Programme de colle 2

Thibault Mabillotte

2 Questions de Cours exigibles

1. Démontrer que le problème de l'arrêt est indécidable

Définition : On définit le problème de l'arrêt de la façon suivante :

entrée : un programme p et une entrée e.

sortie : vrai si et seulement si l'exécution du programme p sur l'entrée e termine.

Théorème : Le problème de l'arrêt est indécidable.

Démonstration : Supposons par l'absurde que le problème de l'arrêt est décidable. Il existe alors une fonction, qu'on suppose écrite en C, **bool** arret(**char**$*$ p, **char**$*$ e) telle que pour tout programme p et toute entrée e, la fonction renvoie vrai si et seulement si l'exécution du programme p sur l'entrée e termine.

Remarque : La fonction arret prend en entrée des chaînes de caractères car un programme ou une entrée est représenté en machine par des fichiers binaires.

On considère alors le programme écrit en C suivant :

```
void paradoxe(char* p) {
  if ( arret(p,p) ) {
    while(true) {}
  }
}
```

On s'intéresse à l'appel paradoxe(paradoxe).

Si l'appel paradoxe(paradoxe) termine.
La conditionnelle arret(p,p) avec p = paradoxe est vrai car l'appel à la fonction paradoxe sur l'entrée paradoxe termine par hypothèse. Ainsi, la fonction lance une boucle while infinie. L'appel ne termine donc pas.

Si l'appel paradoxe(paradoxe) ne termine pas.
La conditionnelle est fausse car l'appel à la fonction paradoxe sur l'entrée paradoxe ne termine par hypothèse. La fonction ne rentre donc pas dans la boucle while et l'appel termine donc immédiatement.

Ainsi, l'appel à la fonction paradoxe sur l'entrée paradoxe s'arrête si et seulement si il ne s'arrête pas. C'est une absurdité donc la fonction arret n'existe pas.

2. Justifier qu'il existe une infinité de problèmes indécidables à l'aide d'un argument de cardinalité

Théorème : Il existe une infinité de fonctions de la forme $f : D \to \mathbb{B}$ pour lesquelles il n'existe pas d'algorithme qui produise en sortie $f(x)$ pour chaque entrée $x \in D$. Autrement dit, il existe une infinité de problèmes indécidables.

Démonstration : Pour montrer qu'il existe une infinité de fonction de la forme $f : D \to \mathbb{B}$ pour lesquelles aucun programme ne peut calculer f on va montrer que l'ensemble $\mathcal{F}(D, \mathbb{B})$ n'est pas dénombrable et que l'ensemble des programmes est dénombrable.

Soit D un ensemble dénombrable non fini (les problèmes de décision sont définis sur des ensembles dénombrables). Montrons que l'ensemble $\mathcal{F}(D, \mathbb{B})$ n'est pas dénombrable. Comme D est dénombrable non fini, montrer que $\mathcal{F}(D, \mathbb{B})$ n'est pas dénombrable revient à montrer que $F := \mathcal{F}(\mathbb{N}, \mathbb{B})$ n'est pas dénombrable.

Remarque : On prend D non fini car, comme nous l'avons vu dans le cours, un problème de décision défini sur un ensemble fini est toujours décidable.

Supposons que F soit dénombrable. Alors il existe une suite de fonction $(f_n) \in F^{\mathbb{N}}$ telle que $\forall f \in F, \exists n \in \mathbb{N}, f = f_n$. On pose alors la fonction :

$$\tilde{f} : \left\{ \begin{array}{ccc} \mathbb{N} & \to & \mathbb{B} \\ n & \mapsto & \neg f_n(n) \end{array} \right.$$

$\tilde{f}$ est bien une fonction de $\mathbb{N}$ dans $\mathbb{B}$ donc appartient à F.

Soit $p \in \mathbb{N}$. $\tilde{f}(p) = \neg f_p(p) \neq f_p(p)$. Autrement dit, il existe $n \in \mathbb{N}$ tel que $\tilde{f}(n) \neq f_p(n)$. Ainsi, $\tilde{f} \neq f_p$. On trouve donc que $\forall n \in \mathbb{N}, \tilde{f} \neq f_p$ donc $\tilde{f} \notin F = \{f_n \,|\, n \in \mathbb{N}\}$. C'est absurde car, par construction, $\tilde{f} \in F$. On en déduit que F n'est pas dénombrable.

