# Fondements de l'informatique. Examen
# Durée: 3h

*Sujet proposé par Olivier Bournez*
*Version 4*
*(corrigé)*

*Les 4 parties sont indépendantes, et peuvent être traitées dans un ordre quelconque. On pourra admettre le résultat d'une question pour passer aux questions suivantes. On pourra utiliser tous les résultats et les théorèmes démontrés ou énoncés en cours ou en petite classe ou dans le polycopié sans chercher à les redémontrer.*

*Il est possible d'avoir la note maximale sans répondre à toutes les questions. La difficulté des questions n'est pas une fonction linéaire ni croissante de leur numérotation.*

*La* ***qualité et clarté de votre argumentation et de votre rédaction*** *sera une partie importante de votre évaluation.*

## 1 Décidabilité

On considère des machines de Turing qui travaillent sur un alphabet qui contient au moins les chiffres $0, 1, \ldots, 9$.

*Indication : aucune propriété avancée sur les décimales de $\pi$ n'est nécessaire pour répondre à toutes les questions qui suivent.*

**Question 1.** *Peut-on décider si le langage accepté par une machine de Turing $M$ est constitué du mot qui correspond aux 42 premières décimales de $\pi$.*

*Solution :* C'est indécidable par le Théorème de Rice : il y a au moins une machine qui le fait, et au moins une machine qui ne le fait pas, et donc la propriété est non-triviale. □

**Question 2.** *On considère le problème de décision suivant :*
- ***Donnée****: Un entier $k$*
- ***Réponse****: Décider s'il y a $k$ zéros consécutifs dans l'écriture décimale de $\pi$.*

*Est-il décidable ? Dans NP ? Dans P ?*

*Solution :* Premier cas : il y a une limite au nombre de 0 consécutifs dans l'écriture décimale de $\pi$, disons $n$. Autrement dit, il y a une suite de $n$ zeros, mais pas de suite de $n+1$ zéros. Dans de cas, le problème est juste de comparer l'entrée $k$ à $n$ et d'accepter si $k \leq n$.

Second cas : il n'y a pas de telle limite, dans ce cas, une machine qui accepte toujours résout le problème.

Je ne sais pas lequel des deux cas est le bon, mais peu importe : c'est donc toujours décidable, et dans P et dans NP. □

On dira qu'une machine de Turing $M$ accepte une entrée $w$ en temps quadratique si elle termine en temps $n^2$ où $n$ est la longueur de $w$.

**Question 3.** *Etant donnés un mot $w$, et une machine de Turing $M$, peut-on décider si $M$ accepte $w$ en temps quadratique ?*

*Solution :* Il suffit de simuler $M$ pendant $n^2$ étapes pour connaître la réponse. □

**Question 4.** *Peut-on décider si une machine de Turing $M$ accepte toutes ses entrées, et ce, en temps quadratique ?*

*Solution :* Le problème est indécidable. On peut réduire le problème de l'arrêt à ce problème : étant donnée une machine $A$ et un mot $w$, on considère le programme $A_w$ qui, sur une entrée $u$, simule $A$ sur $w$, et si $A$ accepte $w$, accepte. Sinon, la simulation fonctionne à jamais.

Par construction $A$ accepte $w$, si et seulement $A_w$ accepte toutes ses entrées en temps quadratique (constant même). La fonction qui à $w$ associe $A_w$ est bien calculable. □

**Question 5.** *On se donne une machine de Turing $M$ qui termine sur toutes ses entrées. Peut-on décider la machine de Turing $M$ accepte toutes ses entrées en temps quadratique ?*

*Solution :* C'est indécidable. On peut par exemple utiliser le théorème du point fixe pour considérer une machine qui connait son propre code. Si on suppose que l'on sait décider la propriété par une machine de Turing $D$, la machine appelle $D$ sur son propre code, et si $D$ lui dit qu'elle accepte toutes ses entrées en temps quadratique, alors elle prend un temps non-quadratique mais fini sur les mots de la forme (disons) $0^n$, et sinon, elle accepte immédiatement (et donc en temps quadratique, puisque constant). On obtient alors une contradiction, et donc la machine de Turing $D$ ne peut pas exister. □

## 2 Sur la nécessité d'autoriser des boucles infinies

Un langage de programmation est dit **Turing complet** si pour tout langage $L$ décidable par une machine de Turing, il y a un programme de ce langage de programmation qui décide $L$.

*Motivation : Le but de cette section est de prouver qu'il faut nécessairement permettre la possibilité de faire des boucles infinies dans les langages de programmation, sinon on n'est pas Turing-complet. Les langages comme Java, C et OCaml sont Turing-complets.*

Nous supposons qu'un langage de programmation vérifie au minimum les hypothèses suivantes :

1. le langage de programmation permet une notion de décision : il y a un critère [1] qui permet de définir "acccepte" (et donc son complémentaire "refuse") pour le résultat de l'exécution d'un programme *lorsqu'il termine* sur une entrée donnée $w$ ;
2. une machine de Turing $M_1$ peut vérifier qu'un texte correspond bien à un programme écrit avec ce langage de programmation ;
3. une machine de Turing $M_2$ est capable de le simuler : si on a un texte qui correspond bien à un programme écrit avec ce langage de programmation, et une entrée $w$, la machine $M_2$ avec le texte de ce programme et $w$ simule [2] l'exécution du programme sur l'entrée $w$.

