École Polytechnique

Promotion 2010, Année 2011–2012

Algorithmique et Programmation (INF431)

Contrôle Classant 2

4 juillet 2012

François Pottier, Philippe Jacquet et Benjamin Werner

Corrigé

Première partie

Travaillez efficacement

Cette partie doit être traitée sur des feuilles de couleur **ROSE**.

La fin de l'année approche. Vous devez réviser m matières différentes ; ces matières sont désignées par les numéros de 1 à m.

Dans chaque matière, vous obtiendrez une note entière comprise entre 0 et g avec $g > 1$. Vous avez pu déterminer, pour chaque matière, une fonction qui vous donne votre note à partir du nombre d'heures de révision que vous consacrez à cette matière. Si vous travaillez la matière i pendant h heures, vous obtiendrez la note $p_i(h)$. Pour simplifier, on suppose que l'on ne peut pas diviser les heures, c'est-à-dire que h est entier. On peut supposer que les fonctions p_i sont croissantes.

Il ne vous reste plus que n heures avant les examens, et vous devez les répartir au mieux entre les matières. C'est-à-dire que vous devez trouvez pour chaque matière i un nombre h_i d'heures tel que :

(1) $\Sigma_{i=1}^{m} h_i = n$

et (2) $\Sigma_{i=1}^{m} p_i(h_i)$ est maximal.

Question 1 Donnez un algorithme qui calcule le nombre maximal de points que vous pouvez obtenir. Vous pouvez écrire cet algorithme en pseudo-code ou en Java.

Si vous l'écrivez en Java, cela sera une fonction `int max (int n, int m, int[][] p)` qui prend donc en argument n et m ; le tableau `p` décrivant les fonctions p_i par `p`$[i][j] = p_i(j)$. Vous pouvez du coup préférer numéroter vos matières de 0 à $m - 1$.

Donnez la complexité en temps de votre algorithme. Cette complexité doit être polynomiale, c'est à dire bornée par $n^\alpha \cdot m^\beta$ où α et β sont des constantes bien choisies. ◇

Solution. C'est quasiment une question de cours, avec une utilisation classique de programmation dynamique. On note $M(i, j)$ le nombre maximal de points qu'on peut obtenir avec juste les matières $1 \ldots i$ et j heures de travail. On a alors :

$$M(1, j) = p_1(j)$$
$$M(i + 1, j) = max_{l \leq j}(p_{i+1}(l) + M(i, j - l))$$

On peut alors remplir itérativement un tableau à deux dimensions contenant les valeurs $M(i, j)$. En java, cela donne :

```
public static int max(int n, int m, int[][] p){
      int[][]max = new int[m+1][n+1]; // max[i][j] max pour mat <= i et h = j
       int i; int j; int k; int r;
       for(i=0; i<m; i++){
           for(j=0; j<=n; j++){max[i][j] = 0;}}
       for(j=0; j<=n; j++){  // heures
           max[0][j] = p[0][j];
           for(i=1; i<m; i++){ //matieres
               for(k=0; k<=j; k++){  // heures
                   r = p[i][k]+max[i-1][j-k];
                   if (r>max[i][j]){
                       max[i][j] = r;
                   }
               }
           }}
       return(max[m-1][n]);
  }
```

La complexité est de $O(n^2 \cdot m)$. □

Question 2 A partir de là, décrivez un algorithme qui calcule effectivement les h_i correspondant à la répartition optimale entre les matières. ◇

Solution. On garde le même algorithme, en gardant la trace des valeurs h_i. Par exemple :

```
public static int max(int n, int m, int[][] p){
      int[][]max = new int[m+1][n+1]; // max[i][j] max pour mat <= i et h = j
      int[][]h = new int[m+1][n+1];  // h[i][j] nbr d'heures consacrees a i
                                     // pour obtenir max[i][j] qd on dispose de j heures
       int i; int j; int k; int r;
       for(i=0; i<m; i++){
           for(j=0; j<=n; j++){max[i][j] = 0;}}
       for(j=0; j<=n; j++){  // heures
           max[0][j] = p[0][j];
           h[0][j] = j;
           for(i=1; i<m; i++){ //matieres
               for(k=0; k<=j; k++){  // heures
                   r = p[i][k]+max[i-1][j-k];
                   if (r>max[i][j]){
                       max[i][j] = r;
                       h[i][j] = k;
                   }
               }
           }}
       int hc = n;
       for(i=m-1; i>=0; i--){
           System.out.print(h[i][hc]);
           hc = hc-h[i][hc];
           System.out.print(" ");
       }
       System.out.println();

       return(max[m-1][n]);
  }
```

Ici on affiche le résultat à la volée, pour montrer comment retrouver les valeurs précises. On n'a pas été trop exigeant sur ce dernier point à la correction. □

Deuxième partie

Preuve de la terminaison d'un algorithme

Cette partie doit être traitée sur des feuilles de couleur **ROSE**.

Soit n un entier supérieur ou égal à deux.

On note $[1, n]$ l'ensemble des entiers compris entre 1 et n.

Dans la suite, on manipule des matrices de dimension $n \times n$ à coefficients réels.

On définit sur ces matrices une opération binaire "$\star$" de la manière suivante. Si A et B sont deux matrices, alors $A \star B$ est la matrice C telle que, pour tout $(i, j) \in [1, n]^2$:

$$C_{ij} = \min_{k \in [1,n]} (A_{ik} + B_{kj}) \, .$$

On note $\mathcal{K}_n$ l'ensemble des matrices de dimension $n \times n$ dont les coefficients sont positifs ou nuls et dont la diagonale est nulle. En d'autres termes, on a $X \in \mathcal{K}_n$ si et seulement si :

$$\begin{array}{cccc} & (1) & \forall (i,j) \in [1,n]^2 & X_{ij} \geq 0 \\ \text{et} & (2) & \forall i \in [1,n] & X_{ii} = 0. \end{array}$$

Dans toute la suite, on fixe une matrice $P \in \mathcal{K}_n$. On considère la suite de matrices $M(t)$, où t est un entier positif ou nul, définie par l'algorithme suivant :

procédure BFL()
  $t \leftarrow 0$
  $M(t) \leftarrow P$
  $t \leftarrow t + 1$
  $M(t) \leftarrow M(t-1) \star M(t-1)$
  tant que $M(t) \neq M(t-1)$ **faire**
    $t \leftarrow t + 1$
    $M(t) \leftarrow M(t-1) \star M(t-1)$

Notre objectif est de prouver que l'algorithme termine. Au préalable, nous allons démontrer quelques propriétés de l'algorithme.

Question 3 Montrer que pour tout entier t strictement positif et pour tout $(i, j) \in [1, n]^2$, on a :

$$\begin{array}{cccc} & (1) & \forall k \in [1,n] & M_{ij}(t) \leq M_{ik}(t-1) + M_{kj}(t-1) \\ \text{et} & (2) & \exists k \in [1,n] & M_{ij}(t) = M_{ik}(t-1) + M_{kj}(t-1). \end{array}$$

◇

Solution. La proposition $\forall k \in [1, n] : M_{ij}(t) \leq M_{ik}(t-1) + M_{kj}(t-1))$ est une conséquence directe de $M_{ij}(t) = \min_{k \in [1,n]} \{M_{ik}(t-1) + M_{kj}(t-1)\}$. La proposition $\exists k \in [1, n] : M_{ij}(t) = M_{ik}(t-1) + M_{kj}(t-1)$ provient du fait que le minimum d'un ensemble fini est nécessairement atteint. □

Question 4 Montrer que pour tout entier t positif ou nul, on a $M(t) \in \mathcal{K}_n$.

