# I Décidabilité

On rappelle que le problème de l'arrêt est indécidable :

**ARRET**
**Instance** : une fonction $f$ et un argument $x$
**Question** : $f$ termine-t-elle sur l'entrée $x$ ?

Pour chacun des problèmes suivants, dire s'il est décidable ou non en le prouvant.
On supposera avoir une machine universelle permettant d'exécuter une fonction pour un temps donné.

1. **ARRET-N**
   **Instance** : une fonction $f$, un argument $x$ et un entier $n$.
   **Question** : est-ce que l'exécution de $f(x)$ termine en moins de $n$ secondes ?

   Solution : Décidable : Il suffit de simuler l'exécution de $f(x)$ pendant $n$ secondes.

2. **SAC-A-DOS**
   **Instance** : un ensemble d'objets de poids $p_1, \dots, p_n$ et de valeurs $v_1, \dots, v_n$, un poids maximal $P$ et une valeur minimale $V$.
   **Question** : existe-t-il un sous-ensemble d'objets de poids total inférieur à $P$ et de valeur totale supérieure à $V$ ?

   Solution : Décidable, car on peut tester tous les sous-ensembles possibles (en nombre fini).

3. **ZERO**
   **Instance** : une fonction $f$ et un argument $x$.
   **Question** : est-ce que $f(x)$ renvoie 0 ?

   Solution : Indécidable, en montrant `ARRET` $\leq$ `ZERO` :
   Soit une instance `(f, x)` de `ARRET`. On lui associe la fonction `let g y = f x; 0`.
   Alors $\Psi$ : `(f, x)` $\mapsto$ `(g, x)` est une réduction de `ARRET` à `ZERO` : elle est calculable et `(f, x)` est une instance positive de `ARRET` (`f x` termine) si et seulement si `(g, x)` est une instance positive de `ZERO` (`g x` renvoie 0).
   Comme `ARRET` est indécidable, `ZERO` l'est donc aussi.

4. Question supplémentaire : Montrer que `ZERO` $\leq$ `ARRET`.

   Solution : Soit une instance `(f, x)` de `ZERO`. On construit la fonction :

   ```
   let g y =
         if f x <> 0 then
               while true do () done (* boucle infinie *)
   ```

   Alors `f x` renvoie 0 si et seulement si `g x` termine. Donc $\Psi$ : `(f, x)` $\mapsto$ `(g, x)` est bien une réduction de `ZERO` à `ARRET`.

# II Semi-décidabilité

Un problème de décision est dit semi-décidable s'il existe un algorithme qui :

- répond correctement (et en temps fini) pour toutes les instances positives du problème ;
- répond correctement ou ne termine pas pour les instances négatives.

On demande simplement que l'algorithme ne réponde jamais « Oui » pour une instance négative : il peut répondre « Non » pour certaines de ces instances et ne pas terminer pour d'autres.

Pour un problème de décision $\Pi$, on définit le problème complémentaire de $\Pi$, noté co-$\Pi$, comme le problème de décision ayant les mêmes instances que $\Pi$, mais des réponses opposées. Autrement dit, les instances positives de co-$\Pi$ sont exactement les instances négatives de $\Pi$, et les instances négatives de co-$\Pi$ sont exactement les instances positives de $\Pi$.

1. Le problème `ARRET` est-il semi-décidable ?
2. Montrer qu'un problème $\Pi$ est décidable si et seulement si $\Pi$ et co-$\Pi$ sont tous les deux semi-décidables.
3. Le problème co-`ARRET` est-il semi-décidable ?
4. `ARRET`$_\forall$

   **Instance** : une fonction $f$
   **Question** : le calcul de $f(x)$ termine-t-il pour tout $x$ ?

   (a) Ce problème est-il décidable ?

   (b) Est-ce que co-`ARRET`$_\forall$ est semi-décidable ?

   (c) Est-ce que `ARRET`$_\forall$ est semi-décidable ?

