Algorithmique & Programmation (INF431)
Contrôle des connaissances CC1
CORRIGÉ

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.

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

```
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`

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. ◇

Solution. À partir du point `x` donné, il faut suivre l'unique chemin déterminé par les champs `parent`, jusqu'à atteindre un point dénué de successeur. Ce point est la racine de l'arbre auquel appartient `x`, donc le représentant de `x`. Le code peut s'écrire sous la forme itérative suivante :

```
public static UFPoint find (UFPoint x)
{
  while (x.parent != null)
    x = x.parent;
  return x;
}
```

Le code peut également s'écrire sous forme récursive :

```
public static UFPoint find (UFPoint x)
{
  if (x.parent != null)
    return find(x.parent);
  else
    return x;
}
```

En Java, la forme itérative est compilée de façon plus efficace. Cependant, la forme récursive sera rendue nécessaire par la compression de chemins (question 7). □

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

Solution. Deux points appartiennent à un même groupe si et seulement si leurs représentants sont les mêmes :

```
public static boolean equivalent (UFPoint x, UFPoint y)
{
  return find(x) == find(y);
}
```

Le test d'égalité des représentants se fait via l'opérateur `==`, qui dénote la comparaison des adresses. On pourrait également utiliser la méthode `equals`, héritée de la classe `Object`, car celle-ci effectue elle-même un test `==`. □

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

Solution. On travaille ici sous l'hypothèse que `x` et `y` appartiennent à deux groupes distincts. En d'autres termes, cette propriété est une *précondition* de la méthode `union`.

À l'aide de la méthode `find`, on se ramène d'abord au cas où `x` et `y` sont les représentants de leurs groupes respectifs. On suppose donc `x` et `y` distincts, hypothèse que l'on peut (si on le souhaite) matérialiser, dans le code, à l'aide de l'instruction **`assert (x != y)`** :

```
public static void union (UFPoint x, UFPoint y)
{
  x = find(x);
  y = find(y);
  assert (x != y);
  x.parent = y;
}
```

Ensuite, il ne reste qu'à fusionner les deux groupes, à l'aide de l'instruction `x.parent = y`, ou bien, symétriquement, `y.parent = x`. Le choix entre ces deux possibilités est arbitraire. (Il pourrait être guidé par des considérations d'équilibrage.)

Notons qu'on pouvait donner une solution plus compacte, à savoir :

