$, c'est-à-dire bornée inférieurement par n^2, à une constante près. Démontrez cela. ◇

Pour améliorer la complexité de la structure de données Union-Find, deux optimisations sont possibles : l'équilibrage (mentionné en cours) et la compression de chemins.

La compression de chemins repose sur une idée simple. Si la méthode Find(x_1) suit un chemin $x_1; x_2; \ldots; y$ à partir de x_1 pour découvrir que le représentant de x_1 est y, alors on voudrait que Find modifie le graphe de sorte que, à l'issue de l'appel à Find, il existe une arête de x_i à y pour chaque i.

Question 7 Modifiez la méthode `find` pour effectuer la compression de chemins. ◇

Deuxième partie

Unification

On s'intéresse maintenant à l'algorithme d'*unification*. Il s'agit d'un algorithme de résolution d'équations, qui a de nombreuses applications dans le domaine du raisonnement et de la preuve assistés par ordinateur. Cet algorithme utilise la structure de données Union-Find.

On se donne un nombre fini de *symboles* : par exemple, A, F. À chaque symbole S, on associe un entier positif ou nul, noté *arité*(S).

On se donne un nombre fini de *variables* : par exemple, t, u, v, w, x, y, z.

Un *terme* T est formé d'un symbole S et de n variables $x_1, \ldots, x_n$, où n est égal à *arité*(S). Ce terme est noté $S(x_1, \ldots, x_n)$. Par exemple, si *arité*$(A) = 0$ et si *arité*$(F) = 3$, alors $A()$ est un terme ; $F(x, y, z)$ est un terme ; $F(x, x, z)$ est un terme.

On considère des équations entre variables et termes, c'est-à-dire des équations de quatre formes possibles : variable = variable, variable = terme, terme = variable, et terme = terme. Un système d'équations est une conjonction d'équations. Par exemple, voici un système d'équations :

$$t = A() \qquad y = F(x, x, t) \qquad z = F(v, w, u) \qquad t = u \qquad y = z$$

Dans la suite, cet exemple sert d'illustration. Nous utilisons d'abord les trois premières équations ci-dessus pour illustrer comment un système d'équations est représenté en mémoire (questions 8 et 9). Ensuite, nous considérons l'ajout d'une équation entre variables. On verra que l'équation $t = u$ correspond à un cas simple traité dans les questions 10 et 11 ; l'équation $y = z$ correspond à un cas plus complexe traité dans les questions 12, 13 et 14. Les questions 15 et 16, qui traitent de la détection de cycles, sont indépendantes des questions 10 à 14.

Pour représenter en mémoire un système d'équations, on fait les deux choix suivants :

C1. On représente les variables $x, y, \ldots$ par des *points* d'Union-Find. Les variables sont ainsi organisées en groupes. On considère que deux variables x et y sont reliées par une équation $x = y$ si et seulement si x et y appartiennent à un même groupe.

C2. À chaque groupe, on associe *au plus une* équation de la forme $x = T$, où x est le représentant du groupe et T est un terme.

Pour transcrire ceci en Java, on pose :

J1. Conformément au point C1, une variable est un objet de classe `UFPoint`.

J2. Un symbole est une constante appartenant à l'énumération `Symbol` (figure 2).

J3. Un terme est un objet de classe `Term` (figure 2). Comme décrit plus haut, un terme `T` est constitué d'un symbole, `T.symbol`, et d'un tableau de variables, `T.children`. La longueur de ce tableau est toujours égale à l'arité du symbole `T.symbol`, qui n'est donc pas mémorisée explicitement.

J4. Conformément au point C2, on ajoute à la classe `UFPoint` un champ `term` (figure 3). Ce champ contient `null` pour indiquer l'absence d'équation, et contient un terme T pour indiquer la présence d'une équation $x = T$. **Le champ `x.term` n'est pertinent que si `x` est un représentant. Si `x` n'est pas un représentant, c'est-à-dire si `x.parent` est non `null`, alors `x.term` est `null`.**

Le terme associé à un groupe peut être consulté ou modifié via les méthodes `getTerm` et `setTerm` (figure 3). On souligne que **ces méthodes consultent ou modifient `find(this).term` et non pas `this.term`**.

