Programmation Avancée (INF441)
Contrôle classant
CORRIGÉ

9 juin 2015

Solution de la question 1 Les sommets du graphe étant représentés par des entiers, le graphe doit implémenter l'interface `ISuccessors<Integer>`. Il faut donc ajouter en tête de la définition de la classe la déclaration `implements ISuccessors<Integer>`.

Notons qu'on ne peut pas écrire `ISuccessors<int>`. En effet, comme on l'a vu en cours, en Java, une variable de types ne peut pas être instanciée par un type primitif ; elle ne peut être instanciée que par un type d'objet.

Pour justifier l'affirmation que `AdjacencyGraph` implémente `ISuccessors<Integer>`, il faut lui ajouter une méthode `successors`, dont l'argument est un sommet `i` de type `Integer` et dont le résultat doit être de type `Iterable<Integer>`. Heureusement, c'est très simple :

```
public Iterable<Integer> successors (Integer i) {
  return successors.get(i);
}
```

L'expression `successors.get(i)` donne une liste de type `LinkedList<Integer>`, qui (par sous-typage) a également le type `Iterable<Integer>`.

Le fait que le champ `successors` et la méthode `successors` portent le même nom est une coïncidence qui en principe ne pose aucun problème, car Java autorise cela. Toutefois, pour plus de clarté, le sujet aurait pu imposer des noms distincts. □

Solution de la question 2 Comme lors de la question précédente, la classe `MatrixGraph` doit implémenter l'interface `ISuccessors<Integer>`. Pour cela, il faut lui ajouter une méthode `successors`.

```
public Iterable<Integer> successors (Integer i) {
  ... // à compléter
}
```

Cette fois, nous n'avons pas d'objet de type `Iterable<Integer>` à portée de main, donc nous devons en construire un de toutes pièces. Heureusement, parce que `Iterable` est une interface fonctionnelle (c'est-à-dire dotée d'une seule méthode), la syntaxe des clôtures de Java 8 nous permet de faire cela simplement en écrivant `() -> { ... }` où les points de suspension constituent le corps de la méthode `iterator()`. En fait, comme le corps de cette méthode sera constitué d'une seule instruction `return ...;` on peut omettre les accolades et le mot-clef `return` et écrire `() -> ...`.

```
public Iterable<Integer> successors (Integer i) {
  return () -> ...; // à compléter
}
```

Le corps de la méthode `iterator()` doit construire et renvoyer un (nouvel) itérateur, capable d'énumérer les successeurs du sommet `i`. À nouveau, il faut le construire de toutes pièces. Cette fois, le mieux est d'employer une classe anonyme : on écrit **`new Iterator<Integer> () { ... }`** où les points de suspension représentent le contenu de la classe.

```
public Iterable<Integer> successors (Integer i) {
  return () -> new Iterator<Integer> () {
    ... // à compléter
  };
}
```

Cette classe anonyme doit être dotée de méthodes `hasNext` et `next`. Elle doit également être dotée d'un état interne modifiable, puisque l'itérateur doit savoir à tout moment quel est le « prochain » successeur à examiner. On prévoit donc un champ modifiable `j`. À tout instant, les successeurs dont l'indice est inférieur à `j` ont déjà été examinés ; ceux dont l'indice est supérieur ou égal à `j` restent à examiner.

Il est alors facile d'implémenter les méthodes `hasNext` et `next`. La solution complète est donnée dans la figure 1.

Notre méthode `hasNext` incrémente `j` jusqu'à atteindre le prochain successeur de `i`, s'il en existe un. Elle renvoie alors cette information. Si à l'issue de la boucle `j < n` est vrai, alors `j` est le prochain successeur ; sinon, `j` vaut `n` et tous les successeurs ont été examinés.

Notre méthode `next` appelle d'abord `hasNext`, ce qui a pour double effet de déterminer s'il existe un prochain successeur, et si oui, de modifier `j` pour contenir l'indice de ce prochain successeur. Donc, si `hasNext` renvoie **`true`**, alors il faut renvoyer la valeur actuelle de `j`, et il faut aussi incrémenter `j`, afin de mémoriser le fait que ce successeur a été examiné. Si au contraire `hasNext` renvoie **`false`**, alors l'énumération est terminée, et il faut lancer l'exception `NoSuchElementException`.

Il faut bien noter que l'utilisateur de notre itérateur peut choisir d'appeler `hasNext` zéro fois, une fois, ou plusieurs fois, avant d'appeler `next`. Une solution qui supposerait (par exemple) que `hasNext` est appelée au moins une fois avant `next` est incorrecte.

