Algorithmique & Programmation (INF431)
Contrôle des connaissances CC2

28 juin 2013

Ce sujet est composé de trois parties entièrement **indépendantes** les unes des autres. La couleur des copies utilisées est indifférente.

Première partie

Rendu de monnaie

Dans cette partie, on demande d'écrire **du pseudo-code** aussi clair et rigoureux que possible.

Dans cette partie, on considère qu'un nombre entier occupe en mémoire un espace $O(1)$. On considère de plus que les opérations arithmétiques usuelles (à savoir addition, soustraction, multiplication, division euclidienne) ont une complexité en temps $O(1)$.

Une banque centrale a mis en circulation n types de pièces de monnaie, dont les valeurs faciales sont notées $v_1, \ldots, v_n$. Quitte à adopter une unité appropriée, on suppose que ces valeurs sont des entiers strictement positifs. On fait en sorte que ces valeurs forment une suite strictement croissante. Enfin, on suppose $n \geq 1$ et $v_1 = 1$. Dans la suite, on suppose la suite $v_1, \ldots, v_n$ fixée et connue.

Par exemple, la « zone euro » a mis en circulation des pièces de 1, 2, 5, 10, 20, 50, 100, et 200 centimes. On ignore les billets et on prend le centime comme unité. On a donc $n = 8$ et les nombres précédents forment la suite $v_1, \ldots, v_8$.

Si s est un entier positif ou nul, on note $m(s)$ le nombre minimal de pièces (de tous types) nécessaires pour former la somme s. Par exemple, dans la zone euro, $m(9)$ est égal à 3, car on peut obtenir 9 centimes à l'aide de 3 pièces (de 2, 2, et 5 centimes) et il est impossible d'obtenir cette somme à l'aide de deux pièces seulement.

Le problème du *rendu de monnaie* consiste, étant donné s, à calculer $m(s)$.

On cherche d'abord un algorithme simple et efficace capable d'exhiber **une** manière, **pas nécessairement optimale**, d'atteindre la somme s.

Question 1 Proposez un algorithme $\mathcal{G}$ qui, étant donné un entier positif ou nul s, produit un entier $\mathcal{G}(s)$ tel qu'il soit possible de former la somme s à l'aide de $\mathcal{G}(s)$ pièces de tous types. On demande que la complexité asymptotique en temps de cet algorithme, exprimée en fonction de n et s, soit $O(n)$. Ceci implique que cette complexité doit être **indépendante de** s. On souhaite que, dans le cas du système de pièces de la zone euro, cet algorithme produise toujours une solution optimale, i.e., $\mathcal{G}(s) = m(s)$. Cependant, on ne demande pas de démontrer cette affirmation. ⋄

Question 2 Ajoutez à votre algorithme $\mathcal{G}$ des instructions **afficher** « rendre k pièces de type i » (où $k \geq 0$ et $1 \leq i \leq n$) de façon à ce que l'algorithme affiche la manière dont il atteint la somme s. On permet, si besoin, que deux instructions « **afficher** » concernent un même type i. ◇

Question 3 Votre algorithme $\mathcal{G}$ produit-il toujours le résultat optimal $m(s)$, quelle que soit la suite $(v_1, \ldots, v_n)$? Justifiez votre réponse. ◇

On cherche maintenant un algorithme capable d'exhiber une manière **optimale** d'atteindre la somme s.

Question 4 Proposez un algorithme $\mathcal{M}$ qui, étant donné un entier positif ou nul s, calcule $m(s)$. Indiquez quelle est sa complexité asymptotique en temps et en espace en fonction de n et s. On exige que cette complexité soit polynomiale, c'est-à-dire de la forme $O(n^a s^b)$ pour des constantes a et b bien choisies. ◇

Question 5 Ajoutez à votre algorithme $\mathcal{M}$ des instructions **afficher** « rendre k pièces de type i » (où $k \geq 0$ et $1 \leq i \leq n$) de façon à ce que l'algorithme affiche une manière optimale d'atteindre la somme s. On permet, si besoin, que deux instructions « **afficher** » concernent un même type i. Cet ajout affecte-t-il la complexité asymptotique en temps ou en espace de l'algorithme ? ◇

Deuxième partie

Graphes et model-checking

Dans cette partie, on demande d'écrire **du pseudo-code** aussi clair et rigoureux que possible. Vous pourrez vous appuyer sur des structures de données de la librairie standard de Java, à condition de rappeler brièvement ce qu'elles font.

