École Polytechnique

Promotion 2010, Année 2011–2012

Algorithmique et Programmation (INF431)

Contrôle Classant 2

4 juillet 2012

François Pottier, Philippe Jacquet et Benjamin Werner

Tous les documents du cours et les notes personnelles sont autorisés. On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction.
Ce sujet est constitué de trois parties totalement indépendantes. Nous vous demandons de traiter les deux premières parties sur des feuilles de couleur **ROSE** et la troisième partie sur des feuilles de couleur **JAUNE**.

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. ◇

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

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 :

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

◇

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

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

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 :

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

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 :

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

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

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. $\diamond$

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. $\diamond$

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

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*. ◇

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.

```
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`

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*. ◇

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. ◇

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. ◇

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

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

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`. ◇

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. ◇

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 ? ◇

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 ». ◇

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.◇

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. ◇

```
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`