Question 8 Le point J4 affirme : « si `x.parent` est non `null`, alors `x.term` est `null` ». Modifiez votre implémentation de la méthode `union` pour préserver cet invariant. Vous ferez cela de la façon la plus simple possible, quitte à effacer de l'information le cas échéant. ◇

Le contenu de la mémoire peut être considéré comme un graphe orienté. La figure 4 en donne un exemple, qui correspond aux trois premières équations du système ci-dessus. **Les sommets de ce graphe sont les variables et les termes, c'est-à-dire les objets de classe `UFPoint` et `Term`.** Un sommet qui représente une variable x a pour successeur :

- soit une variable y (si `x.parent == y`) ;
- soit un terme T (si `x.parent == null` et `x.term == T`) ;
- soit ni l'un ni l'autre (si `x.parent == null` et `x.term == null`).

```
public enum Symbol { A, F }

public class Term {
  // The symbol.
  public final Symbol symbol;
  // The children. (The length of this array is the arity of the symbol.)
  public final UFPoint[] children;
  // Constructor.
  public Term (Symbol symbol, UFPoint[] children)
  {
    this.symbol = symbol;
    this.children = children;
  }
}
```

FIGURE 2 – Les classes `Symbol` et `Term`

```
  // The term attached to this group.
  //   POSSIBLY NULL if there is no term attached to this group.
  // DEFINITELY NULL if this point has a parent.
  private Term term;
  // Accessor methods.
  public Term getTerm () { return find(this).term; }
  public void setTerm (Term term) { find(this).term = term; }
```

FIGURE 3 – Ajouts à la classe `UFPoint`

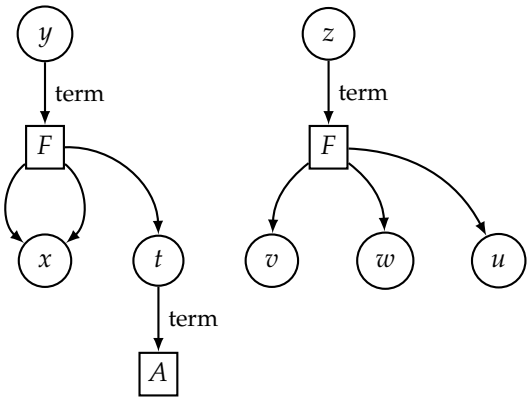

FIGURE 4 – Graphe correspondant aux équations $t = A()$ et $y = F(x, x, t)$ et $z = F(v, w, u)$

Un sommet qui représente un terme $S(x_1, \ldots, x_n)$ a pour successeurs les variables $x_1, \ldots, x_n$. Pour clarifier le dessin, une arête d'une variable x à une variable y est étiquetée « parent », tandis qu'une arête de x à un terme T est étiquetée « term ». (La figure 4 ne contient pas d'arête étiquetée « parent ».) Une arête d'un terme vers une variable n'est pas étiquetée. On note que, en général, **ce graphe peut contenir des cycles.**

Question 9 Les variables et les termes se construisent à l'aide de l'instruction `new` de Java :

```
UFPoint x = new UFPoint ();
UFPoint t = new UFPoint ();
Term A    = new Term (Symbol.A, new UFPoint[] {});
Term Fxxt = new Term (Symbol.F, new UFPoint[] { x, x, t});
```

Complétez ce code pour construire des objets de classe `UFPoint` et `Term` qui correspondent au graphe de la figure 4. ◇

On se place dans une nouvelle classe `Unify`. On souhaite d'abord y définir une méthode `unifyVariables` (figure 5). Cette méthode attend deux variables `x1` et `x2` et doit modifier le graphe de façon à ajouter au système la nouvelle équation $x_1 = x_2$. Elle ne renvoie pas de résultat.

Au moment où `unifyVariables(x1, x2)` est appelée, il se peut que le système contienne déjà une équation de la forme $x_1 = T_1$ et/ou une équation de la forme $x_2 = T_2$. (Autrement dit, `x1.getTerm()` et `x2.getTerm()` peuvent être `null` ou non.) Si tel est le cas, il faut faire en sorte que ces équations ne soient pas perdues, mais soient conservées. Pour le moment (question 10), nous supposons que nous avons au plus une équation $x_1 = T_1$ **ou** une équation $x_2 = T_2$, mais pas les deux. Plus loin (question 13), nous traitons le cas où ces deux équations sont présentes simultanément.

On rappelle que l'appel `union(x1, x2)` exige que `x1` et `x2` appartiennent à deux groupes distincts, et peut en principe installer un pointeur `parent` soit du représentant de `x1` vers le représentant de `x2`, soit l'inverse. On considérera donc qu'on ne sait pas dans quelle direction ce pointeur est installé.