Solution. La proposition est vraie pour $t = 0$.

Comme $M_{ii}(t) \leq 2M_{ii}(t-1)$, on a, par récurrence, $M_{ii}(t) = 0$. De même, la proposition (2) de la question précédente permet de vérifier que si $\forall (i, j) \in [1, n]^2 M_{ij}(t-1) \geq 0$, alors $\forall (i, j) \in [1, n]^2 M_{ij}(t) \geq 0$. Le résultat suit, par récurrence. □

Question 5 Montrer que pour tout entier t strictement positif et pour tout $(i, j) \in [1, n]^2$, on a $M_{ij}(t) \leq M_{ij}(t-1)$. ◇

Solution. La proposition précédente implique que $M_{ij}(t) \leq M_{ii}(t-1) + M_{ij}(t-1)$, comme $M_{ii}(t-1) = 0$, selon la proposition démontrée à la question 4, la proposition est démontrée. □

Soit t un entier strictement positif. On définit l'ensemble $stable(t)$ comme l'ensemble des couples $(i, j) \in [1, n]^2$ tels que :

$$\begin{array}{lll} & (1) & M_{ij}(t) = M_{ij}(t-1) \\ \text{et} & (2) & \forall (k,\ell) \in [1,n]^2 \quad M_{k\ell}(t) < M_{ij}(t) \Rightarrow M_{k\ell}(t) = M_{k\ell}(t-1). \end{array}$$

Question 6 Montrer que $stable(1)$ n'est pas vide. Montrer de plus que si $(i, j) \in stable(t)$ alors pour tout $(k, \ell) \in [1, n]^2$ on a :

$$M_{k\ell}(t) < M_{ij}(t) \Rightarrow (k, \ell) \in stable(t)\ .$$ ◇

Solution. $\forall i \in [1, n] : (i, i) \in stable(1)$. En effet, on a bien $M_{ii}(1) = 0 \leq M_{ii}(0) = 0$. De plus il n'y a pas de j, k tel que $M_{jk}(1) < M_{ii}(1) = 0$.

Pour la seconde proposition, si $M_{k\ell}(t) < M_{ij}(t)$, alors :
- la définition garantit directement que $M_{k\ell}(t-1) = M_{k\ell}(t)$,
- si $M_{k'\ell'} < M_{k\ell}$ on a aussi $M_{k'\ell'} < M_{ij}$, et donc $M_{k'\ell'}(t) = M_{k'\ell'}(t-1)$. □

Question 7 Montrer que pour tout entier t supérieur ou égal à 2, on a $stable(t-1) \subseteq stable(t)$. ◇

Solution. Soit $(i, j) \in stable(t-1)$. On a donc $M_{ij}(t-1) = M_{ij}(t-2)$ nous allons prouver que $M_{ij}(t) = M_{ij}(t-1)$. Nous savons que $\forall (k, \ell) \in [1, n]^2$:

$$M_{k\ell}(t-1) < M_{ij}(t-1) \implies (k, \ell) \in stable(t-1)\ .$$

On va raisonner par l'absurde en supposant que $M_{ij}(t) < M_{ij}(t-1)$.

1. On sait qu'il existe $k \in [1, n]$ tel que $M_{ij}(t) = M_{ik}(t-1) + M_{kj}(t-1)$. Comme $M_{ik}(t-1)$ et $M_{kj}(t-1)$ sont tous les deux inférieurs ou égaux à $M_{ij}(t)$ ils sont donc strictement inférieurs à $M_{ij}(t-1)$. Donc (i, k) et (k, j) appartiennent à $stable(t-1)$.
2. Donc $M_{ik}(t-1) = M_{ik}(t-2)$ et $M_{kj}(t-1) = M_{kj}(t-2)$. Comme $M_{ij}(t) = M_{ik}(t-2) + M_{kj}(t-2) \geq M_{ij}(t-1)$, ceci contredit l'hypothèse $M_{ij}(t) < M_{ij}(t-1)$.

Pour prouver que $(i, j) \in stable(t)$ il reste à prouver que $\forall (k, \ell) M_{k\ell}(t) < M_{ij}(t) \implies M_{k\ell}(t) = M_{k\ell}(t-1)$ ce qui revient à appliquer le résultat précédent puisque $(k, \ell) \in stable(t-1)$. □

Pour tout sous-ensemble U de $[1, n]^2$, on note $\complement U$ le sous-ensemble complémentaire de U vis-à-vis de $[1, n]^2$.

Question 8 Montrer que pour tout entier t supérieur ou égal à 2, $\complement stable(t-1) \neq \emptyset$ implique $stable(t) - stable(t-1) \neq \emptyset$. En déduire que le cardinal de l'ensemble $\complement stable(t-1)$ est un variant de la boucle. ◇

Solution. Soit (i, j) l'élément de $\complement stable(t-1)$ qui minimise $M_{ij}(t-1)$. Nous allons montrer que $(i, j) \in stable(t)$. Nous savons déjà que tous les couples (k, ℓ) tels que $M_{k\ell}(t-1) < M_{ij}(t-1)$ sont donc éléments de $stable(t-1)$ et de $stable(t)$, au vu de la question précédente.

Il faut d'abord montrer que $M_{ij}(t) = M_{ij}(t-1)$. Supposons par l'absurde que $M_{ij}(t) < M_{ij}(t-1)$. Il existe k tel que $M_{ik}(t-1) + M_{kj}(t-1) = M_{ij}(t) < M_{ij}(t-1)$ donc (i, k) et (k, j) appartiennent à $stable(t-1)$. Dans ce cas $M_{ik}(t-1) = M_{ik}(t-2)$ et $M_{kj}(t-1) = M_{kj}(t-2)$ donc $M_{ij}(t) = M_{ik}(t-2) + M_{kj}(t-2) \geq M_{ij}(t-1)$ ce qui est contradictoire.

Ensuite il faut montrer que pour $(k,\ell)[1,n]^2$ tel que $M_{k\ell}(t) < M_{ij}(t)$, alors $M_{k\ell}(t) = M_{k\ell}(t-1)$. Il suffit d'utiliser $u \in [1,n]$ tel que $M_{k\ell}(t) = M_{ku}(t-1)+M_{u\ell}(t-1)$. Comme $M_{k\ell}(t) < M_{ij}(t) = M_{ij}(t-1)$ alors $M_{ku}(t-1)$ et $M_{u\ell}(t-1)$ sont strictement inférieurs à $M_{ij}(t-1)$. Cela fait que (k,u) et (u,ℓ) sont tout deux éléments de $stable(t-1)$. Donc $M_{ku}(t-1) = M_{ku}(t-2)$ et $M_{u\ell}(t-1) = M_{u\ell}(t-2)$. De l'identité $M_{k\ell}(t) = M_{ku}(t-2)+M_{u\ell}(t-2)$ on tire $M_{k\ell}(t) \geq M_{k\ell}(t-1)$, c'est-à-dire $M_{k\ell}(t) = M_{k\ell}(t-1)$.□

Les questions précédentes permettent de conclure que l'algorithme BFL termine et que le nombre d'itérations de la boucle est borné par n^2. Dans ce qui suit, on souhaite obtenir une borne supérieure plus précise en remarquant que cet algorithme calcule certains chemins optimaux dans un graphe.