Certains élèves ont répondu à cette question en construisant d'abord une liste des successeurs du sommet `i`, puis en renvoyant cette liste, comme lors de la question 1. Cette solution est correcte au sens où elle permet bien d'itérer sur les successeurs. Toutefois, elle exige un espace $O(n)$, alors que notre solution exige un espace $O(1)$. De plus, elle consomme un temps $O(n)$ avant même que le client exige le premier élément de la liste, alors que notre solution effectue le calcul à la demande. Elle a été moins bien notée. Une remarque analogue s'applique à la question 3. □

Solution de la question 3 Une solution complète est donnée dans la figure 2.

L'état interne de notre itérateur est constitué ici de la pile, `stack`, et de l'ensemble des sommets marqués, `visited`. On insère initialement dans la pile le sommet `root`.

Ici, `stack` et `visited` sont des variables locales de la clôture qui construit l'itérateur. L'itérateur y a donc accès. Une autre possibilité serait de déplacer les trois lignes qui définissent `stack` et `visited` au-delà de la ligne **`return new`** `Iterator<V>`. Dans ce cas, `stack` et `visited` seraient des champs de l'itérateur. L'instruction `stack.push(root);` devrait alors être placée entre accolades et deviendrait un « *initializer* ».

La méthode `hasNext` retire de la tête de la pile tout sommet déjà marqué. Cela fait, soit la pile est vide, auquel cas l'énumération est terminée ; soit la pile est non vide, auquel cas on a un sommet non marqué en tête de pile.

La méthode `next` appelle d'abord `hasNext`, ce qui garantit, si `hasNext` renvoie **`true`**, que l'on a un sommet non marqué en tête de pile. On dépile ce sommet `v` et on le marque. On ajoute tous ses successeurs à la pile. (Une simple boucle **`for`** est possible ici puisque `g.successors(v)` est de type `Iterable<V>`.) On peut alors renvoyer `v`.

```
public Iterable<Integer> successors (Integer i) {
  return () -> new Iterator<Integer> () {
    private int j = 0;
    public boolean hasNext () {
      while (j < n && !matrix[i][j])
        j++;
      return j < n;
    }
    public Integer next () {
      if (hasNext()) return j++;
      throw new NoSuchElementException ();
    }
  };
}
```

FIGURE 1 – Solution de la question 2

```
public class Traversal<V> {
  public Iterable<V> traverse (ISuccessors<V> g, V root) {
    return () -> {

      // The internal state of the iterator: a stack and a set of
      // visited vertices.
      Stack<V> stack = new Stack<V> ();
      stack.push(root);
      HashSet<V> visited = new HashSet<V> ();

      return new Iterator<V> () {

        // hasNext ensures we have an unmarked vertex on top of the stack
        // (or the stack is empty).
        public boolean hasNext () {
          while (!stack.empty() && visited.contains(stack.peek()))
            stack.pop();
          return !stack.empty();
        }

        // next returns a vertex v and at the same time pushes the
        // successors of v onto the stack for later examination.
        public V next() {
          if (hasNext()) {
            V v = stack.pop();
            visited.add(v);
            for (V w : g.successors(v))
              stack.push(w);
            return v;
          }
          throw new NoSuchElementException ();
        }

      };
    };
  }
}
```

FIGURE 2 – Solution de la question 3

Nous avons choisi une variante où un sommet `v` est marqué au moment où il est retiré de la pile. Les successeurs de `v` sont alors insérés dans la pile sans aucun test. La pile peut alors contenir plusieurs fois un même sommet. On peut préférer une autre variante, où un sommet `v` est marqué au moment où il est inséré dans la pile, et n'est inséré que s'il n'est pas déjà marqué. La pile ne contient alors pas de doublons. La méthode `hasNext` se simplifie alors : il lui suffit de tester si la pile est vide.

Il faut souligner qu'une nouvelle pile `stack` et un nouvel ensemble `visited` doivent être créés à chaque fois qu'un nouvel itérateur est créé. Les trois lignes qui initialisent `stack` et `visited` ne doivent donc pas être situées dans la méthode `traverse`. Nous les avons situées dans le corps de la clôture `() -> { ... }`. On aurait pu également les situer dans l'itérateur, c'est-à-dire après la ligne `return new Iterator <V > () {`. Dans ce cas, l'instruction `stack.push(root);` devrait être placée entre accolades. C'est ce que Java appelle un « *initializer* ». C'est, en gros, un constructeur anonyme pour une classe anonyme.