On désigne par *model-checking* une famille de techniques visant à vérifier formellement des systèmes informatiques (schémas de matériel, protocoles, voire programmes) en examinant chaque état du système.

Un système est spécifié sous la forme d'un état initial $\sigma_0 \in \Sigma$ et d'une relation de transition $\tau \subseteq \Sigma^2$. Ici, Σ désigne l'ensemble des états. Un état σ peut consister en un vecteur de k bits (par exemple dans le cas d'un circuit électronique doté de k bits de stockage), mais on peut avoir aussi des entiers ($\mathbb{N}$, $\mathbb{Z}$) ou d'autres types de données comme des chaînes de caractères.

Une *trace* d'exécution est une suite (finie ou infinie) d'états $\sigma_0, \sigma_1, \ldots$ telle que $\forall i, (\sigma_i, \sigma_{i+1}) \in \tau$.

On parle de vérification d'une propriété de *sûreté* quand il s'agit de montrer que certains états indésirables sont inaccessibles.

1 Model-checking énumératif

Nous supposons que tout élément de Σ peut être désigné de façon unique par une chaîne de caractères (autrement dit, deux éléments distincts de Σ sont identifiés par deux chaînes différentes ; en revanche il peut exister des chaînes ne correspondant à aucun élément de Σ). Nous supposerons que nous disposons d'une fonction qui, étant donné $\sigma \in \Sigma$, renvoie un tableau des états *successeurs* de σ, c'est-à-dire un tableau contenant les états $\sigma' \in \Sigma$ tels que $(\sigma, \sigma') \in \tau$.

Nous programmons cela en Java ainsi :

```
interface Etat {
  String identifiant();
  Etat[] successeurs();
}
```

La méthode `identifiant()` renvoie l'identifiant de l'état (rappel : deux états différents ont des identifiants différents). La méthode `successeurs()` renvoie un tableau contenant les successeurs de l'état (dans un ordre quelconque).

Dans un premier temps, **nous ne faisons aucune autre hypothèse sur l'ensemble Σ, qui peut être infini**.

Question 6 Donnez une fonction :

```
static boolean estAccessible (Etat initial, String destination)
```

Cette fonction doit satisfaire les propriétés suivantes :

1. S'il existe une trace $\sigma_0, \ldots, \sigma_n$ telle que σ_0 est l'état `initial` et σ_n est un état désigné par l'identifiant `destination`, alors `estAccessible(initial, destination)` **doit terminer** et renvoyer «vrai».
2. S'il n'existe pas de telle trace, alors `estAccessible(initial, destination)` peut terminer et renvoyer «faux», ou ne pas terminer. On impose cependant qu'il termine forcément si Σ est fini. ◇

Nous examinons maintenant le cas où non seulement Σ peut être infini, mais de plus **l'ensemble des successeurs d'un état peut être infini**. Nous devons donc modifier la signature de la méthode `successeurs()`.

```
interface Etat {
  String identifiant();
  Iterator<Etat> successeurs();
}
```

La méthode `successeurs()` fournit maintenant un *itérateur* sur les successeurs de l'état, c'est-à-dire un objet qui permet de les énumérer (dans un ordre quelconque). D'après la définition de l'interface `Iterator<Etat>`, les deux méthodes offertes par un itérateur sont :

1. `boolean hasNext()` indique si oui ou non il y a encore au moins un successeur ;
2. `Etat next()` obtient le prochain successeur (ou lève une exception s'il n'y en a plus).

Question 7 Même question que la précédente, dans ce nouveau cas où l'ensemble des successeurs d'un état peut être infini. Nous vous rappelons que l'algorithme **doit terminer lorsqu'il existe une trace vers la destination**, même si certains sommets ont un nombre infini de successeurs. Expliquez le fonctionnement de l'algorithme. ◇

2 Model-checking implicite

On suppose maintenant que Σ est de la forme $\{0,1\}^k$, pour un entier k fixé. (Donc, en particulier, Σ **est fini**.) On s'intéresse toujours au graphe orienté dont les sommets sont les états et dont les arêtes sont données par la relation τ.

La relation τ est représentée de façon compacte sous forme d'une fonction T qui attend $\sigma, \sigma' \in \{0,1\}^k$ et qui renvoie 1 si $(\sigma, \sigma') \in \tau$ et 0 sinon.