On considère un graphe orienté constitué de n sommets numérotés de 1 à n. Le graphe est complet : entre deux sommets i et j arbitraires, il existe un arc. La quantité P_{ij} représente le poids de l'arc (i,j). On souligne le fait que les poids P_{ij} et P_{ji} ne sont pas nécessairement égaux.

Un chemin C menant du sommet i au sommet j est une suite de sommets $(r_0, r_1, \ldots, r_\ell)$ avec $r_0 = i$ et $r_\ell = j$. L'entier ℓ est la longueur du chemin C. La quantité $\sum_{k=0}^{\ell-1} P_{r_k r_{k+1}}$ est le poids du chemin C.

Question 9 Montrer que pour tout entier t positif ou nul et pour tout $(i,j) \in [1,n]^2$, la quantité $M_{ij}(t)$ est égale au poids minimal d'un chemin menant du sommet i au sommet j et de longueur inférieure ou égale à 2^t. ◇

Solution. On commence par remarquer que s'il existe un chemin de i à j qui est de longueur n et de poids p, alors il existe un chemin de même poids p et de longueur $n+m$, quel que soit m positif. Il suffit pour cela d'ajouter m passages par le sommet i au début du chemin.

Il suffit donc de montrer que $M_{ij}(t)$ est le poids minimal d'un chemin de i à j de longueur égale à 2^t.

Si $t > 1$, alors un chemin de i à j de longueur $2t$ a un milieu ; c'est-à-dire qu'il se décompose un chemin de i à k et un chemin de k à j, tous deux de longueur t.

A partir de là, on voit que trouver un chemin de poids minimal de longueur $2t$ revient à trouver le sommet k qui minimise la somme des poids des chemins de longueur t de i à k et de k à j.

Le résultat suit par récurrence.

Voici une autre présentation possible :

La proposition est trivialement vraie pour $t = 0$. On va montrer la proposition par récurrence. On constate que n'importe quel chemin de longueur $\ell \leq 2^t$ entre i et j peut être étendu à un chemin de même poids en ajoutant des arcs (j,j) de poids nul. Supposons qu'elle soit vraie pour t donné, c'est-à-dire :

$$M_{ij}(t) = \min_{\substack{(r_0,\cdots,r_{2^t})\in[1,n]^{2^t+1} \\ r_0=i \wedge r_{2^t}=j}} \{\sum_{k=0}^{2^t-1} P_{r_k r_{k+1}}\}$$

En partant de

$$M_{ij}(t+1) = \min_{k\in[1,n]} \{M_{ik}(t) + M_{kj}(t)\}$$

on réécrit

$$M_{ij}(t+1) = \min_{k\in[1,n]} \left\{ \min_{\substack{(r_0,\cdots,r_{2^t})\in[1,n]^{2^t+1} \\ r_0=i \wedge r_{2^t}=k}} \{\sum_{k=0}^{2^t-1} P_{r_k r_{k+1}}\} + \min_{\substack{(r_{2^t},\cdots,r_{2^{t+1}})\in[1,n]^{2^t+1} \\ r_{2^t}=k \wedge r_{2^{t+1}}=j}} \{\sum_{k=2^t}^{2^{t+1}-1} P_{r_k r_{k+1}}\} \right\}$$

qui se résume en

$$M_{ij}(t+1) = \min_{\substack{(r_0,\cdots,r_{2^{t+1}})\in[1,n]^{2^{t+1}+1} \\ r_0=i\wedge r_{2^{t+1}}=j}} \{\sum_{k=0}^{2^{t+1}-1} P_{r_k r_{k+1}}\}$$

ce qui prouve la propriété pour $t+1$. □

Question 10 En combien d'itérations au maximum l'algorithme BFL termine-t-il ? Démontrez-le. ◇

Solution. On appelle chemin optimal entre deux sommets i et j un chemin dont le poids correspond au poids minimal toutes longueurs confondues. Entre deux sommets i et j il existe un chemin optimal qui ne contient pas de boucle, donc qui est de longueur inférieure ou égale à n. Donc les coefficients de $M(t)$ correspondent aux poids optimaux si $2^t \geq n$, donc ne peuvent plus décroître. En conséquence l'algorithme BFL termine en moins de $\lceil \log_2 n \rceil$ itérations. □

Troisième partie

Une araignée concurrente

Cette partie doit être traitée sur des feuilles de couleur **JAUNE**.

Dans cette partie, on s'intéresse à l'implémentation d'une « araignée Web », c'est-à-dire d'un programme qui découvre et parcourt un ensemble de pages Web reliées entre elles par des « hyperliens ».

1 Hypothèses

On suppose qu'une page Web est identifiée de façon unique par une adresse, ou « URL ». On suppose de plus que, lorsqu'on connaît une URL, on peut (grâce au réseau) obtenir le texte de la page Web correspondante, puis analyser ce texte pour en extraire les hyperliens, c'est-à-dire les URLs, qu'il contient.

On peut donc considérer que le Web forme un graphe orienté dont les sommets sont les URLs et dont les arêtes sont données par les hyperliens. On suppose que ce graphe est fini.

Les hypothèses ci-dessus se traduisent en Java de la façon suivante. On suppose donnée une classe `URL` (figure 1). Un objet de classe `URL` est immuable, et représente une certaine URL, fixée une fois pour toutes au moment où cet objet est construit. La classe `URL` implémente les méthodes `equals` et `hashCode`, de sorte que les objets de classe `URL` peuvent être comparés et peuvent servir de clefs dans une table de hash. Enfin, la classe `URL` fournit une méthode `successors`, qui n'attend aucun argument et qui renvoie la liste des successeurs de l'objet `this` dans le graphe.

Cette liste est représentée par un objet de type `Iterable<URL>`. Cela signifie que l'on peut énumérer les éléments de cette liste à l'aide d'une boucle `for`. Plus précisément, si `urls` est une liste de type `Iterable<URL>`, alors la boucle `for (URL url : urls) { ... }` permet d'itérer sur la liste `urls`.

On souligne le fait qu'un appel à `successors` peut exiger un temps très long : en effet, cette méthode envoie une requête à travers le réseau et ne rend la main à son appelant que lorsque le serveur distant a répondu à cette requête. Cela peut prendre plusieurs secondes ! Pendant ce temps, le *thread* qui a appelé `successors` est bloqué.