Certains élèves ont répondu à cette question en effectuant immédiatement (dans la méthode `traverse`) la totalité du parcours de graphe, en construisant une liste des sommets découverts, puis en renvoyant cette liste, qui implémente l'interface `Iterable<V>`. Cette solution peut être considérée comme correcte, si les sommets apparaissent bien dans la liste dans l'ordre où ils ont été découverts. Toutefois, elle consomme un temps et un espace $O(n)$ avant même que le client exige le premier élément de la liste, alors que notre solution effectue la traversée du graphe de façon incrémentale, au fur et à mesure que les sommets sont exigés par le client, et consomme en général moins d'espace. Elle a été moins bien notée. □

Solution de la question 4 La fonction `weight` s'écrit très naturellement de façon récursive :

```
let rec weight t =
  match t with
  | Leaf ->
      1
  | Node (t1, t2) ->
      weight t1 + weight t2
```

Ce code effectue un nombre constant d'opérations à chaque feuille et à chaque nœud de l'arbre. Sa complexité asymptotique en temps est donc $O(n)$, où n est la taille de l'arbre, c'est-à-dire son nombre de feuilles et de nœuds. On peut remarquer que, si un arbre a n feuilles, alors il a $n - 1$ nœuds. On peut donc dire également que la complexité asymptotique est $O(p)$, où p est le poids de l'arbre.

Certains élèves ont écrit que la complexité est $O(2^h)$, où h est la hauteur de l'arbre. C'est vrai, mais c'est une analyse trop pessimiste : si l'arbre est très déséquilibré, on peut avoir $h = n$, et dans ce cas, la borne $O(n)$ est beaucoup plus précise que la borne $O(2^h)$. Lorsque l'on analyse la complexité d'un algorithme, il faut choisir en fonction de quels paramètres on souhaite exprimer cette complexité ; certains choix permettent d'exprimer une borne plus précise. □

Solution de la question 5 La fonction `enumerate` s'écrit elle aussi sous forme récursive :

```
let rec enumerate n f =
  if n = 1 then
    f Leaf
  else
    for n1 = 1 to n - 1 do
      let n2 = n - n1 in
      enumerate n1 (fun t1 ->
        enumerate n2 (fun t2 ->
          f (Node (t1, t2))
        )
      )
    done
```

Dans le cas de base, où `n` vaut 1, il existe un seul arbre de poids `n`, à savoir `Leaf`. On soumet donc cet arbre à la fonction `f`.

Dans le cas général, où `n` est strictement supérieur à 1, tous les arbres de poids `n` sont de la forme `Node (t1, t2)`, où `t1` et `t2` sont deux sous-arbres, dont les poids respectifs `n1` et `n2` ont pour somme `n`. On énumère donc (à l'aide d'une boucle `for`) toutes les manières possibles de décomposer `n` en une somme `n1 + n2`; puis, pour chacune d'elles, on énumère tous les arbres `t1` de poids `n1`, et pour chacun d'eux, tous les arbres `t2` de poids `n2`. Au cœur de ces trois boucles imbriquées, on soumet l'arbre `Node(t1, t2)` à la fonction `f`.

Ce code est peu efficace en temps, en particulier parce que `enumerate i` est appelée de nombreuses fois pour une même valeur de `i`. On peut imaginer mémoiser (stocker) pour chaque entier `i` la liste des tous les arbres de poids `i`. On devrait alors gagner du temps, au prix d'une plus grande consommation d'espace.

Cette question a été peu traitée et souvent mal traitée. Beaucoup d'élèves ont considéré (à tort !) que la question était erronée et que la fonction `enumerate` devait attendre un argument supplémentaire, à savoir un arbre.

L'auteur du sujet aurait pu clarifier la question en soulignant qu'il fallait produire **tous** les arbres de poids n **à partir de rien**. Il aurait pu également demander d'abord **combien** il existe d'arbres de poids n. La réponse est donnée par $A(1) = 1$ et $A(n) = \Sigma_{i=1}^{n-1} A(i)A(n-i)$. Ce sont les nombres de Catalan.

Certains élèves ont eu l'idée d'énumérer d'abord les arbres de poids $n-1$, puis pour chacun d'eux, d'énumérer toutes les façons de remplacer une feuille par l'arbre à deux feuilles `Node (Leaf, Leaf)`. Malheureusement, cette approche conduit à produire plusieurs fois un même arbre de poids n. □

Solution de la question 6 D'après la question 4, l'appel `weight t1` a un coût linéaire vis-à-vis de la taille du sous-arbre `t1`. La concaténation `weights t1 @ ...` a un coût linéaire vis-à-vis de la taille de la liste `weights t1`, c'est-à-dire (à nouveau) linéaire vis-à-vis de la taille du sous-arbre `t1`. D'après ces deux remarques, la fonction `weights` effectue à chaque nœud de l'arbre des opérations dont le coût est linéaire. Il en résulte que le coût asymptotique d'un appel à `weights t` est, dans le cas le pire, $O(n^2)$, où n est la taille de l'arbre `t`.