Nous ne définissons pas ici précisément comment T est représentée. Nous nous contentons de supposer qu'on peut lui associer un entier naturel N, sa «taille», tel que $k \leq N$. Nous supposons qu'évaluer $T(\sigma, \sigma')$ demande un temps $O(N)$ et un espace $O(N)$. Ceci correspond par exemple au cas où T est représentée par un circuit de N portes logiques dont les entrées sont les $2k$ booléens représentant σ et σ'.

Question 8 Comment peut-on programmer une méthode `successeurs` conforme à la première version de l'interface `Etat`, qui précède la question 6 ? On rappelle que cette méthode attend $\sigma \in \{0,1\}^k$ et renvoie un tableau des $\sigma' \in \{0,1\}^k$ tels que $(\sigma, \sigma') \in \tau$. Il est inutile de de fournir du pseudo-code. ◇

Question 9 Quelle est la complexité en temps et en espace des algorithmes classiques de parcours en largeur d'abord et de parcours en profondeur d'abord, lorsqu'on les applique au graphe qui nous intéresse ? Exprimez-la en fonction de k et N. Justifiez. ◇

Nous voudrions maintenant un algorithme permettant de déterminer s'il existe un chemin d'un état σ à un état σ' et de complexité en espace *polynomiale* (en fait, quadratique) vis-à-vis de N.

Question 10 Donnez un algorithme qui, étant donnés σ, σ', d ($\sigma, \sigma' \in \{0,1\}^k$, $d \in \mathbb{N}$), détermine s'il existe de σ à σ' un chemin de longueur au plus 2^d. Sa complexité en espace doit être $O(dk+N)$. Nous vous suggérons très fortement de procéder par récurrence sur d. Vous pourrez soit donner du pseudo-code, soit fournir une explication claire et précise de ce que l'algorithme fait. Justifiez de sa complexité en espace. ◇

Question 11 Déduisez-en un algorithme de complexité en espace polynomiale vis-à-vis de N qui, étant donnés deux états σ et σ', détermine s'il existe un chemin de σ à σ'. Justifiez sa complexité. ◇

Troisième partie

Barrières

Dans cette partie, on demande d'écrire non pas du pseudo-code, mais **du code Java**.

On s'intéresse à l'algorithme de Floyd et Warshall, dont on souhaite construire une version parallélisée.

Étant donnée une matrice g de dimensions $n \times n$ à coefficients dans $\mathbb{Z} \cup \{\infty\}$, l'algorithme de Floyd et Warshall calcule successivement les matrices d_k, pour k variant de 0 inclus à n inclus, définies par les équations suivantes :

$$\begin{aligned} d_0(i,j) &= g(i,j) \\ d_{k+1}(i,j) &= \min\begin{pmatrix} d_k(i,j) \\ d_k(i,k) + d_k(k,j) \end{pmatrix} \end{aligned}$$

Les opérations + et min sont celles de $\mathbb{Z} \cup \{\infty\}$. Le résultat de l'algorithme est la matrice d_n.

La figure 1 présente une version de cet algorithme en Java. La valeur ∞ est représentée par `Integer.MAX_VALUE`. La fonction `add` effectue une addition dans $\mathbb{Z} \cup \{\infty\}$. La fonction `line` initialise la ligne i de la matrice d_{k+1}. Elle est paramétrée par `src` et `dst`, qui représentent les matrices d_k et d_{k+1}, ainsi que par les entiers `k` et `i`. La fonction principale `fw` alloue d'abord suffisamment d'espace pour stocker les matrices d_k. Ensuite, elle initialise d_0. Enfin, pour k variant de 0 inclus à n exclus, elle calcule d_{k+1} à l'aide des données contenues dans d_k.

```
static int add (int x, int y)
{
  if (x == Integer.MAX_VALUE || y == Integer.MAX_VALUE)
    return Integer.MAX_VALUE;
  else
    return x + y;
}

static void line (int[][] src, int[][] dst, int k, int i)
{
  final int n = src.length;
  for (int j = 0; j < n; j++)
    dst[i][j] = Math.min(
      src[i][j],
      add(src[i][k], src[k][j])
    );
}

static int[][] fw (int[][] g)
{
  final int n = g.length;
  final int[][][] d = new int [n + 1][n][n];

  d[0] = g;

  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      line(d[k], d[k + 1], k, i);

  return d[n];
}
```

FIGURE 1 – L'algorithme de Floyd et Warshall

On rappelle qu'une matrice de dimension 2 est représentée en Java par un tableau de tableaux. Une matrice à coefficients entiers de taille $m \times n$ est allouée à l'aide de l'expression `new int [m][n]`.