```
find(x).parent = find(y);
```

Cette écriture est équivalente à la précédente (sans l'instruction **`assert`**). □

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

Solution. Comme le signale l'énoncé, l'invariant peut s'exprimer sous la forme équivalente : « à partir de tout point x il est possible d'atteindre un point y dénué de successeur ». Nous pouvons donc supposer cette propriété vraie avant l'appel `union(x, y)`, et devons démontrer qu'elle est toujours satisfaite après cet appel.

Les deux premières lignes de la méthode `union` appellent `find`, qui ne modifie pas le graphe, donc ne brise pas l'invariant. Une fois ces deux appels effectués, nous sommes ramenés à une situation où `x` et `y` sont leurs propres représentants respectifs (i.e., n'ont pas de successeur), et sont distincts.

Il reste donc à vérifier que l'instruction `x.parent = y` préserve l'invariant. Pour plus de clarté, notons $\mathcal{G}$ le graphe avant cette instruction et $\mathcal{G}'$ le graphe modifié par l'exécution de cette instruction. Dans $\mathcal{G}$, `x` n'a pas de successeur ; dans $\mathcal{G}'$, son sucesseur est `y`. En dehors de cela, $\mathcal{G}$ et $\mathcal{G}'$ sont identiques. $\mathcal{G}$ est donc inclus dans $\mathcal{G}'$.

Considérons maintenant un sommet z quelconque, et montrons qu'à partir de z il existe dans $\mathcal{G}'$ un chemin qui mène à un sommet dénué de successeur. Nous savons que dans $\mathcal{G}$ il existe un tel chemin ; soit r son extrémité. Puisque $\mathcal{G}$ est inclus dans $\mathcal{G}'$, ce chemin de z à r existe toujours dans $\mathcal{G}'$. Deux cas se présentent maintenant. Si r est distinct de x, alors r est toujours dénué de successeur dans $\mathcal{G}'$, et nous sommes satisfaits. Si r est `x`, alors nous avons dans $\mathcal{G}'$ un chemin de z à `x`, puis une arête de `x` à `y`, donc un chemin de z à `y`. Puisque `y` est dénué de successeur dans $\mathcal{G}'$, le but est atteint. □

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. ◇

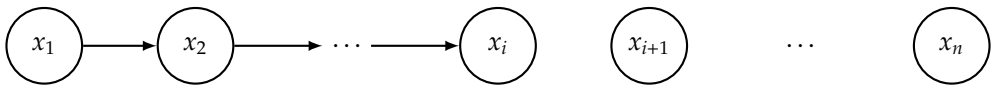
Solution. Supposons que la méthode `find` ne termine pas. Alors, c'est la boucle `while` qui ne termine pas. Puisque, à chaque itération, on passe d'un point `x` à son successeur `x.parent`, il existe donc dans le graphe $\mathcal{G}$ un chemin infini. Puisque ce graphe est fini, il existe donc dans le graphe un cycle. Or, nous savons que le graphe est acyclique ; cette propriété est un invariant. Contradiction.

Une itération de la boucle `while` est exécutée en temps $O(1)$. Le nombre d'itérations de cette boucle est borné par la longueur maximale d'un chemin dans $\mathcal{G}$. Puisque ce graphe comporte n sommets, cette longueur est elle-même bornée par n. Donc, la complexité de `find` est $O(n)$. □

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. ◇

Solution. À l'aide de Create, on crée d'abord n points, que l'on nomme $x_1, \ldots, x_n$. Puis, pour i variant de 1 à $n-1$, on appelle Union(x_1, x_{i+1}). Nous prétendons alors que le graphe obtenu avant l'itération i est le suivant :



Nous prétendons de plus que le coût des opérations Union effectuées avant l'itération i est au moins $i(i-1)/2$, où l'on compte un coût 1 à chaque fois que l'on consulte un pointeur `parent`.

Pour démontrer cette propriété, on vérifie d'abord qu'elle est vraie avant la première itération (i.e., pour $i = 1$), ensuite qu'elle est préservée à chaque itération (i.e., si elle est vraie au rang i, alors elle est vraie au rang $i+1$).

Pour $i = 1$, le résultat est immédiat. Les n points sont isolés, donc le graphe a bien la forme ci-dessus ; et le coût des unions effectuées jusqu'ici est nul.

Supposons à présent que le graphe a la forme décrite par le schéma ci-dessus, au rang i, et que l'on s'apprête à exécuter l'opération Union(x_1, x_{i+1}). Notre implémentation de la méthode `union` appelle d'abord Find(x_1) et Find(x_{i+1}). Le premier de ces appels consulte les pointeurs `parent` de tous les points $x_1, x_2, \ldots, x_i$, donc, à lui seul, a un coût i. Ces appels renvoient respectivement x_i et x_{i+1}, ce qui provoque ensuite l'installation d'une arête de x_i vers x_{i+1}. Une fois l'opération Union(x_1, x_{i+1}) terminée, le graphe a bien la forme annoncée, au rang $i+1$, et le coût total des appels précédents à Union est au moins $i(i-1)/2 + i$, soit $(i+1)i/2$.

Nous avons donc exhibé une suite de n opérations Create et n opérations Union dont le coût total est d'au moins $n(n+1)/2$ opérations élémentaires d'accès à la mémoire. De façon plus abstraite, on peut donc affirmer que nous avons exhibé une suite de $O(n)$ opérations dont la complexité en temps est $\Omega(n^2)$. C'est ce qui était demandé. □

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. ◇

Solution. Comme l'identité de y n'est pas connue avant que l'on ait atteint la fin du chemin issu de x, on ne peut pas installer ces arêtes au fur et à mesure que l'on *descend* le long du chemin. Il faut le faire *au retour*. La forme itérative de la méthode FIND n'est donc plus adaptée ; nous nous appuyons ici sur la forme récursive de cette méthode, donnée lors de la réponse à la question 1.

La modification est extrêmement simple : après l'appel récursif à `find(x.parent)` et avant de transmettre le résultat, il faut mettre à jour le champ `x.parent`. En Java, on peut écrire :

```
public static UFPoint find (UFPoint x)
{
  if (x.parent != null)
    return x.parent = find(x.parent);
  else
    return x;
}
```

Il faut comprendre que `x.parent = find(x.parent)` n'est pas une expression de comparaison (l'opérateur de comparaison s'écrit `==`) mais une instruction d'écriture dans le champ `x.parent`. On aurait pu remplacer la ligne `return x.parent = find(x.parent);` par les lignes suivantes :

```
  {
    UFPoint y = find(x.parent);
    x.parent = y;
    return y;
  }
```

C'est équivalent. □

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. ◇

Solution. Il suffit pour cela d'écrire la valeur `null` dans le champ `term` au moment où le champ `parent` prend une valeur non triviale.