On souhaite donc réaliser un parcours de graphe à l'aide de plusieurs *threads*, de façon à ce que le parcours puisse progresser à une cadence raisonnable même si certains appels à `successors` demandent un temps très long.

Tout au long de cette partie, on demande d'écrire **non pas du pseudo-code**, mais **du code Java correct**.

Dans tout ce sujet comme dans le cours, on prétend, pour simplifier les choses, que **l'exception** `InterruptedException` **n'existe pas**. De plus, on demande d'utiliser des verrous explicites, c'est-à-dire des objets de classe `ReentrantLock`, classe qui implémente l'interface `Lock` (figure 3). On **n'utilisera donc pas le mot-clef** `synchronized` de Java. Enfin, **on n'utilisera pas le mot-clef** `volatile` de Java.

2 Une table de hash partagée

Pour mémoriser l'ensemble des sommets découverts pendant le parcours, on souhaite utiliser une table de hash, c'est-à-dire un objet de classe `HashSet`, dont la définition est rappelée dans la figure 2. On notera que cette classe est paramétrée par le type `E` des éléments de l'ensemble. On notera également que la méthode `add` renvoie un booléen qui indique si l'élément `e` a effectivement été ajouté, ou en d'autres termes, si l'élément `e` était absent de l'ensemble.

Question 11 Expliquez très brièvement pourquoi l'objet qui représente l'ensemble des sommets découverts pendant le parcours devra être *partagé*, c'est-à-dire utilisé par plusieurs *threads*. ◇

Solution. Plusieurs *threads* seront chargés de visiter indépendamment différents sommets. Or, lorsqu'on visite un sommet, il faut (comme dans le cas séquentiel) vérifier s'il a déjà été marqué comme « visité », et si ce n'est pas le cas, le marquer. Plusieurs *threads* ont donc besoin d'accéder à la table de hash qui représente l'ensemble des sommets marqués. □

La documentation de l'API Java indique que la classe `HashSet` n'est pas « synchronisée ». En d'autres termes, si plusieurs *threads* tentent simultanément d'accéder à un même objet de classe `HashSet`, alors une *race condition* se produit, et le programme peut se comporter de façon incohérente.

Pour garantir qu'un seul *thread* à la fois peut accéder à cet objet, on souhaite donc le protéger à l'aide d'un verrou.

Question 12 En utilisant la classe `HashSet`, implémentez une classe `SafeHashSet`, qui se présente exactement comme la classe `HashSet` (c'est-à-dire qui propose un constructeur sans argument et les méthodes `add`, `contains`, et `size`), de façon à ce qu'un objet de type `SafeHashSet<E>` puisse sans danger être utilisé simultanément par plusieurs *threads*. ◇

Solution. La classe `SafeHashSet` (figure 8) contient deux champs : la table de hash sous-jacente, `set`, et un verrou, `l`. Le constructeur initialise ces deux champs. (On pourrait également donner

```
public class URL {
  // the fields and constructor are omitted
  public boolean equals (Object o);
  public int hashCode ();
  public Iterable<URL> successors ();
}
```

FIGURE 1 – La classe `URL`

une valeur initiale à ces deux champs via des *initialiseurs*, et ne pas écrire de constructeur ; Java fournirait alors un *constructeur par défaut*.)

Ces champs doivent être déclarés `private` pour empêcher quiconque d'y accéder autrement que via les méthodes `add`, `contains` et `size`.

Une fois ces champs initialisés, leur valeur n'est plus jamais modifiée : pour souligner cela, ces champs peuvent être déclarés `final`.

Notons que, une fois l'initialisation terminée (et seulement alors), plusieurs *threads* pourront accéder simultanément au champ `l`. Cela ne constitue pas une *race condition*, car deux accès simultanés à un même emplacement mémoire sont permis si ce sont deux accès *en lecture*.

Chacune des trois méthodes `add`, `contains` et `size` est implémentée simplement en appelant la méthode correspondante de la table `set` et en plaçant cet appel dans une *section critique* protégée par le verrou `l`. Ces sections critiques sont délimitées, comme on l'a vu en cours, par des appels aux méthodes `lock` et `unlock`. La construction `try/finally` doit être utilisée pour garantir que le verrou est relâché quoiqu'il arrive (par exemple, même si une exception est lancée).

Soulignons le fait que les méthodes `contains` et `size` doivent constituer des sections critiques, même si elles ne font que *consulter* la table de hash. En effet, une écriture et une lecture simultanée constituent bien une *race condition*. Un appel à `contains` ou `size` pourrait produire un résultat totalement imprévisible si un appel à `add` avait lieu simultanément.

Signalons que, au lieu de construire un objet distinct, `set`, de classe `HashSet`, on pouvait déclarer que `SafeHashSet` *hérite* de `HashSet`, c'est-à-dire écrire `SafeHashSet<E> extends HashSet<E>`. Dans ce cas, au lieu d'écrire `set.add(e)`, on écrivait `super.add(e)`. Il n'est pas immédiat de se convaincre que cette solution est correcte. En particulier, si (par exemple) la méthode `add` de la classe `HashSet` appelle la méthode `contains` (c'est possible ; nous ne savons pas comment la méthode `HashSet.add` est écrite), alors un appel à `SafeHashSet.add` provoquera un appel à `HashSet.add`, qui lui-même provoquera un appel à `SafeHashSet.contains`. De ce fait, le verrou sera pris deux fois successivement. Cela ne provoque pas de « *deadlock* » parce que les verrous de Java sont « ré-entrants » : un même *thread* peut prendre un même verrou plusieurs fois. Cette solution était donc acceptée. Elle est toutefois un peu moins sûre, car `SafeHashSet` hérite alors de *toutes* les méthodes et de *tous* les champs de la classe `HashSet`, et si certains de ces méthodes ou champs ne sont pas déclarés `private`, alors il sera possible d'y accéder sans prendre le verrou, ce qui est dangereux. □

3 Un premier algorithme de parcours concurrent

Pour réaliser un parcours de graphe concurrent, une idée simple consiste à lancer un nouveau *thread* pour chaque sommet **nouvellement découvert**.

Ces *threads* **partageront** un objet `discovered` de type `SafeHashSet<URL>` qui représentera l'ensemble des URLs déjà découvertes.

Pour décrire la tâche à effectuer lorsqu'une URL est nouvellement découverte, on se propose d'écrire une classe `URLTask`.

Question 13 Implémentez une classe `URLTask` en respectant les consignes suivantes :
- la classe `URLTask` doit être dotée d'un constructeur dont les deux arguments sont l'objet `discovered`, de type `SafeHashSet<URL>`, et un objet `url`, de type `URL` ;
- la classe `URLTask` doit implémenter l'interface `Runnable`, dont la définition est rappelée dans la figure 4 ;
- la méthode `run` suppose que le sommet `url` a déjà été ajouté à l'ensemble `discovered` mais n'a pas encore été traité ;

– vous n'utiliserez aucune variable globale, c'est-à-dire aucun champ `static`.

Vous utiliserez la classe `Thread`, dont la définition est rappelée dans la figure 5. ◇

Solution. La solution est donnée dans la figure 9. On utilise deux champs immuables, initialisés par le constructeur, pour que la méthode `run`, qui n'attend aucun argument, puisse avoir accès à `discovered` et à `url`.