Solution :

1. Le problème de l'arrêt (`ARRET`) est semi-décidable :

```
let arret f x =
      f x;
      true
```

   Si $f$ termine sur l'entrée $x$ (instance positive), l'appel `arret f x` renvoie `true`. Si $f$ ne termine pas sur l'entrée $x$, l'appel `arret f x` ne termine pas non plus.

2. $\Rightarrow$ Si $\Pi$ est décidable, alors $\Pi$ est évidemment semi-décidable (par le même algorithme décidant $\Pi$). De plus, si $\Pi$ est décidable, alors co-$\Pi$ est également décidable (il suffit d'inverser la réponse), donc semi-décidable.

   Soient $A_1$ et $A_2$ deux algorithmes permettant respectivement de semi-décider $\Pi$ et co-$\Pi$.
   On peut alors exécuter les deux algorithmes alternativement :

   Algorithme $A$

   **Entrée :** une instance $x$ de $\Pi$
   `n <- 0`
   **Tant que** `Vrai` **:**
   - `n <- n + 1`
   - Exécuter $A_1(x)$ pendant `n` secondes
   - **Si** $A_1$ a répondu « Oui » **:**
     - **Renvoyer** « Oui »
   - Exécuter $A_2$ pendant `n` secondes
   - **Si** $A_2$ a répondu « Oui » **:**
     - **Renvoyer** « Non »

   Si $x$ est une instance positive de $\Pi$, alors $A_1(x)$ va répondre « Oui » en un temps fini, et l'algorithme $A$ va donc répondre « Oui » en un temps fini. Si $x$ est une instance négative de $\Pi$, alors $A_2(x)$ va répondre « Oui » en un temps fini, et l'algorithme $A$ va donc répondre « Non » en un temps fini. Ainsi, l'algorithme $A$ décide bien le problème $\Pi$.

3. On sait que `ARRET` n'est pas décidable et que `ARRET` est semi-décidable : la contraposée de la question précédente montre que co-`ARRET` n'est pas semi-décidable.

4. (a) Supposons que `ARRET`$_\forall$ soit décidable. La fonction suivante décide alors le problème de l'arrêt :

```
let arret f x =
      arret_pour_tout (fun y -> f x)
```

   Donc `ARRET`$_\forall$ n'est pas décidable.

(b) Supposons que l'on dispose d'une fonction `coarret_pour_tout` qui semi-décide ce problème. On définit alors :

```
let coarret f x =
      coarret_pour_tout (fun y -> f x)
```

Cette fonction semi-décide co-ARRET, qui n'est pas semi-décidable : absurde.

(c) Supposons que l'on dispose d'une fonction `arret_pour_tout` qui semi-décide ce problème. On peut alors considérer l'algorithme suivant :

- Entrée : une instance $(f, x)$ de co-ARRET
- Algorithme :
  Définir la fonction $g$ qui prend en entrée un entier $n$ et qui :
  - simule les $n$ premières étapes du calcul de $f(x)$ ;
  - si ce calcul a terminé (en au plus $n$ étapes, donc), boucle à l'infini ;
  - sinon, termine (et ne renvoie rien).

  Renvoyer le résultat de l'appel `arret_pour_tout g`

On remarque que :

- si $(f, x)$ est une instance positive de co-ARRET (c'est-à-dire si $f$ ne termine pas sur l'entrée $x$), alors $g$ est une instance positive de $\text{ARRET}_\forall$, et comme la fonction `arret_pour_tout` semi-décide ce problème, l'algorithme renvoie « Oui » en temps fini ;
- si $(f, x)$ est une instance négative de co-ARRET, alors $g$ est une instance négative de $\text{ARRET}_\forall$, donc l'algorithme renvoie « Non » ou ne termine pas.

Ainsi, on a obtenu un algorithme qui semi-décide co-ARRET : ce problème n'étant pas semi-décidable, c'est absurde. On en déduit que $\text{ARRET}_\forall$ n'est pas semi-décidable.