Un programme est un code écrit dans un langage de programmation quelconque. Autrement dit, un programme est une chaine de caractère. Montrons que l'ensemble des chaînes de caractères est dénombrable.

Soit Σ un alphabet. Notons C l'ensemble des chaînes de caractères sur Σ. Si $n \in \mathbb{N}$, on note C_n l'ensemble des chaînes de caractères de taille n. On a alors l'égalité suivante :

$$C = \bigsqcup_n C_n$$

De plus, $\text{card}(C_n) = \text{card}(\Sigma)^n$ donc C_n est de cardinal fini donc C_n est dénombrable. C une union dénombrable d'ensembles dénombrables donc C est dénombrable.

Ainsi, l'ensemble des programmes étant dénombrable et l'ensemble des fonctions de D dans $\mathbb{B}$ n'étant pas dénombrable, il existe une infinité de fonction $f : D \to \mathbb{B}$ pour lesquelles il n'existe pas d'algorithme qui produisent en sortie $f(x)$ pour tout $x \in D$.

Cela signifie donc bien qu'il existe une infinité de problèmes de décision indécidables.

Remarque : Madame Monfleur est passée très vite sur cette démonstration dans le cours. Elle n'a pas prouvé le caractère dénombrable de l'ensemble des programmes et le caractère indénombrable de l'ensemble des fonctions de la forme $f : D \to \mathbb{B}$. Dans le doute, ne prouvez pas ces points là et attendez que le colleur vous dise de le faire.

3. Monter que P $\subset$ NP

Définition : Soit $\mathcal{P}$ un problème de décision associé à $f : D \to \mathbb{B}$. On dit que $\mathcal{P}$ est vérifiable s'il existe une fonction $v : D \times \Sigma^* \to \mathbb{B}$ telle que :

$$\forall x \in D,\ f(x) = \text{Vrai} \iff \exists c \in \{0,1\}^*,\ v(x,c) = \text{Vrai}$$

v est appelé un vérificateur et c un certificat. Autrement dit, un problème est vérifiable s'il existe un vérificateur qui détermine si un certificat est bien une solution du problème.

Définition : Soit $\mathcal{P}$ un problème de décision associé à $f : D \to \mathbb{B}$. On dit que $\mathcal{P}$ appartient à la classe **NP** si :

1. $\mathcal{P}$ est vérifiable avec un certificat dont la taille est polynomiale en la taille de l'entrée associée.
2. La fonction de vérification correspond à un problème de décision de la classe **P**.

Théorème : **P** $\subset$ **NP**.

Preuve : Soit $f : D \to \mathbb{B}$ et $\mathcal{P}$ le problème de décision associé appartenant à la classe **P**. Montrons que $\mathcal{P} \in$ **NP**.

$\mathcal{P} \in$ **P** donc il existe un algorithme avec une complexité polynomiale qui calcul f. On pose alors la fonction :

$$v : \left\{ \begin{array}{ccc} D \times \{0,1\}^* & \to & \mathbb{B} \\ (x,c) & \mapsto & f(x) \end{array} \right.$$

Comme f est calculable avec une complexité polynomiale, on en déduit que v aussi et donc que $v \in$ **P**.

Montrons que v est un vérificateur pour $\mathcal{P}$. Soit $x \in D$. Si $f(x) = \text{Vrai}$ alors $v(x,\epsilon) = f(x) = \text{Vrai}$. Réciproquement, si $f(x) = \text{Faux}$ alors $\forall c \in \{0,1\}^*,\ v(x,c) = f(x) = \text{Faux}$.
Ainsi, v est bien un vérificateur pour $\mathcal{P}$.

En particulier, comme $|\epsilon| = \mathcal{O}(|x|)$, $\mathcal{P}$ est vérifiable avec un certificat de taille polynomiale en la taille des entrées de $\mathcal{P}$.

Ainsi, $\mathcal{P}$ est un problème vérifiable par un vérificateur de la classe **P** avec des certificats de taille polynomiale. Donc $\mathcal{P} \in$ **NP**.