```
public static void union (UFPoint x, UFPoint y)
{
  x = find(x);
  y = find(y);
  assert (x != y);
  x.parent = y;
  x.term = null;
}
```

Il est inutile de sauvegarder la valeur précédemment contenue dans `x.term`. L'utilisateur s'en chargera, si nécessaire, en appelant `x.getTerm()` avant d'appeler `union(x, y)`.

Ceux qui avaient employé l'écriture compacte lors de la question 3 devaient faire attention. Ici, la proposition suivante :

```
find(x).term = null;
find(x).parent = find(y);
```

```
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`

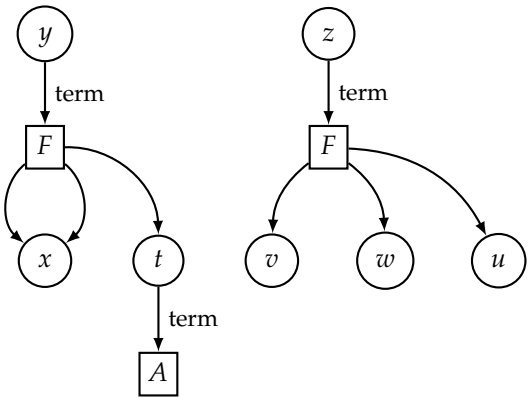


FIGURE 4 – Graphe correspondant aux équations $t = A()$ et $y = F(x, x, t)$ et $z = F(v, w, u)$

était inefficace car `find(x)` y est calculé deux fois. Pire, la proposition suivante :

```
find(x).parent = find(y);
find(x).term = null;
```

était incorrecte. La première instruction modifie le représentant de `x`, donc la seconde expression `find(x)` ne produit pas la même valeur que la première. □

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`).

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. ◇

Solution. On construit les autres variables, ainsi que le terme $F(v, w, u)$, sur le même modèle :

```
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});
UFPoint y = new UFPoint ();
UFPoint z = new UFPoint ();
UFPoint v = new UFPoint ();
UFPoint w = new UFPoint ();
UFPoint u = new UFPoint ();
Term Fvwu = new Term (Symbol.F, new UFPoint[] { v, w, u});
t.setTerm(A);
y.setTerm(Fxxt);
z.setTerm(Fvwu);
```

Une fois les variables et les termes construits, trois appels à `setTerm` permettent de mettre en place les trois équations souhaitées, qui toutes trois sont de la forme variable = terme. □

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). ◇

Solution. L'implémentation de la méthode `unifyVariables` est donnée par la figure 7. On vérifie d'abord si `x1` et `x2` appartiennent au même groupe. Si c'est le cas, il n'y a rien à faire : l'équation $x_1 = x_2$ était déjà connue. Sinon, on continue.

Deux appels à `getTerm` nous donnent les termes `T1` et `T2` portés respectivement par (les représentants de) `x1` et `x2` ; ces termes peuvent être égaux à `null`. Si `T1` est non `null`, cela signifie qu'on a l'équation $x_1 = T_1$; et symétriquement. Notons qu'il faut effectuer ces appels à `getTerm` avant l'instruction `union(x1, x2)`, sans quoi l'un des deux termes sera perdu.

Un appel à `union(x1, x2)` fusionne les groupes représentés par `x1` et `x2` en installant un pointeur `parent` du représentant de `x1` vers celui de `x2`, ou l'inverse (nous ne souhaitons pas faire d'hypothèse à ce sujet). Après cette fusion, il faut faire en sorte de ne pas perdre l'équation $x_1 = T_1$ ou $x_2 = T_2$. Pour cela, on utilise un appel à `x1.setTerm` ou `x2.setTerm` (c'est équivalent). Si `T1` est `null`, on conserve le terme `T2`, et vice-versa.

Reste le cas où `T1` et `T2` sont tous deux non `null`, que l'on ne demandait pas de traiter pour le moment.