On suppose que le sommet `url` est *nouvellement découvert*, c'est-à-dire qu'il a déjà été marqué comme « visité », mais que cette visite n'a pas encore lieu : en particulier, `url.successors()` n'a pas encore été appelé.

La première chose à faire est donc d'appeler `url.successors()` pour obtenir une liste des successeurs du sommet `url`. On itère ensuite sur cette liste. Pour chaque successeur `successor` :

1. si `discovered.add(successor)` renvoie `false`, alors ce successeur a déjà été marqué précédemment, donc il n'y a plus rien à faire à son sujet ;
2. si `discovered.add(successor)` renvoie `true`, c'est que ce successeur n'était pas encore connu ; l'appel à `add` vient de le marquer ; il reste donc à « visiter » ce successeur nouvellement découvert. Nous lançons donc un *nouveau thread* chargé de cette tâche.

Notons qu'il serait *incorrect* d'écrire :

```
if (!discovered.contains(successor)) {
  discovered.add(successor); ...
}
```

En effet, cette séquence de deux appels de méthodes n'est pas exécutée de façon atomique, donc il est théoriquement possible dans ce cas que deux *threads* réussissent tous deux à entrer dans le corps du « `if` ». Ceci conduirait le sommet `successor` à être étudié deux fois.

Une fois l'énumération des successeurs terminée, la tâche du *thread* chargé d'étudier le sommet `url` est terminée. Sa méthode `run` termine, et ce *thread* meurt. Notons qu'il est inutile d'utiliser `join` pour attendre la terminaison des *threads* chargés d'examiner les successeurs du sommet `url`. Si on utilisait `join` de cette manière, on augmenterait sans raison la consommation en mémoire du programme. □

Pour compléter la définition de l'algorithme de parcours, il reste à écrire le code qui lance le parcours à partir d'une URL racine.

Question 14 Dans une classe `NaiveTraversal`, implémentez une méthode `traverse` dont l'argument `root`, de type `URL`, représente le point de départ du parcours. La méthode `traverse` ne renvoie aucun résultat et n'attend pas que le parcours soit terminé. Elle n'utilise aucune variable globale. ◇

Solution. La solution est donnée dans la figure 10. On crée la table `discovered`, initialement vide. On y ajoute le sommet `root`, puis on lance un premier *thread* auxiliaire, chargé de « visiter » ce sommet. Puisque l'énoncé ne demandait pas d'attendre la terminaison du parcours, il n'y a rien de plus à faire. □

Question 15 Démontrez brièvement mais rigoureusement que la méthode `successors` est appelée au plus une fois pour chaque URL. ◇

Solution. Un appel à `url.successors()` a lieu uniquement au début de la méthode `run` de la classe `URLTask`. Il suffit donc de démontrer qu'au plus un *thread* de classe `URLTask` est lancé pour chaque URL. Or, on voit que ces *threads* ne sont créés et lancés que lorsqu'une URL est nouvellement découverte : l'expression `new Thread (new URLTask (..., successor)).start()` est soumise à la condition `if (discovered.contains(successor))`. De plus, une fois marquée, une URL reste marquée pour toujours ; donc, une URL est *nouvellement découverte* au plus une fois. □

Question 16 Ce parcours de graphe est-il en profondeur d'abord ? en largeur d'abord ? ni l'un ni l'autre ? Justifiez brièvement. ◇

Solution. Ni l'un ni l'autre. On sait que les différentes stratégies *séquentielles* de parcours de graphe sont définies par la manière dont on gère la *frontière*, c'est-à-dire l'ensemble des sommets déjà marqués mais pas encore effectivement « visités ». Le parcours en profondeur d'abord correspond au cas où la frontière est gérée comme une pile (LIFO) ; le parcours en largeur d'abord correspond au cas où elle est gérée comme une file (FIFO). Dans chacun de ces deux cas, on retire *un seul sommet à la fois* de la frontière, pour le traiter. Or, ici, il n'y a pas vraiment de frontière. On peut dire, si l'on veut, que la frontière est implicitement représentée par l'ensemble des *threads* `URLTask` qui ont été lancés. Cependant, tous ces *threads* s'exécutent *simultanément*, donc on travaille *sur tous les sommets de la frontière à la fois*. La façon dont le parcours progresse est non déterministe, puisqu'on ne sait pas a priori quels *threads* s'exécuteront plus rapidement que les autres. □

Question 17 Démontrez brièvement mais rigoureusement pourquoi ce parcours de graphe termine nécessairement. Vous expliquerez en particulier pourquoi un « *deadlock* » ne peut pas se produire, c'est-à-dire pourquoi un *thread* ne peut pas être bloqué éternellement par un appel à la méthode `lock`. ◇

Solution. Les sections critiques que nous utilisons (et qui apparaissent dans la définition de la classe `SafeHashSet`) ne contiennent aucun appel à une opération bloquante, donc sont exécutées en un temps fini. (Si la classe `HashSet` est correctement implémentée, ce temps doit être $O(1)$.) Par conséquent, aucun *thread* ne conserve le verrou infiniment longtemps. Il en découle (moyennant une hypothèse d'équité, ou *fairness*, qui n'a pas été discutée en cours) qu'un *thread* qui tente de prendre le verrou y parviendra nécessairement. En d'autres termes, un *thread* ne peut pas être bloqué éternellement par un appel à la méthode `lock`.

Supposons qu'un appel à la méthode `url.successors()` termine toujours, c'est-à-dire que les serveurs Web distants répondent toujours à nos requêtes. Parce que nous avons supposé que le graphe du Web est fini, `url.successors()` produit une liste finie.

Il découle des remarques précédentes qu'un *thread* `URLTask` termine toujours.

Enfin, nous avons justifié, lors de la question 15, qu'au plus un *thread* `URLTask` est lancé pour chaque URL. Parce que nous avons supposé que le graphe du Web est fini, le nombre d'URLs est fini, donc le nombre total de *threads* lancés est fini.

Il en découle que le parcours termine nécessairement : au bout d'un certain temps, il ne reste plus aucun *thread*. □

Question 18 Modifiez les classes `URLTask` et `NaiveTraversal` de façon à ce que la méthode `traverse` **attende** la fin du parcours, c'est-à-dire la terminaison de tous les *threads* auxiliaires, et **renvoie** le nombre total d'URLs découvertes pendant le parcours.

Vous pourrez utiliser les méthodes `join`, `sleep` (figure 5) et/ou la classe `AtomicInteger`, dont la définition est rappelée dans la figure 6. Vous pourrez utiliser d'autres mécanismes si vous le souhaitez.

Cette question admet plusieurs solutions, plus ou moins efficaces en termes de temps et d'espace. Toute solution correcte sera acceptée, mais une solution plus efficace pourra recevoir une meilleure note. ◇

Solution. Dans tous les cas, le nombre total d'URLs découvertes pendant le parcours est égal au cardinal de l'ensemble `discovered` à la fin du parcours. Le problème se réduit donc à attendre la fin du parcours ; cela fait, il ne reste qu'à appeler `discovered.size()`.