# III Castor affairé

On considère une version idéalisée du langage OCaml où la mémoire est illimitée et le type `int` permet de représenter des entiers arbitrairement grands.

On dit qu'une fonction $f : \mathbb{N} \longrightarrow \mathbb{N}$ est calculable s'il existe une fonction OCaml `f : int -> int`, dont l'exécution termine pour tout argument positif ou nul, qui calcule les images par $f$.

On appelle programme une expression OCaml de type `int`. Si l'évaluation de cette expression termine (sans erreur), on dit que le programme calcule la valeur de l'expression. On suppose que les programmes OCaml sont écrits en utilisant les 128 caractères ASCII, et l'on appelle taille d'un programme son nombre de caractères.

Par exemple, les deux programmes suivants calculent respectivement les valeurs 13 et 120 :

```
7 + 6
```

```
let rec fact n =
      if n = 0 then 1
      else n * fact (n - 1)
      in
      fact 5
```

alors que le programme ci-dessous ne termine pas, et ne calcule donc pas de valeur :

```
let rec fact n = n * fact (n - 1) in
fact 5
```

Un castor affairé (busy beaver en anglais) est un programme dont l'exécution termine et qui calcule une valeur la plus grande possible parmi tous les programmes de même taille. On définit le problème de décision CASTOR :

**CASTOR**

**Instance** : le code source d'un programme OCaml $P$
**Question** : $P$ est-il un castor affairé ?

Par exemple, le programme suivant :

```
let k=99 in k*k*k
```

est un programme de taille 17, qui termine et renvoie 970299. Ce n'est pas un castor affairé, puisque le programme suivant (qui n'est toujours pas un castor affairé) renvoie une valeur plus grande :

```
99999999999999999
```

Pour un entier $n \geq 0$, on notera $C(n)$ la valeur maximale renvoyée par un programme OCaml de taille $n$ qui termine et renvoie un entier. On pose $C(0) = 0$ par convention.

On cherche à montrer que la fonction $C$ n'est pas calculable, et que le problème `CASTOR` n'est pas décidable.

1. Combien existe-t-il de programmes OCaml à $n$ caractères, en supposant qu'on dispose de 128 caractères différents possibles ? Expliquer pourquoi le fait qu'il y en ait un nombre fini ne permet pas de conclure que la fonction $C$ est calculable.
2. Montrer que la fonction $C$ est correctement définie.
3. Montrer que $C$ est une fonction croissante, puis que pour $n \in \mathbb{N}$, on a $C(n+2) > C(n)$.
4. Que vaut $C(1)$ ?

On considère $f : \mathbb{N} \longrightarrow \mathbb{N}$ une fonction calculable.

5. Montrer qu'il existe un entier $k$ tel que pour tout $n \in \mathbb{N}$, $f(n) \leq C(\lfloor \log_{10} n \rfloor + k)$.
6. Montrer que $C$ n'est pas calculable.
7. En déduire que `CASTOR` n'est pas décidable.
8. Montrer, en utilisant la non-calculabilité de $C$, que le problème de l'arrêt est indécidable. On ne demande pas simplement de prouver la non-décidabilité du problème de l'arrêt, comme cela a pu être fait dans le cours, mais bien de la déduire de la non-calculabilité de $C$.

Solution :