Soulignons que l'on n'a pas fait l'hypothèse que `x1` et `x2` sont les représentants de leur groupe. Toutes les méthodes que nous avons utilisées ici (`equivalent`, `getTerm`, `union`, `setTerm`) se comportent correctement même si ce n'est pas le cas. □

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. ◇

Solution. Deux réponses sont possibles, selon que l'on ajoute une arête `parent` de t vers u ou l'inverse. La figure 9 présente la situation obtenue si l'on ajoute une arête `parent` de u vers t. Dans ce cas, t doit conserver son champ `term`, qui pointe vers le terme $A()$. On aurait pu ajouter une arête `parent` de t vers u ; dans ce cas, `t.term` aurait pris la valeur `null`, et `u.term` aurait été modifié pour pointer vers le terme $A()$. □

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. ◇

Solution. L'énoncé de cette question contenait une erreur. La restriction « sans utiliser l'instruction **`new`** de Java » était censée imposer le fait qu'on ne devait pas créer de nouveaux points via `new UFPoint (...)`. Cependant, lorsque les termes `T1` et `T2` portaient des symboles distincts, il fallait créer et lancer une exception via `throw new Unsatisfiable ()`. Les élèves qui ont écrit qu'ils ne savaient pas créer une exception sans utiliser **`new`** n'ont pas été sanctionnés.

La solution apparaît dans la figure 7. On vérifie d'abord que les termes `T1` et `T2` ont le même symbole. Si ce n'est pas le cas, un échec est signalé. Si le symbole, disons S, est bien le même des deux côtés, alors on appelle n fois la méthode `unifyVariables`, où n est l'arité de S, pour ajouter au système les équations notées $x_i = y_i$ dans le point C3. On peut supposer que les termes ont été correctement construits, donc que les tableaux `T1.children` et `T2.children` ont la même longueur, à savoir n. □

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`**. ◇

Solution. La solution apparaît dans la figure 7. Dans le dernier cas, où T_1 et T_2 sont deux termes non **`null`**, il suffit d'appeler `unifyTerms` pour ajouter au système l'équation $T_1 = T_2$. Aucun appel à `x1.setTerm` n'est nécessaire ici, car le représentant de `x1` et `x2` porte déjà un terme approprié (ce terme est soit T_1, soit T_2, peu importe lequel). □

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. ◇

```
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`

Solution. À première vue, il n'est pas évident que ces méthodes terminent, puisqu'elles sont mutuellement récursives. Heureusement, il existe un argument de terminaison simple. En bref, l'argument est : « le nombre de classes d'équivalence diminue strictement ». Plus précisément, on remarque que :

1. ni `unifyVariables` ni `unifyTerm` ne créent de nouvelle variable ; donc, pendant l'exécution de ces méthodes, le nombre de variables est fixe ;
2. pour que ces méthodes ne terminent pas, il faudrait qu'il existe une exécution le long de laquelle, infiniment souvent, `unifyVariables` appelle `unifyTerms` ;
3. or, à chaque fois que `unifyVariables` appelle `unifyTerms`, cet appel est précédé d'un appel à `union`, qui a pour effet de faire diminuer strictement le nombre de classes d'équivalence.

On voit qu'une telle exécution infinie ne peut exister. Donc, l'algorithme termine. □

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 ? ◇

Solution. Une réponse possible est donnée par la figure 10. Il existe une autre réponse, où l'arête `parent` est orientée de y vers x. Dans les deux cas, le graphe présente un cycle. □

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. ◇

Solution. Une solution est donnée dans la figure 8. On utilise un parcours en profondeur d'abord. S'il existe un cycle, ce cycle passe nécessairement par une variable ; il suffit donc, pour parcourir le graphe, de marquer les variables. On sait que, afin de détecter la présence d'un cycle, il faut trois marques : « encore non découvert », « découvert et en cours de traitement », « découvert et traité ». On choisit ici de coder ces trois valeurs à l'aide de deux ensembles : `discovered` représente l'ensemble des sommets découverts et en cours de traitement ; `completed` représente l'ensemble des sommets découverts et traités. Un sommet encore non découvert n'appartient à aucun de ces ensembles.

La méthode principale, `detectCycle`, initialise les deux ensembles, qui sont initialement vides, puis appelle `visitPoint` sur chacun des sommets du graphe.

La méthode récursive `visitPoint` se ramène d'abord au cas où `x` est le représentant de son groupe. Ainsi, c'est toujours le représentant du groupe qui appartient (ou non) aux ensembles `discovered` et `completed`. En d'autres termes, on ne marque pas les variables, mais les groupes. La méthode `visitPoint` teste ensuite l'appartenance de `x` aux deux ensembles : si `x` appartient à `discovered`, un cycle a été découvert ; si `x` a déjà été entièrement traité, il est inutile de le traiter à nouveau. Si `x` est nouvellement découvert, alors on marque `x` comme tel, puis on traite le terme `x.term`, puis on marque `x` comme traité.

Pour traiter le terme `x.term`, on vérifie s'il est non `null`, et si oui, on visite successivement chacune des variables qui apparaissent dans ce terme. □