4. Monter que s'il existe un problème NP-complet dans la classe P alors P = NP

Théorème : Soient $\mathcal{P}_1$ et $\mathcal{P}_2$ deux problèmes de décision associés à $f_1 : D_1 \to \mathbb{B}$ et $f_2 : D_2 \to \mathbb{B}$. Si $\mathcal{P}_1 \leq_p \mathcal{P}_2$ et $\mathcal{P}_2 \in \mathbf{P}$ alors $\mathcal{P}_1 \in \mathbf{P}$.

Démonstration : Montrons que $\mathcal{P}_1 \in \mathbf{P}$. $\mathcal{P}_2 \in \mathbf{P}$ donc il existe un algorithme A_2 calculable en temps polynomiale qui résout le problème $\mathcal{P}_2$. $f_1 \leq_P f_2$ donc il existe un algorithme A calculable en temps polynomiale qui transforme $x \in D_1$ en $g(x) \in D_2$.

On construit l'algorithme A_1 de la façon suivante :

- entrée : $x \in D_1$ une instance du problème $\mathcal{P}_1$.

1 Applique A sur x qui renvoie $g(x)$.

2 Applique A_2 sur $g(x)$ qui renvoie $f_2\big(g(x)\big)$.

- sortie : $f_2\big(g(x)\big) = f_1(x)$.

Justifions que A_1 est bien polynomiale sur la taille de x.
À l'opération (1), l'algorithme calcul $g(x)$ en appliquant l'algorithme A à x ce qui a un coup $P(|x|)$. À l'opération (2), l'algorithme calcul $f_2\big(g(x)\big)$ en appliquant l'algorithme A_2 à $g(x)$ ce qui à à un coup $P_2(|g(x)|)$.
Le coup total de l'algorithme A est donc $P(|x|) + P(|g(x)|)$. Pour justifier que A_1 est polynomiale en la taille de x, il faut montrer que $g(x)$ est de taille polynomiale en celle de x.

L'algorithme A calcul $g(x)$ en $P(|x|)$ opérations. Or, pour créer $g(x)$, il faut l'encoder en binaire dans la mémoire de l'ordinateur ce qui se fait $|g(x)|$ opérations. Comme ces opérations sont comptées parmi les $P(|x|)$ opérations de l'algorithme A on peut donc écrire que :

$$|g(x)| \leq P(|x|)$$

Le coup total de l'algorithme A_1 est donc inférieur à $P(|x|) + P_2\big(P(|x|)\big) = \big(P + P_2 \circ P\big)(|x|)$ ce qui est bien polynomiale en $|x|$.

Ainsi, l'algorithme A_1 calcul $f_1(x)$ en un temps polynomiale en la taille de x donc $\mathcal{P}_1 \in \mathbf{P}$.

Définition : Un problème est NP-complet s'il vérifie les deux propriétés suivantes :

1. Il appartient à la classe **NP**.
2. Tout problème de décision de la classe **NP** se réduit polynomialement à lui.

Théorème : Si un problème de la classe **P** est NP-complet alors **P** = **NP**.

Démonstration : Soient $\mathcal{P}_0 \in \mathbf{P}$ un problème NP-complet. Comme $\mathbf{P} \subset \mathbf{NP}$ il suffit de monter que $\mathbf{NP} \subset \mathbf{P}$.

Montrons que $\mathbf{NP} \subset \mathbf{P}$. Soit $\mathcal{P} \in \mathbf{NP}$.
$\mathcal{P}_0$ est NP-complet donc tout problème **NP** se réduit polynomialement à lui. En particulier $\mathcal{P} \leq_P \mathcal{P}_0$. Or $\mathcal{P}_0$ appartient à **P** donc $\mathcal{P}$ appartient aussi à la classe **P**.
$\mathcal{P} \in \mathbf{P}$ donc $\mathbf{NP} \subset \mathbf{P}$.

Ainsi, **P** = **NP**.

5. Donner l'algorithme de Peterson et montrer qu'il garanti l'exclusion mutuelle

Définition : Un mutex est une "clef" qu'un thread doit détenir pour rentrer en section critique (on dit qu'un thread prend un mutex). Un mutex ne peut être détenu que par un seul thread à la fois.