Les questions 10 et 11 sont indépendantes.

Question 10 Implémentez la méthode `unifyVariables`. On souligne à nouveau que les termes `x1.getTerm()` et `x2.getTerm()` peuvent être `null` ou non, ce qui donne lieu à quatre cas. Pour le moment, dans le cas où tous deux sont non `null`, vous lancerez une exception, via `throw new Error ()`. Ce cas sera traité plus loin (question 13). ◇

Question 11 Dessinez le graphe obtenu lorsque, à partir de l'état décrit par le graphe de la figure 4, on appelle `unifyVariables(t, u)` pour ajouter au système l'équation $t = u$. Deux dessins sont possibles, suivant la direction dans laquelle le pointeur `parent` est installé. Vous en choisirez un. ◇

Lorsque l'algorithme d'unification rencontre une équation de la forme terme = terme, on souhaite qu'il se comporte de la façon suivante :

C3. Une équation de la forme $S(x_1, \ldots, x_n) = S(y_1, \ldots, y_n)$ est *décomposée*, c'est-à-dire remplacée par les équations $x_i = y_i$, pour $i \in \{1, \ldots, n\}$.

C4. Une équation de la forme $S_1(\ldots) = S_2(\ldots)$, où S_1 et S_2 sont deux symboles distincts, provoque un échec de l'algorithme.

On souhaite maintenant écrire dans la classe `Unify` une méthode `unifyTerms` (figure 5). Cette méthode attend deux termes (non nuls) `T1` et `T2` et ajoute au système la nouvelle équation $T_1 = T_2$ **en obéissant aux points C3 et C4**. Elle ne renvoie pas de résultat mais peut lancer l'exception `Unsatisfiable` (figure 5) pour signaler l'échec de l'algorithme.

Question 12 À l'aide de la méthode `unifyVariables` et **sans utiliser l'instruction `new` de Java,** implémentez la méthode `unifyTerms`. On ne demande pas ici de modifier `unifyVariables` : ce sera l'objet de la question 13. ◇

On revient à présent au cas qui a été laissé de côté lors de la question 10. On remarque que, si on a deux équations portant sur une même variable, disons $x = T_1$ et $x = T_2$, alors on peut les remplacer par les équations équivalentes $x = T_1$ et $T_1 = T_2$.

Question 13 À l'aide de la méthode `unifyTerms`, complétez votre implémentation de la méthode `unifyVariables` pour traiter le cas où les termes `x1.getTerm()` et `x2.getTerm()` sont tous deux non `null`. ◇

On rappelle que, en général, le graphe formé par les variables et les termes peut être cyclique.

Question 14 Démontrez que les méthodes `unifyVariables` et `unifyTerm` terminent. ◇

Question 15 Si au système d'équations de la figure 4 on ajoutait l'équation $y = x$, en appelant `unifyVariables(y, x)`, quel nouveau graphe obtiendrait-on ? ◇

On souhaite écrire une méthode `detectCycle` (figure 5) qui attend une liste de toutes les variables et détermine si le graphe contient ou non un cycle. Cette méthode ne renvoie aucun résultat. Si le graphe contient un cycle, elle lance l'exception `Unsatisfiable`.

Question 16 Implémentez la méthode `detectCycle`. Afin de marquer certains sommets du graphe, vous pouvez employer les classes `HashSet` ou `HashMap` de la librairie Java. Vous pouvez considérer que la définition de ces classes est celle (simplifiée) donnée par la figure 6. ◇

```
public class Unify {
  public static void unifyVariables (UFPoint x1, UFPoint x2)
    throws Unsatisfiable { ... }
  public static void unifyTerms (Term T1, Term T2)
    throws Unsatisfiable { ... }
  public static void detectCycle (List<UFPoint> points)
    throws Unsatisfiable { ... }
}
public class Unsatisfiable extends Exception {}
```

FIGURE 5 – Squelette de la classe `Unify`

```
public class HashSet<E> {
  public HashSet ();
  public void add (E e);
  public void remove (E e);
  public boolean contains (E e);
}
```

```
public class HashMap<K, V> {
  public HashMap ();
  public void put (K key, V value);
  public void remove (K key);
  public V get (K key);
} // get returns null if key is absent
```

FIGURE 6 – Les classes `java.util.HashSet` et `java.util.HashMap`