Pour attendre la fin du parcours, une solution simple, que nous ne présentons pas en détail ici, consiste à utiliser systématiquement l'opération `join` pour attendre la terminaison des *threads*

que l'on lance. (C'est un peu pénible à écrire : dans la méthode `run` de la classe `URLTask`, il faut construire dans la première boucle une liste des objets de classe `Thread` que l'on a créés et lancés, puis utiliser une seconde boucle pour attendre la terminaison de tous ces *threads*. Si l'on commet l'erreur d'écrire une seule boucle où on lance un *thread* `t` et où on appelle aussitôt `t.join()`, alors on détruit tout le parallélisme : le parcours du graphe redevient séquentiel.) Comme on l'a déjà mentionné plus tôt, cette solution n'est pas idéale, car elle prolonge considérablement la durée de vie des *threads* `URLTask`, et provoque ainsi une augmentation de l'espace mémoire nécessaire au parcours.

Une solution plus intéressante consiste à maintenir un compteur du nombre de *threads* `URLTask` existants. Le *thread* principal, qui a exécuté l'appel à `traverse(root)`, attend alors que ce compteur tombe à zéro, ce qui indique que le parcours est terminé.

Ce compteur peut être représenté soit par un objet contenant un champ entier, protégé par un verrou ; soit par un objet de classe `AtomicInteger`. Nous présentons la seconde approche (figure 11).

Le code est modifié en ajoutant un compteur partagé, `activeThreads`, auquel tous les *threads* ont accès. À chaque fois que l'on lance un nouveau *thread* auxiliaire, on incrémente ce compteur, et à chaque fois qu'un *thread* auxiliaire termine son exécution, on décrémente ce compteur. Dans la méthode `run` de la classe `URLTaskBis`, on prend garde à effectuer les incrémentations *avant* la décrémentation, de sorte que le compteur ne passe jamais transitoirement à zéro : s'il atteint la valeur zéro, c'est qu'il n'existe plus aucun *thread* auxiliaire, donc que le parcours est terminé.

Le *thread* principal attend que le compteur prenne la valeur zéro. On utilise une attente active, c'est-à-dire une simple boucle `while`, rendue moins agressive par un appel à `Thread.sleep`. Cet appel retarde la terminaison de la boucle de 100 millisecondes (dans le pire cas), ce qui est peu si on suppose que les appels à `url.successors()` peuvent prendre un temps très important.

En Java, l'appel à `Thread.sleep` peut lancer l'exception `InterruptedException`, ce que nous devons déclarer. Ce détail pouvait être omis.

Pour éviter une attente active de la part du *thread* principal, on peut imaginer d'autres solutions encore.

Par exemple, on peut représenter le compteur partagé non pas par un objet de classe `AtomicInteger`, mais par un objet ordinaire possédant un champ entier protégé par un verrou. En plus des méthodes `increment` et `decrement`, on prévoit une méthode bloquante `awaitZero`. On utilise une variable de condition pour transmettre un signal depuis le *thread* qui a appelé `decrement` et qui trouve que le compteur tombe à zéro vers le *thread* qui a appelé `awaitZero`.

Ou encore, on peut conserver l'idée de représenter le compteur partagé par un objet de classe `AtomicInteger`, et poser que si un *thread* voit le compteur tomber à zéro, alors il doit envoyer un message au *thread* principal. Pour envoyer ce message, on peut utiliser une file d'attente partagée, comme celle étudiée en cours. Le *thread* principal appelle `take`, et bloque, puisque la file est initialement vide. Le *thread* qui constate que le compteur est tombé à zéro appelle `put`, et permet ainsi au *thread* principal de reprendre son exécution. □

4 Un second algorithme de parcours concurrent

Pour réaliser un parcours de graphe concurrent, une idée un peu plus évoluée consiste à placer les sommets nouvellement découverts dans une file d'attente partagée et à utiliser un nombre fixe de *threads*, appelés « travailleurs », pour traiter les éléments de la file d'attente.

Question 19 Selon vous, quels sont en principe les avantages de cette approche vis-à-vis de la précédente ? ◇

Solution. Cette approche permet de limiter le nombre de *threads* qui seront simultanément actifs. Parce que chaque *thread* exige beaucoup d'espace mémoire (pour stocker sa pile, en particulier), c'est important.

De plus, cela permet de limiter le nombre de connexions réseau qui seront simultanément ouvertes. En effet, chaque appel à `url.successors()` établit une connexion réseau, qui reste ouverte tant que la réponse n'a pas été reçue. Il se peut que le système d'exploitation limite le nombre de connexions réseau qu'un processus peut ouvrir simultanément.

Notons que, ici, le fait de lancer plus de *threads* que la machine n'a de cœurs ne conduit pas nécessairement à un problème de performance. En effet, ces *threads* passent la plus grande partie de leur temps à être bloqués par l'appel à `successors()`. □

On rappelle qu'une file d'attente partagée est un objet qui satisfait l'interface `BlockingQueue`, dont la définition est rappelée dans la figure 7. On rappelle que la méthode `take` peut être bloquante (si la file est vide) et que la méthode `put` peut l'être également (si la file a une capacité maximale et si celle-ci a été atteinte).

L'interface `BlockingQueue` est implémentée entre autres par la classe `ArrayBlockingQueue`. Une file d'attente de type `ArrayBlockingQueue` a une capacité maximale, fixée lors de la création : c'est l'unique argument du constructeur.

Comme précédemment, les *threads* « travailleurs » partagent un objet `discovered`, de type `SafeHashSet<URL>`, qui représente l'ensemble des URLs déjà découverts.

De plus, les « travailleurs » partagent un objet `queue`, de type `BlockingQueue<URL>`, qui représente l'ensemble des URLs en attente de traitement. Ces URLs ont déjà été ajoutées à l'ensemble `discovered`, mais n'ont pas encore été traitées.

Pour décrire la tâche à effectuer par chaque « travailleur », on se propose d'écrire une classe `WorkerTask`.

Question 20 Implémentez une classe `WorkerTask` en respectant les consignes suivantes :
- la classe `WorkerTask` doit être dotée d'un constructeur dont les deux arguments sont l'objet `discovered` et l'objet `queue` ;
- la classe `WorkerTask` doit implémenter l'interface `Runnable`, dont la définition est rappelée dans la figure 4 ;
- vous n'utiliserez aucune variable globale, c'est-à-dire aucun champ `static`.

Vous utiliserez la classe `Thread`, dont la définition est rappelée dans la figure 5. Vous ne vous inquièterez pas, pour le moment, de la terminaison des « travailleurs ». ◇

Solution. La solution est donnée dans la figure 12. On utilise deux champs immuables, initialisés par le constructeur, pour que la méthode `run`, qui n'attend aucun argument, puisse avoir accès à `discovered` et à `queue`.

On maintient l'invariant que les URLs qui se trouvent dans la file d'attente `queue` ont déjà été marquées comme « visitées », mais que cette visite n'a pas encore lieu.