Les questions 12 et 13 concernent deux modifications que l'on souhaite apporter à ce code. Elles sont indépendantes l'une de l'autre.

Question 12 Le code de la figure 1 alloue $n + 1$ matrices en mémoire en plus de la matrice `g` qui est donnée initialement. Modifiez la fonction `fw` de la figure 1 pour allouer exactement une matrice en plus de la matrice `g`. Ainsi, l'algorithme modifié utilise exactement deux matrices, dont `g`. Il lui est permis de modifier le contenu de la matrice `g`. Une fois le calcul terminé, il renvoie (l'adresse de) la matrice qui représente d_n.

On souligne que **l'on souhaite bien utiliser deux matrices** et non pas une seule. En d'autres termes, les arguments `src` et `dst` que l'on fournit à la fonction `line` doivent être deux matrices stockées à des emplacements distincts en mémoire. Cela sera nécessaire plus loin. ◇

Question 13 On fixe une constante `T`, entier strictement positif. On souhaite que la seconde boucle, où `i` évolue de 0 inclus à n exclus, soit sub-divisée en `T` segments de tailles approximativement égales. Modifiez la fonction `fw` de la figure 1 pour remplacer cette boucle par deux boucles imbriquées, à savoir une première boucle où un indice `t` désigne l'un des `T` segments,

puis une seconde boucle où un indice `i` évolue à l'intérieur du segment désigné par `t`.

On ne demande pas de combiner cette modification avec celle que vous avez proposée lors de la question précédente. On demande de repartir de la fonction `fw` de la figure 1. ◇

On met maintenant de côté les deux modifications étudiées dans les questions 12 et 13. On y reviendra lors de la question 17.

Pour les questions 14 à 17, par souci de concision, **on propose (et on exige) d'utiliser Java auquel on a ajouté `spawn` et `sync`**. L'instruction `spawn { ... }` crée et lance un nouveau thread. L'instruction `sync` attend la terminaison des threads lancés plus haut par `spawn`. Ainsi, par exemple, le code pseudo-Java suivant :

```
for (int t = 0; t < T; t++)
  spawn { System.out.println("Je suis le thread " + t); }
sync;
```

s'écrirait ordinairement en Java :

```
// Allocate an array of thread identifiers.
Thread[] thread = new Thread [T];
// Create and start the threads.
for (int t = 0; t < T; t++) {
  final int value_of_t = t;
  thread[t] = new Thread (new Runnable () {
    public void run () {
      int t = value_of_t;
      System.out.println("Je suis le thread " + t);
    }
  });
  thread[t].start();
}
// Wait for them to finish.
for (int t = 0; t < T; t++)
  thread[t].join();
```

On rappelle que chaque thread a ses propres variables locales, qui ne peuvent être ni lues ni modifiées par les autres threads. Dans l'exemple ci-dessus, on voit que le thread créateur (le code à l'extérieur de `spawn`) a une variable locale nommée `t`. Le thread créé (le code à l'intérieur de `spawn`) a également une variable locale nommée `t`, qui en pseudo-Java n'est pas déclarée explicitement. Ces variables sont **distinctes**. Cependant, nous adoptons la convention que la valeur de la première est copiée, au moment où le thread est créé, vers la seconde. C'est pourquoi le code pseudo-Java ci-dessus a un sens, et chaque thread affiche bien son propre identifiant, bien que la variable locale `t` du thread créateur soit modifiée (elle varie de 0 à `T`).

Question 14 Modifiez la fonction `fw` de la figure 1 pour que la boucle interne (qui concerne `i`) soit exécutée en parallèle. À chaque itération de la boucle externe (qui concerne `k`), il faut donc créer `n` threads, dont chacun traite une valeur de `i`, puis attendre leur terminaison. Quelles structures en mémoire sont partagées entre les différents threads ? Justifiez pourquoi votre code ne provoque pas de race condition. ◇