Certains élèves ont noté que si l'arbre est équilibré, alors le coût est $O(n \log n)$. C'est vrai. Toutefois, on ne faisait pas cette hypothèse. □

Solution de la question 7 Il s'agit d'une simple variation sur la définition du type `tree` :

```
type wtree =
  | WLeaf
  | WNode of int * wtree * wtree
```

On stocke dans chaque nœud, en plus des deux sous-arbres, un entier, censé être le poids du sous-arbre gauche. □

Solution de la question 8 Le fait de calculer à la fois l'arbre annoté et son poids nous permet d'effectuer une seule passe sur l'arbre, et d'obtenir ainsi une complexité linéaire. Une formulation plus naïve, où l'on tenterait d'écrire une fonction récursive de type `tree -> wtree`, exigerait d'appeler la fonction `weight` à chaque nœud, et conduirait à une complexité en temps quadratique.

```
let rec annotate t =
  match t with
  | Leaf ->
      WLeaf, 1
  | Node (t1, t2) ->
      let wt1, w1 = annotate t1
```

```
    and wt2, w2 = annotate t2 in
    WNode (w1, wt1, wt2), w1 + w2
```

Dans le cas d'une feuille, on renvoie une paire d'une feuille `WLeaf` et du poids de l'arbre, à savoir 1.

Dans le cas d'un nœud binaire, on effectue deux appels récursifs pour obtenir les sous-arbres annotés `wt1` et `wt2` ainsi que les poids `w1` et `w2` des sous-arbres `t1` et `t2`. Il reste alors à construire un arbre annoté par le poids de son sous-arbre gauche, c'est-à-dire `wt1`, et à renvoyer le poids de l'arbre `t`, à savoir `w1 + w2`. □

Solution de la question 9 Encore une fois, la fonction demandée s'écrit directement sous forme récursive :

```
let rec wtree_iter wt f =
  match wt with
  | WLeaf ->
      ()
  | WNode (w1, wt1, wt2) ->
      wtree_iter wt1 f;
      f w1;
      wtree_iter wt2 f
```

Dans le cas `WLeaf`, il n'y a pas d'annotation, donc rien à faire. Dans le cas `WNode`, on traverse le sous-arbre gauche `wt1`, puis on soumet à la fonction `f` l'annotation `w1`, puis on traverse le sous-arbre droit `wt2`. □

Solution de la question 10 Il faut appeler d'abord `annotate` pour obtenir un arbre annoté `wt`, puis `wtree_iter` pour parcourir cet arbre :

```
let weights t =
  let wt, _ = annotate t in
  let ws = ref [] in
  wtree_iter wt (fun w -> ws := w :: !ws);
  List.rev !ws
```

Afin d'accumuler les éléments de la séquence de poids, on se donne une référence `ws`, dont le contenu initial est la liste vide. À chaque fois que `wtree_iter` produit un élément, on l'ajoute à la liste stockée dans `ws`. Le contenu final de `ws` est la liste souhaitée, mais dans l'ordre inverse de celui souhaité, car on a ajouté les éléments, l'un après l'autre, en tête de liste. On la renverse donc à l'aide de la fonction `List.rev`.

La complexité en temps de ce code est linéaire. En effet, `annotate`, `wtree_iter` et `List.rev` ont une complexité linéaire ; et la fonction `fun w -> ...`, qui est appelée pour chaque élément, a un coût constant. □

Solution de la question 11 Encore une fois, le code s'écrit sous forme récursive, et a une complexité linéaire :

```
let rec weights_iter f t =
  match t with
  | Leaf ->
      1
  | Node (t1, t2) ->
      let w1 = weights_iter f t1 in
      f w1;
      let w2 = weights_iter f t2 in
      w1 + w2
```

Dans le cas d'une feuille, on n'appelle pas `f`, puisque la séquence de poids est vide, et on renvoie le poids 1.

Dans le cas d'un nœud binaire, un premier appel récursif permet d'appliquer `f` à la première partie de la séquence des poids, et en même temps de calculer le poids du sous-arbre `t1`. Alors, on peut appliquer `f` à `w1`, qui est l'élément suivant de la séquence de poids. Enfin, le second appel récursif applique `f` à la dernière partie de la séquence de poids, et permet d'obtenir le poids `w2` du sous-arbre `t2`. Il ne reste qu'à calculer et renvoyer le poids total. □