Un mutex est associé à deux fonctions : lock et unlock. Quand un thread exécute la fonction lock, soit il prend le mutex et rentre en section critique soit il attend que le mutex soit disponible. Une fois la section critique terminé, le thread qui possédait le mutex exécute la fonction unlock et rend le mutex. Les autres threads peuvent alors le récupérer.

L'algorithme de Peterson est un algorithme qui implémente un mutex pour deux threads. Cependant, cet algorithme a recourt à l'attente active : un thread bloqué par le mutex travail dans une boucle while plutôt que d'être mis en veille. Il n'est donc pas efficace en pratique.

L'algorithme de Peterson utilise deux variables globales, un tableau de booléen want qui détermine si un thread désire entrer en section critique et un entier turn qui détermine le thread qui est en droit de rentrer en section critique.

```
int turn = 0;
bool want[2] = {false, false};
// initialement aucun thread ne veut acceder a la section critique
void lock(int i) {
  int other = 1-i;
  want[i] = true;
  turn = other;
  while (want[other] && turn == other) {}
  // tant que thread i-1 veut acceder a la section critique et que c'est son tour
  // le thread i attend
}
void unlock(int i) {
  want[i] = false;
  // quand le thread i a fini de travailler dans la section critique
  // il informe le mutex qu'il n'a plus besoin d'y acceder afin de le liberer
}
```

Montrons que cette solution garantit l'exclusion mutuelle, c'est-à-dire que les deux threads ne peuvent pas accéder à la section critique simultanément.

On note T_0 le premier thread et T_1 le second. Supposons par l'absurde que les deux threads accèdent à la section critique simultanément. On définit et numérote arbitrairement l'ensemble des actions que les deux threads vont réaliser lors de l'exécution :

1. T_0 pose want[0]=**true**.
2. T_1 pose want[1]=**true**.
3. T_0 pose turn = 1.
4. T_1 pose turn = 0.
5. T_0 lit want[1].
6. T_1 lit want[0].
7. T_0 lit turn.
8. T_1 lit turn.

On note $i \prec j$ pour dire que l'action i set réalisée avant l'action j. Ainsi, comme les threads exécutent le code de manière séquentielle, on sait que $1 \prec 3 \prec 5 \prec 7$ et $2 \prec 4 \prec 6 \prec 8$.

Supposons sans perte de généralité que $3 \prec 4$. C'est-à-dire que T_0 pose turn $= 1$ avant que T_1 pose turn $= 0$. Comme l'action 6 et 8 se déroulent après l'action 4, on sait qu'au moment où T_1 lit want[0] et turn, turn vaut 0.

Par hypothèse, T_1 est rentré en section critique donc la conditionnelle want[0] && turn=0 est fausse. Or, comme nous l'avons vu, à ce moment là turn est égal à 0 donc T_1 lit forcément que want[0] est faux. Autrement dit, $6 \prec 1$ car T_0 est aussi en section critique par hypothèse.

Ainsi, on a $1 \prec 3 \prec 4 \prec 6 \prec 1$. On trouve une contradiction donc T_0 et T_1 n'accèdent pas à la section critique simultanément.

6. Donner l'algorithme de la boulangerie de Lamport et montrer qu'il garanti l'exclusion mutuelle

L'algorithme de la Boulangerie de Lamport est un algorithme qui permet d'obtenir l'exclusion mutuelle d'un nombre quelconque de threads. Tout comme l'algorithme de Peterson, il utilise l'attente active et n'a donc aucun intérêt pratique.

Cet algorithme utilise un système similaire à une fil d'attente dans un commerce : chaque thread reçoit un ticket et attend son tour pour entrer en section critique.

```
bool want[n];
int label[n];

void lock(int i) {
  want[i] = true;
  // phase de recuperation du numero de passage
  int number = 0;
  for (int k=0;k<n;k+=1) {
    if (number < label[k]) {
      number = label[k];
    }
  }
  label[i] = number+1;
  // phase d'attente de son tour
  for (int k=0;k<n;k+=1) {
    if (k!=i) {
      while ( want[k] && ( label[k] < label[i] || ( label[k] == label[i] && k <i ) ) ) {}
    }
  }
}

void unlock(int i) {
  want[i] = false;
}
```

Montrons que cette solution garanti l'exclusion mutuelle, c'est-à-dire que deux threads ne peuvent pas accéder à la section critique simultanément.