1. Le nombre de programmes contenant $n$ caractères est $128^n$ (qui est bien fini). Malheureusement, on ne peut pas se contenter de tous les exécuter et ne garder que celui qui s'exécute sans erreur, termine son calcul en temps fini et renvoie un entier, le plus grand possible, car certains de ces programmes peuvent avoir une exécution qui ne termine pas.
2. La fonction $C$ est correctement définie, car chaque l'exécution de chaque programme peut soit terminer en temps fini, soit ne pas terminer (il n'y a pas d'entre deux, les programmes contenant des erreurs terminent en temps fini). De plus, le programme contenant $n$ répétitions du caractère `1` est correct, termine en temps fini et renvoie un entier. Ainsi, l'ensemble des entiers renvoyés par des programmes de taille $n$ qui terminent est un ensemble fini, non vide, inclus dans $\mathbb{Z}$, il possède donc un maximum.
3. On peut rajouter une espace à la fin d'un programme sans changer le déroulé de son exécution (l'espace sera ignorée). On en déduit que pour $n \in \mathbb{N}^*$, $C(n+1) \geq C(n)$ : si on dispose d'un castor affairé de taille $n$, alors il existe un programme de taille $n+1$ qui renvoie le même entier, qui est donc inférieur ou égal à l'entier renvoyé par un castor affairé de taille $n+1$.
   Par ailleurs, en ajoutant `+1` (deux caractères) à la fin d'un castor affairé de taille $n$, on obtient un programme de taille $n+2$ renvoyant $C(n)+1$. On en déduit que $C(n+2) \geq C(n)+1$.
4. Les seuls programmes corrects de taille 1 sont ceux constitués d'exactement un chiffre, donc $C(1) = 9$. Cela ne contredit pas la non-calculabilité, car cela ne donne pas de méthode générale permettant de calculer $C(n)$ pour tout entier $n$.
5. Notons $m = \lfloor \log_{10} n \rfloor$, et $n = (c_m c_{m-1} c_1 c_0)_{10}$ l'écriture décimale de $n$. $f$ étant calculable, il existe une fonction OCaml commençant par `let f x =` permettant de définir une fonction calculant $f(x)$, dont l'exécution termine toujours. Notons $k_0$ la taille de son code source. En le complétant par `in f` $c_m c_{m-1} c_1 c_0$, on obtient un programme de taille $k_0 + 2 + m + 1$, dont l'exécution termine toujours et qui calcule $f(n)$. En posant $k = k_0 + 3$, on obtient un programme de taille $m + k$ qui renvoie un entier, donc cet entier est inférieur ou égal à l'entier renvoyé par un castor affairé de taille $m + k$, soit $f(n) \leq C(m + k)$.

6. On a $\dfrac{n}{\lfloor \log_{10} n \rfloor + k + 2} \underset{n \to +\infty}{\longrightarrow} +\infty$, donc pour $n_0$ assez grand, $n_0 \geq \lfloor \log_{10} n_0 \rfloor + k + 2$. Pour un tel $n_0$, par la question 3, on a :

$$f(n_0) \leq C(\lfloor \log_{10} n_0 \rfloor + k) < C(\lfloor \log_{10} n_0 \rfloor + k + 2) \leq C(n_0)$$

On a donc $f(n_0) < C(n_0)$, d'où $f \neq C$. Comme ce résultat est valable pour toute fonction calculable $f$, on en déduit que $C$ n'est pas calculable.

7. Supposons le problème décidable, et considérons alors l'algorithme suivant, qui prend en entrée un entier $n \geq 0$ :
   - on énumère tous les programmes de taille $n$ ;
   - pour chacun de ces programmes, on décide s'il s'agit d'un castor affairé (cette décision se fait en temps fini par hypothèse) ;
   - si c'est le cas, on l'exécute (en temps fini, puisque c'est un castor affairé) à l'aide d'une machine universelle et l'on renvoie son résultat.

   Cet algorithme renvoie toujours une valeur (puisqu'il existe un castor affairé de taille $n$) en temps fini d'après les remarques, et cette valeur est exactement $C(n)$. Cela contredit la non-calculabilité de cette fonction, donc le problème `CASTOR` n'est pas décidable.

8. Supposons que le problème `ARRET` est décidable. Alors le programme suivant pourrait calculer $C(n)$, pour tout $n$ :
   - on énumère (sans les exécuter) tous les programmes OCaml de taille $n$ ;
   - on décide, à l'aide d'un programme qui résout `ARRET`, lesquels ont une exécution en temps fini ;
   - on exécute les programmes qui terminent à l'aide d'une machine universelle, en gardant en mémoire les valeurs renvoyées, si ce sont des entiers ;
   - on renvoie le maximum de tous ces entiers.