Puisqu'on ne s'inquiète pas pour le moment de la terminaison des travailleurs, on utilise une boucle `while (true) { ... }` pour extraire des éléments de la file d'attente, tant qu'il y en a. Lorsqu'il n'y aura plus d'éléments, les travailleurs resteront bloqués par un appel à `take`.

Signalons qu'il ne *faut pas* écrire `while (!queue.isEmpty()) ...`. En effet, dans ce cas, un travailleur qui trouverait la file vide mourrait. Ce n'est pas ce que l'on veut : un travailleur qui trouve la file vide doit attendre que de nouveaux éléments soient ajoutés à la file.

Pour chaque élément `url` extrait de la file, on procède comme précédemment (figure 9), à ceci près que les successeurs nouvellement découverts ne sont pas traités par un nouveau *thread*, mais insérés dans la file d'attente.

En Java, le bloc `try/catch` est rendu nécessaire du fait que `put` et `take` peuvent lancer l'exception `InterruptedException`. Ce détail pouvait être omis. □

Pour compléter la définition de ce second algorithme de parcours, il reste à écrire le code qui lance le parcours à partir d'une URL racine.

Question 21 Dans une classe `Traversal`, implémentez une méthode `traverse` dont l'argument `root`, de type `URL`, représente le point de départ du parcours. La méthode `traverse` ne renvoie aucun résultat et n'attend pas que le parcours soit terminé. Elle n'utilise aucune variable globale. ◇

Solution. La solution est donnée dans la figure 13. La méthode `traverse` crée la table de hash et la file d'attente partagées, insère le sommet `root` dans la file d'attente, puis lance les *threads* « travailleurs », chargés de traiter les éléments de la file d'attente.

Il faut fixer arbitrairement le nombre de travailleurs, ainsi que la capacité de la file d'attente. Pour plus de clarté, on nomme ces constantes.

Quelques élèves ont noté que le fait d'utiliser une file d'attente de capacité bornée peut conduire à un « *deadlock* ». En effet, si la file est pleine et si tous les « travailleurs » sont bloqués par un appel à `put`, alors la situation ne pourra pas évoluer. Il peut sembler préférable d'utiliser une file d'attente dont la capacité n'est bornée que par la mémoire disponible. Encore faudra-t-il, dans ce cas, réfléchir à ce qui se passera si un appel à `put` échoue par manque de mémoire. □

Dans la suite, on suppose que l'on dispose d'un objet spécial `poison`, de type `URL`, distinct de tous les objets de type `URL` qui constituent les sommets du graphe.

Question 22 Modifiez les classes `WorkerTask` et `Traversal` de façon à ce que la méthode `traverse` attende la fin du parcours et renvoie le nombre total d'URLs découvertes pendant le parcours. Vous ferez en sorte que, une fois le parcours terminé, tous les *threads* « travailleurs » terminent. Comme lors de la question 18, vous pourrez utiliser les méthodes `join`, `sleep` (figure 5) et/ou la classe `AtomicInteger`, dont la définition est rappelée dans la figure 6. ◇

Solution. Comme lors de la question 18, on décide de maintenir un compteur du nombre de sommets qui ont été découverts mais qui n'ont pas encore été entièrement traités. Dans la solution à la question 18, ce compteur était nommé `activeThreads`. Ici, nous l'appelons `activeURLs`, parce qu'il ne mesure plus le nombre de *threads* actifs.

On voit dans les figures 14 et 15 comment ce compteur est maintenu à jour. Il est incrémenté lorsqu'un nouveau sommet est découvert (et ajouté à la file d'attente), et décrémenté lorsqu'un sommet a été entièrement traité.

Notons que la valeur de ce compteur *n'est pas* égale au nombre d'éléments de la file d'attente. En effet, un sommet est retiré de la file d'attente avant d'être complètement traité. Le nombre d'éléments de la file d'attente peut parfaitement prendre la valeur zéro alors que le parcours n'est pas terminé. Au contraire, si `activeURLs` prend la valeur zéro, alors tous les sommets découverts ont été entièrement traités, donc le parcours est terminé.

Nous adoptons la convention que, au moment où `activeURLs` est décrémenté (par un *thread* travailleur) et prend la valeur zéro, c'est ce travailleur qui porte la responsabilité de signaler cela à tous les autres travailleurs, afin que tous terminent.

Or, ces autres travailleurs sont actuellement bloqués par un appel à `take`, puisque la file d'attente est maintenant nécessairement vide. Comment leur indiquer qu'ils doivent cesser leur attente et mourir ?

L'idée, suggérée (nous l'espérons) par l'énoncé, était d'envoyer aux travailleurs un « poison » qui provoque leur mort. Puisque l'objet `poison` a le type `URL`, il peut être inséré dans la file d'attente. Nous adoptons la convention que, lorsqu'un travailleur trouve un poison dans la file d'attente, il le remet dans la file (pour que d'autres puissent le trouver également) puis meurt.

Le travailleur qui découvre que `activeURLs` a atteint la valeur zéro insère donc dans la file d'attente l'objet `poison`. Ceci provoque l'arrêt de tous les travailleurs.

Le *thread* principal attend simplement que tous les travailleurs aient terminé, puis comme le demande l'énoncé, renvoie le nombre d'URLs découvertes, `discovered.size()`. □

```
public class HashSet<E> {
  public HashSet ();
  public boolean add (E e);
  public boolean contains (E e);
  public int size ();
  // the other methods are omitted
}
```

FIGURE 2 – La classe `java.util.HashSet`

```
public interface Lock {
  void lock ();
  void unlock ();
  // the other methods are omitted
}
```

FIGURE 3 – L'interface `java.util.concurrent.locks.Lock`

```
public interface Runnable {
  void run ();
}
```

FIGURE 4 – L'interface `java.lang.Runnable`

```
public class Thread {
  public Thread (Runnable r);
  public void start ();
  public void join ();
  public static void sleep (long millis);
  // the other methods are omitted
}
```

FIGURE 5 – La classe `java.lang.Thread`

```
public class AtomicInteger {
  public AtomicInteger ();
  public int get ();
  public int decrementAndGet ();
  public int incrementAndGet ();
  // the other methods are omitted
}
```

FIGURE 6 – La classe `java.util.concurrent.atomic.AtomicInteger`

```
public interface BlockingQueue<E> {
  void put (E e);
  E take();
  // the other methods are omitted
}
```

FIGURE 7 – L'interface `java.util.concurrent.BlockingQueue`

```
// This is a simple synchronized version of HashSet,
// implemented using explicit locks.

import java.util.HashSet;
import java.util.concurrent.locks.*;

public class SafeHashSet<E> {

  // The underlying unprotected HashSet.
  private final HashSet<E> set;
  // The lock.
  private final Lock l;

  public SafeHashSet ()
  {
    this.set = new HashSet<E> ();
    this.l = new ReentrantLock ();
  }

  public boolean add (E e)
  {
    l.lock();
    try {
      return set.add(e);
    } finally {
      l.unlock();
    }
  }

  public boolean contains (E e)
  {
    l.lock();
    try {
      return set.contains(e);
    } finally {
      l.unlock();
    }
  }

  public int size ()
  {
    l.lock();
    try {
      return set.size();
    } finally {
      l.unlock();
    }
  }

}
```