Commençons par montrer un lemme : à chaque appel à la fonction lock par un thread k, la valeur de label[k] augmente.

Montrons ce lemme. Par construction de la fonction lock, la nouvelle valeur de label[k] vaut $1 + \max_{1 \leq i \leq n} \text{label}[i]$. Or, $k \in [\![1, n]\!]$ donc label[k] $\geq 1 +$ label[k]. La valeur de label[k] a donc bien augmenté.

Montrons que cet algorithme garanti l'exclusion mutuelle. Soient T_i et T_j deux threads. Supposons par l'absurde que T_i et T_j rentrent en section critique simultanément.

Supposons sans perte de généralité qu'une fois entrée en section critique, label[i], $i <$ label[j], j pour l'ordre lexicographique.
La conditionnelle de la boucle while impose que pour rentrer dans la section critique un thread doit constater qu'aucun autre thread ne veut y rentrer ou qu'il est le premier dans la liste d'attente. Si un thread ne respecte une de ces deux conditions il ne peut pas rentrer en section critique.

Supposons que T_j ai lu un ordre différent de label [i] , i < label [j] , j. T_j a donc lu une valeur non actualisée de label [i]. Or, d'après le lemme, cette valeur non actualisée est nécessairement plus petite. Donc T_j constate que label [i] , i < label [j] , j ce qui est contraire à ce qu'on vient de supposer.

Il est donc impossible que T_i et T_j constate un ordre contradictoire. Autrement dit, T_j n'a pas constaté qu'il était premier dans la liste d'attente.

La seule option restante et que T_j ai lu want[i]=false. Or, comme la modification du tableau want par un thread se fait avant l'attribution du label et que T_i et T_j doit être en section critique simultanément par hypothèse, cela signifie que T_i n'a pas encore obtenu son label et va donc en obtenir un plus grand que celui de T_j. Cependant, cela contredit l'ordre label [i] , i < label [j] , j.

Autrement dit, T_j n'a pas constaté que T_i ne désirer pas rentrer dans la section critique.

On peut donc conclure que T_j n'est pas rentré en section critique. C'est absurde car T_i et T_j sont supposé être en section critique simultanément. Ainsi T_i et T_j ne peuvent pas entrer en section critique simultanément. Il y a bien exclusion mutuelle.

7. Présenter une solution au problème producteur-consommateur

Définition : Le problème producteurs/consommateurs consiste en une mémoire de capacité c_{max} partagée entre des producteurs qui y écrivent et des consommateurs qui l'effacent avec les règles suivantes :

- On ne peut pas produire si la mémoire est pleine.
- On ne peut pas consommer si la mémoire est vide.

On choisit de modéliser la mémoire par une file représentée dans l'ordinateur avec un tableau de taille c_max. On propose une solution basée sur deux sémaphore, cases_libres et case_occupees.

Un thread producteur attend que le sémaphore cases_libres soit strictement positif pour écrire puis incrémente le sémaphore case_occupees pour réveiller un thread consommateur qui peut alors consommer.

Un thread consommateur attend que le sémaphore cases_occupees soit strictement positif pour effacer puis incrément le sémaphore cases_libres pour réveiller un thread consommateur qui peut alors produire.

```
mutex m;
sem_t cases_libres;
sem_t cases_ccupees;

sem_init(&cases_libres,0,cmax)
// le semapore cases_libres est initialise a cmax
sem_init(&cases_occupees,0,0)
// le semaphore cases_occupees est initialise a 0

int memoire[cmax];
int cpt = 0;

void producteur() {
  while (true) {
    wait(&cases_libres);
    lock(&m);
    memoire[cpt] = 1;
    cpt += 1;
    unlock(&m);
    post(&cases_occupees);
  }
}

void consommateur() {
  while (true) {
    wait(&cases_occupees);
    lock(&m);
    memoire[cpt-1] = 0;
    cpt -= 1;
    unlock(&m);
    post(&cases_libres);
  }
}
```

Remarque : Pour faciliter la lecture de l'algorithme, le nom des fonctions associées aux mutex et aux sémaphores ont été simplifiées.

Remarque : La présence de mutex autour de la section critique permet d'éviter que deux threads producteurs écrivent dans la même case ou que deux threads consommateurs effacent la même case. Ils sont donc nécessaires.