   Ainsi, le problème du castor affairé serait calculable. C'est faux, donc `ARRET` n'est pas décidable.

# IV Classes de complexité

On rappelle que P $\subset$ NP $\subset$ EXP $\subset$ Décidables.
Donner la classe de complexité la plus précise possible des problèmes suivants :

1. `REGEXP-EQUIV`

   Instance : deux expressions régulières $e_1$ et $e_2$.
   Question : $L(e_1) = L(e_2)$ ?

   Solution : On peut :
   - transformer $e_1$ et $e_2$ en automates $A_1$ et $A_2$ avec l'algorithme de Thompson ou de Berry-Sethi (complexité polynomiale)
   - déterminiser $A_1$ et $A_2$ (complexité exponentielle)
   - construire des automates produits $A'_1$ et $A'_2$ reconnaissants $L(A_1) \cap \overline{L(A_2)}$ et $L(A_2) \cap \overline{L(A_1)}$ (complexité polynomiale)
   - tester si $L(A'_1) = \emptyset$ et $L(A'_2) = \emptyset$ en cherchant un chemin d'un état initial vers un état final par parcours en profondeur (complexité polynomiale)

   D'où `REGEXP-EQUIV` $\in$ EXP.

2.

`CHEMIN-≤`

Instance : un graphe $G = (S, A)$, deux sommets $s, t \in S$ et un entier $k$.
Question : existe-t-il un chemin élémentaire de $s$ à $t$ de longueur $\leq k$ ?

Solution : `CHEMIN-≤` $\in$ P avec un parcours en largeur.

3.

`CHEMIN-≥`

Instance : un graphe $G = (S, A)$, deux sommets $s, t \in S$ et un entier $k$.
Question : existe-t-il un chemin élémentaire de $s$ à $t$ de longueur $\geq k$ ?

Solution : `CHEMIN-≥` $\in$ NP, où un certificat est un chemin de longueur supérieur ou égal à $k$, qui est bien de taille polynomiale. On peut vérifier un certificat en temps polynomial : il suffit de vérifier que le chemin est élémentaire et de longueur supérieure ou égale à $k$.

4.

`CHEMIN-≥-ARBRE`

Instance : un arbre $G = (S, A)$.
Question : existe-t-il un chemin élémentaire de $s$ à $t$ de longueur $\geq k$ ?

Solution : On peut enraciner $G$ puis calculer le diamètre $d$ (c'est-à-dire la distance maximum entre deux sommets, qui est aussi la longueur maximum d'un chemin élémentaire) de $G$ avec une fonction récursive en complexité linéaire.
Il suffit ensuite de tester si $d \geq k$.

5.

`CHEMIN-HAMILTONIEN`

Instance : un graphe $G = (S, A)$.
Question : $G$ admet-il un chemin hamiltonien, c'est-à-dire un chemin passant exactement une fois par chaque sommet ?

Solution : `CHEMIN-HAMILTONIEN` $\in$ NP, où un certificat est un chemin hamiltonien. On peut vérifier un certificat en temps polynomial : il suffit de vérifier que le chemin passe exactement une fois par chaque sommet.

6.

`COUPLAGE-PARFAIT-BIPARTI`

Instance : un graphe biparti $G = (S, A)$.
Question : $G$ admet-il un couplage parfait ?

Solution : On calcule un couplage maximum avec l'algorithme des chemins augmentants (en complexité $O(|S||A|)$), et on vérifie que sa taille est $\frac{|S|}{2}$. Donc `COUPLAGE-PARFAIT-BIPARTI` $\in$ P.

# V $k$-COLOR

Soit $G = (S, A)$ un graphe non orienté. On appelle $k$-coloration de $G$ une fonction $c : S \longrightarrow \{1, 2, \ldots, k\}$ telle que : $\forall \{u, v\} \in A$, $c(u) \neq c(v)$.
Pour $k \in \mathbb{N}^*$, on considère le problème suivant :

**$k$-COLOR**

Entrée : un graphe $G = (S, A)$ non orienté
Sortie : $G$ est-il $k$-colorable ?