**Question 6.** *Supposons qu'un certain langage de programmation ne permet pas de faire des boucles infinies : ses programmes terminent toujours sur toute entrée.*

*Montrer qu'il est impossible que ce langage de programmation soit Turing complet.*

*Indication : On pourra raisonner par l'absurde, et construire un langage $L$ décidable [3] tel qu'aucun programme du langage de programmation ne peut le décider.*

1. Par exemple, le fait qu'une certaine variable vaille 1, qu'il affiche "oui", etc...
2. Et en particulier, quand le programme termine, $M_2$ est capable de détecter si le résultat de son exécution satisfait le critère de "accepte" ou de son contraire "refuse".
3. Par machine de Turing.

*Solution :* On suppose par l'absurde que le langage de programmation est Turing complet. On considère la machine de Turing qui sur une entrée $w$ fait la chose suivante : elle vérifie si $w$ est bien un programme du langage de programmation. Si ce n'est pas le cas, elle accepte. Sinon, elle simule le programme $w$ sur l'entrée $w$. Cela doit terminer par hypothèse. Si la simulation accepte, elle refuse et symétriquement.

On considère le langage $L$ associé à cette machine : c'est un langage décidable par définition, puisque décidé par une machine de Turing qui termine toujours. Il doit y avoir un programme $w$ dans le langage de programmation qui le décide. La question de savoir si $w \in L$ ou pas, mène dans tous les cas à une contradiction. □

# 3 Programmes booléens

*Motivation : Les programmes booléens constituent un exemple de langage de programmation qui n'est pas Turing complet par la section précédente. Cependant, nous allons montrer qu'ils suffisent pour défnir ce qu'est un problème NP-complet, et même suffisent à donner une preuve alternative au théorème de Cook-Levin.*

Un *programme Booléen* est un programme avec un nombre fini d'instructions $I_1; I_2; \ldots; I_m$ où chaque instruction $I$ et expression $E$, et variable $X$ est définie de façon inductive [4] par les règles suivantes :

$$\begin{array}{lcl} I & \rightarrow & X := E | I_1; I_2 | if\ E\ then\ I_1\ else\ I_2 \\ E & \rightarrow & X | true | false | E_1 \vee E_2 | E_1 \wedge E_2 | \neg E_1 \\ X & \rightarrow & X_0 | X_1 | \ldots | X_v \end{array}$$

pour un nombre fini $v$ de variables [5] $X_1, \ldots, X_v$.

Les programmes Booléens travaillent sur des variables qui sont booléennes : elles ne peuvent prendre que la valeur 1 (vrai) ou 0 (faux). Ils fonctionnent avec la sémantique attendue : par exemple, $X := Y \vee Z$ met dans la variable $X$ le résultat du *ou logique* sur la valeur de la variable $Y$ et $Z$. L'instruction $I; I'$ signifie effectuer l'instruction $I$ puis l'instruction $I'$.

*Remarque : Les programmes Booléens ne permettent pas de faire des boucles : si l'on se donne un tel programme, il va effectuer un nombre fini $m$ d'instructions et s'arrêter.*

On s'intéresse au problème de décision suivant :
SAT-PROGRAMME-BOOL
— **Donnée**: Un programme Booléen
— **Réponse**: Déterminer s'il existe des valeurs pour chacune des variables $X_1, \ldots, X_m$ (dans $\{0, 1\}$) telles que si l'on exécute le programme avec ces valeurs initiales, alors à la fin de l'exécution du programme la variable $X_1$ vaut 1.

**Question 7.** *Prouver que le problème* SAT-PROGRAMME-BOOL *est NP-complet. On pourra utiliser la NP-complétude du problème* 3-SAT.

*Solution :* Le problème est dans NP car la donnée des valeurs initiales constitue un certificat vérifiable en temps polynomial : il suffit de propager les valeurs selon les règles du programme.

Le problème 3-SAT se réduit à ce problème : il suffit de construire un programme qui évalue une formule 3-SAT en fonction de la valeur de ses variables, et qui place le résultat dans $X_1$. Cela constituera une réduction de 3-SAT vers ce problème, et cela se fait bien en temps polynomial. □

4. La première ligne veut dire par exemple, que l'ensemble des instructions est le plus petit ensemble de mots, tel que $X := E$ est une instruction, et si $I_1$ et $I_2$ sont des instructions, alors $I_1; I_2$ est une instruction, et si $E$ est une expression, alors $if\ E\ then\ I_1\ else\ I_2$ est une instruction.