```
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`

```
public static void unifyVariables (UFPoint x1, UFPoint x2) throws Unsatisfiable
{
  if (UFPoint.equivalent(x1, x2))
    return;
  Term T1 = x1.getTerm();
  Term T2 = x2.getTerm();
  UFPoint.union(x1, x2);
  if (T1 == null)
    x1.setTerm(T2);
  else if (T2 == null)
    x1.setTerm(T1);
  else
    // throw new Error (); // réponse provisoire à la question 10
    unifyTerms(T1, T2);
}

public static void unifyTerms (Term T1, Term T2) throws Unsatisfiable
{
  if (T1.symbol != T2.symbol)
    throw new Unsatisfiable ();
  assert (T1.children.length == T2.children.length);
  for (int i = 0; i < T1.children.length; i++)
    unifyVariables(T1.children[i], T2.children[i]);
}
```

FIGURE 7 – L'algorithme d'unification : `unifyVariables` et `unifyTerms`

```
public static void detectCycle (List<UFPoint> points) throws Unsatisfiable
{
  HashSet<UFPoint> discovered = new HashSet<UFPoint> ();
  HashSet<UFPoint> completed = new HashSet<UFPoint> ();
  for (UFPoint point : points)
    visitPoint(discovered, completed, point);
}

private static void visitPoint (
  HashSet<UFPoint> discovered, HashSet<UFPoint> completed, UFPoint x
) throws Unsatisfiable
{
  x = UFPoint.find(x);
  if (discovered.contains(x))
    throw new Unsatisfiable ();
  if (completed.contains(x))
    return;
  discovered.add(x);
  Term T = x.getTerm();
  if (T != null)
    for (UFPoint y : T.children)
      visitPoint(discovered, completed, y);
  discovered.remove(x);
  completed.add(x);
}
```

FIGURE 8 – Détection de cycles

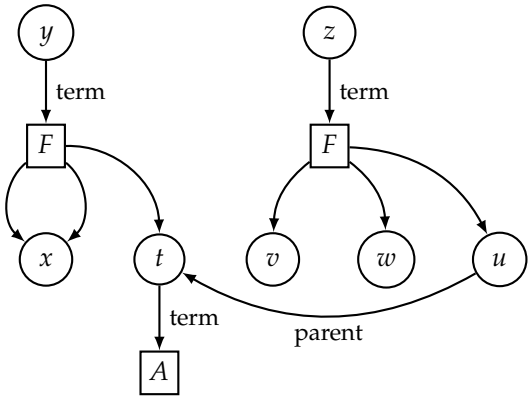


FIGURE 9 – Graphe après appel à `unifyVariables`(t, u)

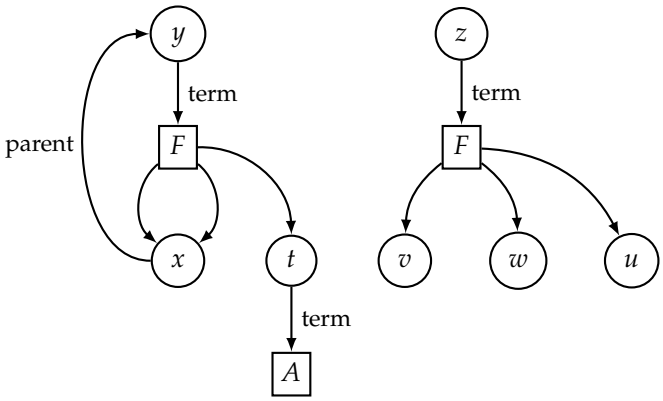


FIGURE 10 – Graphe après appel à `unifyVariables`(y, x)