1. Montrer que `1-COLOR` et `2-COLOR` appartiennent à P.

> Solution : `1-COLOR` est vrai si et seulement si le graphe ne contient aucune arête. `2-COLOR` est vrai si et seulement si le graphe est biparti, ce qui peut être testé en temps linéaire (parcours en profondeur en alternant les couleurs).

2. Montrer que `3-COLOR` appartient à NP.

> Solution : On peut vérifier en temps polynomial qu'une coloration est correcte en parcourant les arêtes du graphe.

3. Montrer que `3-COLOR` se réduit polynomialement à `3-SAT`.

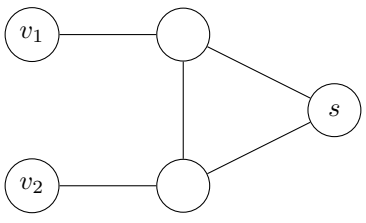
> Solution : Soit $G = (S, A)$ un graphe non orienté. Pour chaque sommet $u \in S$, on créé des variables $x_{u,1}, x_{u,2}, x_{u,3}$ ($x_{u,i}$ va être vrai ssi le sommet $u$ est colorié avec la couleur $i$).
> Pour chaque sommet $u$, on ajoute la clause $(x_{u,1} \vee x_{u,2} \vee x_{u,3})$ (chaque sommet doit être colorié) et $\neg x_{u,j} \vee \neg x_{u,j'}$ pour $j \neq j'$ (un sommet ne peut pas être colorié avec deux couleurs différentes).
> Pour chaque arête $(u, v) \in A$, on ajoute les clauses $\neg x_{u,i} \vee \neg x_{v,i}$ (deux sommets adjacents ne peuvent pas être coloriés de la même couleur).
> La formule $\phi$ obtenue par conjonction de ces clauses est une instance de `3-SAT` et est satisfiable si et seulement si $G$ est 3-colorable. De plus, la construction et la taille de $\phi$ sont polynomiales en la taille de $G$.

Dans la suite, on veut trouver une réduction polynomiale de `3-SAT` à `3-COLOR`.
On considère une formule $\varphi$ de `3-SAT` de variables $x_1, ..., x_n$ et on veut construire un graphe $G$ qui soit 3-colorable si et seulement si $\varphi$ est satisfiable.
On ajoute $n$ sommets dans $G$ (encore appelés $x_1$, ..., $x_n$ par abus de notation) correspondant à $x_1$, ..., $x_n$, $n$ sommets correspondant à $\neg x_1$, ..., $\neg x_n$ et 3 sommets $V, F, B$ reliés deux à deux.
Dans un 3-coloriage de $G$, $S$ et $F$ doivent être de couleurs différentes. Chaque variable $x_i$ sera considérée comme fausse si le sommet correspondant est de la même couleur que $F$ et vraie s'il est de la même couleur que $V$.

4. Expliquer comment ajouter des arêtes à $G$ pour que chaque variable $x_i$ soit vraie ou fausse (c'est-à-dire coloriée avec la même couleur que $F$ ou la même couleur que $V$) et de valeur opposée à $\neg x_i$.

> Solution : Pour chaque $i$, on relie $x_i$, $\neg x_i$ et $B$ deux à deux. Ainsi, $x_i$ est coloriée avec la même couleur que $V$ ou $F$ et $\neg x_i$ avec la couleur opposée.

On considère un sous-graphe (*gadget*) de la forme suivante à ajouter dans $G$ :

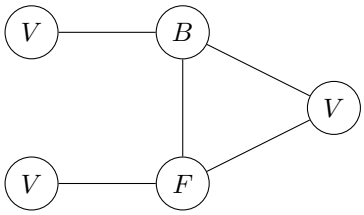



5. Montrer que si $v_1$ et $v_2$ sont de la même couleur que $F$ alors la couleur de $s$ est imposée et préciser cette dernière.