Le code écrit lors de la question 14 présente un inconvénient. La création et la destruction d'un thread coûtent cher. Or, à chaque itération de la boucle externe, donc pour chaque valeur de `k`, on crée puis on détruit `n` threads. Il semble naïf de détruire `n` threads pour en recréer aussitôt `n` nouveaux ; pourquoi plutôt ne pas les laisser continuer ? Il suffirait alors de créer `n` threads tout au début et de les détruire tout à la fin.

```
static int[][] fwpara (int[][] g)
{
  final int n = g.length;
  final int[][][] d = new int [n + 1][n][n];

  d[0] = g;

  for (int i = 0; i < n; i++)
    spawn {
      for (int k = 0; k < n; k++)
        line(d[k], d[k + 1], k, i);
    }
  sync;

  return d[n];
}
```

FIGURE 2 – Une version modifiée de l'algorithme (question 15)

```
public class Barrier {
  public Barrier (int n);
  public void await (int phase);
}
```

FIGURE 3 – Squelette de la classe `Barrier`

Question 15 La figure 2 présente une version de l'algorithme modifiée pour utiliser `n` threads auxiliaires en tout. Expliquez pourquoi ce code est incorrect. ◇

Afin de réparer ce problème, on se donne un nouveau mécanisme de synchronisation, connu sous le nom de **barrière**. Une barrière est destinée à faciliter la synchronisation entre `n` threads. Intuitivement, elle permet de bloquer ceux qui « arrivent les premiers » à la barrière : elle ne s'ouvre que lorsque les `n` threads sont arrivés, et les débloque alors tous d'un seul coup.

En Java, cette notion peut être représentée par une classe `Barrier` (figure 3) qui offre deux opérations :

1. L'opération **`new`** permet de créer une nouvelle barrière. On fixe alors la valeur de `n`, qui indique combien de threads partageront cette barrière.
2. La méthode `await` permet à l'un des `n` threads participants d'indiquer qu'il a « atteint la barrière ». Il est alors bloqué jusqu'à ce que **tous** les threads participants aient appelé `await`. Le paramètre `phase` est utile si la barrière doit servir plusieurs fois. On adopte la convention que lorsqu'un thread atteint la barrière pour la première fois, il appelle `await(0)` ; la fois suivante, `await(1)` ; et ainsi de suite.

Question 16 À l'aide d'une barrière, corrigez le code de la figure 2. ◇

Question 17 Combinez le code que vous avez proposé lors de la question précédente avec les idées des questions 12 et 13 de façon à obtenir une version de l'algorithme qui utilise en tout deux matrices et `T` threads. ◇

On souhaite maintenant construire plusieurs implémentations de la classe `Barrier`, à l'aide de différents mécanismes étudiés pendant le cours.

Pour les questions 18 et 19, on fait l'hypothèse que **la barrière ne sert qu'une fois**. Autrement dit, si une barrière `b` a été créée par `new Barrier (n)`, alors on attend que chacun des `n` threads appelle exactement une fois `b.await(0)`, après quoi la barrière `b` n'est plus utilisée.

On rappelle que la méthode `decrementAndGet` de la classe `AtomicInteger` (figure 4) décrémente le compteur et renvoie sa nouvelle valeur, le tout de façon atomique. De même, `incrementAndGet` incrémente le compteur et renvoie sa nouvelle valeur, le tout de façon atomique. On souligne qu'un objet de classe `AtomicInteger` **peut** être utilisé simultanément par plusieurs threads ; cela n'est pas considéré comme une race condition.

Question 18 À l'aide de la classe `AtomicInteger` (figure 4), proposez une implémentation de la classe `Barrier` (figure 3). L'attente active est autorisée. ◇

On change maintenant d'outil et on souhaite implémenter une barrière à l'aide d'un verrou et d'une variable de condition.

Question 19 À l'aide de la classe `ReentrantLock` et des interfaces `Lock` et `Condition` (figure 5), et sans utiliser la classe `AtomicInteger`, proposez une implémentation de la classe `Barrier` (figure 3). ◇

On suppose maintenant que **la barrière sert plusieurs fois**, c'est-à-dire que chacun des `n` threads appelle `await(0)`, puis `await(1)`, etc.

Question 20 Modifiez votre solution à la question 19 de façon à ce que la barrière puisse servir plusieurs fois. ◇

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

FIGURE 4 – La classe `AtomicInteger`

```
public class ReentrantLock implements Lock {
  // methods omitted
}
public interface Lock {
  void lock ();
  void unlock ();
  Condition newCondition();
  // the other methods are omitted
}
public interface Condition {
  void awaitUninterruptibly ();
  void signal ();
  void signalAll ();
  // the other methods are omitted
}
```

FIGURE 5 – La classe `ReentrantLock` et les interfaces `Lock` et `Condition`