FIGURE 8 – La classe `SafeHashSet` (question 12)

```
class URLTask implements Runnable {

  // The URL that we must visit.
  private final URL url;
  // The (shared) set of discovered URLs.
  private final SafeHashSet<URL> discovered;

  URLTask (SafeHashSet<URL> discovered, URL url)
  {
    this.discovered = discovered;
    this.url = url;
  }

  // The code that visits this URL.
  public void run ()
  {
    for (URL successor : url.successors())
      if (discovered.add(successor))
        new Thread (new URLTask (discovered, successor)).start();
  }

}
```

FIGURE 9 – La classe URLTask (question 13)

```
// This class implements a naive concurrent graph traversal,
// using one thread per newly-discovered vertex.

public class NaiveTraversal {

  public void traverse (URL root)
  {
    // Create a set of the discovered (marked) vertices.
    SafeHashSet<URL> discovered = new SafeHashSet<URL> ();
    // Discover the root URL.
    discovered.add(root);
    new Thread (new URLTask (discovered, root)).start();
  }

}
```

FIGURE 10 – La classe NaiveTraversal (question 14)

```
import java.util.concurrent.atomic.AtomicInteger;

class URLTaskBis implements Runnable {

  private final URL url;
  private final SafeHashSet<URL> discovered;
  // The (shared) counter of active threads.
  private final AtomicInteger activeThreads;

  URLTaskBis (SafeHashSet<URL> discovered,
     AtomicInteger activeThreads, URL url)
  {
    this.discovered = discovered;
    this.activeThreads = activeThreads;
    this.url = url;
  }

  public void run ()
  {
    for (URL successor : url.successors())
      if (discovered.add(successor)) {
        // Count the new thread that we are about to launch.
        activeThreads.getAndIncrement();
        new Thread (new URLTaskBis
          (discovered, activeThreads, successor)
        ).start();
      }
    // Count us as dead, since we are about to die.
    activeThreads.getAndDecrement();
  }

}

public class NaiveTraversalBis {

  public int traverse (URL root) throws InterruptedException
  {
    SafeHashSet<URL> discovered = new SafeHashSet<URL> ();
    // Create a counter of the number of active threads.
    AtomicInteger activeThreads = new AtomicInteger ();
    // Discover the root URL.
    discovered.add(root);
    activeThreads.getAndIncrement();
    new Thread (new URLTaskBis
      (discovered, activeThreads, root)
    ).start();
    // Wait (slowly) for the number of active threads to drop to zero.
    while (activeThreads.get() > 0)
      Thread.sleep(100);
    // Return the number of discovered URLs.
    return discovered.size();
  }

}
```

FIGURE 11 – Les classes `URLTaskBis` et `NaiveTraversalBis` (question 18)

```
import java.util.concurrent.BlockingQueue;

class WorkerTask implements Runnable {

  // The (shared) set of discovered URLs.
  private final SafeHashSet<URL> discovered;
  // A (shared) queue of URLs that must be visited.
  private final BlockingQueue<URL> queue;

  WorkerTask (SafeHashSet<URL> discovered, BlockingQueue<URL> queue)
  {
    this.discovered = discovered;
    this.queue = queue;
  }

  public void run ()
  {
    try {
      while (true) {
        URL url = queue.take();
        for (URL successor : url.successors())
          if (discovered.add(successor))
            queue.put(successor);
      }
    }
    catch (InterruptedException e) {
      // if interrupted, die
    }
  }

}
```

FIGURE 12 – La classe `WorkerTask` (question 20)

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;

// This class implements a concurrent graph traversal,
// using a fixed number of worker threads.

public class Traversal {

  // The number of worker threads.
  public static final int WORKERS = 10;     // for example
  // The capacity of the waiting queue.
  public static final int CAPACITY = 1024; // for example

  public void traverse (URL root) throws InterruptedException
  {
    // Create a set of the discovered (marked) vertices.
    SafeHashSet<URL> discovered = new SafeHashSet<URL> ();
    // Create a queue of URLs that must be processed.
    BlockingQueue<URL> queue = new ArrayBlockingQueue<URL> (CAPACITY);

    // Discover the root URL.
    discovered.add(root);
    queue.put(root);
    // Spawn the worker threads.
    for (int w = 0; w < WORKERS; w++)
      new Thread (new WorkerTask (discovered, queue)).start();
  }

}
```

FIGURE 13 – La classe Traversal (question 21)

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

class WorkerTaskBis implements Runnable {

  // The (shared) set of discovered URLs.
  private final SafeHashSet<URL> discovered;
  // A (shared) queue of URLs that must be visited.
  private final BlockingQueue<URL> queue;
  // A (shared) counter of the URLs that have been
  // discovered but not yet completely dealt with.
  private final AtomicInteger activeURLs;

  WorkerTaskBis (SafeHashSet<URL> discovered,
    BlockingQueue<URL> queue, AtomicInteger activeURLs)
  {
    this.discovered = discovered;
    this.queue = queue;
    this.activeURLs = activeURLs;
  }

  public void run ()
  {
    try {
      while (true) {
        URL url = queue.take();
        if (url.equals(URL.poison)) {
          queue.put(url); // put the poison back in the queue
          return;          // die
        }
        for (URL successor : url.successors())
          if (discovered.add(successor)) {
            activeURLs.getAndIncrement();
            queue.put(successor);
          }
        //
        if (activeURLs.decrementAndGet() == 0)
          // if there are no more active URLs,
          // then all workers should now stop.
          queue.put(URL.poison);
      }
    }
    catch (InterruptedException e) {
      // if interrupted, die
    }
  }

}
```

FIGURE 14 – La classe `WorkerTaskBis` (question 22)

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

// This class implements a concurrent graph traversal,
// using a fixed number of worker threads.

public class TraversalBis {

  // The number of worker threads.
  public static final int WORKERS = 10;     // for example
  // The capacity of the waiting queue.
  public static final int CAPACITY = 1024; // for example

  public int traverse (URL root) throws InterruptedException
  {
    // Create a set of the discovered (marked) vertices.
    SafeHashSet<URL> discovered = new SafeHashSet<URL> ();
    // Create a queue of URLs that must be processed.
    BlockingQueue<URL> queue = new ArrayBlockingQueue<URL> (CAPACITY);
    // Create a count of the URLs that have been
    // discovered but not yet completely dealt with.
    AtomicInteger activeURLs = new AtomicInteger ();

    // Discover the root URL.
    discovered.add(root);
    activeURLs.getAndIncrement();
    queue.put(root);
    // Spawn the worker threads.
    Thread[] worker = new Thread [WORKERS];
    for (int w = 0; w < WORKERS; w++) {
      worker[w] =
        new Thread (new WorkerTaskBis
          (discovered, queue, activeURLs));
      worker[w].start();
    }

    // Wait for all workers to be finished.
    for (int w = 0; w < WORKERS; w++)
      worker[w].join();

    // Return the number of discovered URLs.
    return discovered.size();
  }

}
```

FIGURE 15 – La classe `TraversalBis` (question 22)