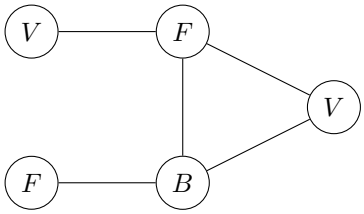
> Solution : On trouve que $s$ doit être de la même couleur que $F$.

6. Montrer que si $v_1$ ou $v_2$ est de la même couleur que $V$ alors il existe un coloriage de $G$ où $s$ est de la même couleur que $V$.

Solution : Si $v_1 = v_2 = V$, on peut utiliser le coloriage :




Si $v_1 = V$ et $v_2 = F$ (le cas $v_1 = F$ et $v_2 = V$ étant symétrique):

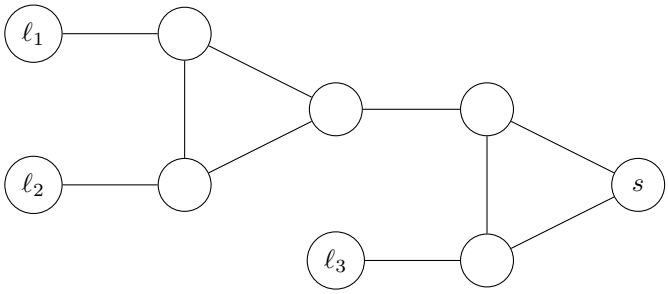



7. Quelle formule logique le gadget ci-dessus permet-il de représenter ?

Solution : $v_1 \vee v_2$.

8. Quel gadget ajouter à $G$ de façon pour représenter une clause $\ell_1 \vee \ell_2 \vee \ell_3$ ?

Solution : Comme $\ell_1 \vee \ell_2 \vee \ell_3 = (\ell_1 \vee \ell_2) \vee \ell_3$, on connecte la sortie du gadget de $\ell_1 \vee \ell_2$ à l'entrée du gadget de $\ell_3$ :




9. Montrer que 3-COLOR est NP-complet.

Solution : Pour chaque clause $\ell_1 \vee \ell_2 \vee \ell_3$ de $\varphi$, on ajoute un gadget comme ci-dessus ainsi qu'une arête entre $s$ et $B$ et une arête entre $s$ et $F$, pour forcer la valeur de $s$ à $V$.
Soit $G$ le graphe obtenu. Montrons que $G$ est 3-colorable si et seulement si $\varphi$ est satisfiable.

- Supposons $\phi$ satisfiable. Alors il existe une valuation mettant au moins un littéral de chaque clause à vrai. On colorie les sommets correspondants à ces littéraux avec $V$ et les autres avec $F$. Alors chaque gadget possède au moins un sommet en entrée avec la couleur $V$ donc il peut-être colorié d'après la question 6.
- Supposons que $G$ est 3-colorable. Alors chaque gadget est colorié correctement. On peut alors construire une valuation $v$ en prenant vrai pour chaque littéral correspondant à un sommet colorié avec $V$ et faux pour les autres.
  Chaque gadget a une sortie coloriée avec $V$ donc au moins un sommet en entrée colorié avec $V$. Donc chaque clause possède au moins un littéral à vrai : cette valuation satisfait $\varphi$.

10. Montrer que $k$-COLOR est NP-complet pour $k \geq 4$.

Solution : On montre que $k$-COLOR $\in$ NP en utilisant une coloration comme certificat.
Montrons $k$-COLOR $\leq_p (k+1)$-COLOR.
Soit $G$ une instance de $k$-COLOR. Soit $G'$ le graphe obtenu en ajoutant un sommet $s$ et en le reliant à tous les sommets de $G$. Alors $G$ est $k$-colorable si et seulement si $G'$ est $(k+1)$-colorable.
Ainsi $k$-COLOR $\leq_p (k+1)$-COLOR. Comme 3-COLOR est NP-complet, on en déduit par récurrence immédiate que $k$-COLOR est NP-complet pour tout $k \geq 4$.