5. Comme dans l'exemple qui suit, et dans la suite, pour aider à la lisibilité, nous nous autoriserons à appeler ces variables par d'autres noms, comme $X, Y, Z, S, Go, etc.$.

*Motivation : Nous allons chercher à obtenir dans le reste de cette section une preuve alternative du Théorème de Cook-Levin, et à montrer directement la NP-complétude du problème* SAT-PROGRAMME-BOOL.

**Question 8.** *Considérons une machine de Turing $V$, sur l'alphabet $\Sigma = \{0, 1\}$, qui fonctionne en temps polynomial : en temps $\pi(n)$ sur les entrées de taille $n$ pour un polynôme $\pi$. Montrer que pour tout $n$, on peut construire un programme Booléen $\mathcal{P}_n$ tel que pour chaque entrée $w = w_1 \dots w_n$, si l'on exécute ce programme $\mathcal{P}_n$ avec initialement $X_1 = w_1$, ..., $X_n = w_n$, alors ce programme se terminera avec $X_1 = 1$ si et seulement si $V$ accepte le mot $w$.*

Indication : On fixera un codage des configurations de $V$ par des variables booléennes que l'on précisera, et on écrira le programme Booléen $\mathcal{P}_n$ de la forme

$$Init; UneEtape; UneEtape; UneEtape; \dots; UneEtape$$

où $Init$ est une ou plusieurs instructions que l'on précisera, et $UneEtape$ est une ou plusieurs instructions que l'on précisera, qui sera répété un nombre de fois que l'on précisera.

*Montrer que l'on peut déterminer la description de ce programme Booléen $\mathcal{P}_n$ en temps polynomial en $n$.*

*Solution :*

Voici une possibilité. Plein d'autres possibilités sont envisageables.

Supposons que cette machine de Turing $V$ fait un nombre d'étapes au plus $\pi(n)$ pour un certain polynôme $\pi$ où $n$ désigne la taille de ses entrées.

Nous devons coder l'état interne $q \in Q$ de la machine de Turing $V$. Pour faire cela, nous introduisons des variables booléennes $Instr_1$, ..., $Instr_m$, où $m = |Q|$ avec l'idée que nous souhaitons que si $q = \ell$ alors $Instr_\ell$ aura la valeur vraie, et toutes les autres $Instr_j$ pour $j \neq \ell$ auront la valeur fausse.

On peut voir le ruban comme deux mots $L$ et $R$ : le contenu à droite et à gauche de la tête de lecture, avec la convention par exemple que la première lettre de $R$ correspond à la case en face de la tête de lecture. Les mots $L$ et $R$ sont des mots sur l'alphabet $a \in \{0, 1, B\}$. En temps $\pi(n)$, c'est-à-dire en $\pi(n)$ étapes, en raison de comment sont définis les instructions (si on préfère puisque le ruban de la machine de Turing se déplace au plus d'une case à chaque étape), nous savons que ces mots $L$ et $R$ grandissent au plus d'une lettre par étape, et restent donc de longueur inférieure à $n + \pi(n) + 1$. Posons $\pi'(n) = n + \pi(n) + 1$.

Nous codons alors le contenu du mot $R$ par des variables booléennes $R_i^a$, où $a \in \{0, 1, B\}$, et $1 \leq i \leq \pi'(n)$ : l'idée est que nous souhaitons que $R_i^a$ aura la valeur vraie si et seulement si la lettre numéro $i$ du mot $R$ est $a$ (si jamais il n'y a pas de lettre numéro $i$ dans le mot $R$, nous considérons que c'est un blanc).

De même, nous codons alors le contenu du mot $L$ par des variables booléennes $L_i^a$, où $a \in \{0, 1, B\}$, et $1 \leq i \leq \pi'(n)$ (avec la même convention que pour $R$ s'il n'y a pas de lettre numéro $i$).

Pour nous simplifier la vie, nous écrirons

— $L_i :\equiv a$ (dont l'effet est remplacer la lettre numéro $i$ de $L$ par $a$) comme synonyme de l'instruction (composée)

$$L_i^0 := false; L_i^1 := false; L_i^B := false; L_i^a := true.$$

— $R_i :\equiv a$ (dont l'effet est remplacer la lettre numéro $i$ de $R$ par $a$) comme synonyme de l'instruction (composée)

$$R_i^0 := false; R_i^1 := false; R_i^B := false; R_i^a := true.$$

— $L_i :\equiv L_j$ (dont l'effet est de remplacer la valeur de la lettre numéro $i$ de $L$ par la valeur de la lettre numéro $j$ de $L$) comme synonyme de l'instruction (composée)

$$L_i^0 := L_j^0; L_i^1 := L_j^1; L_i^B := L_j^B.$$

— $R_i :\equiv R_j$ est défini de la même façon.
— $L_i :\equiv L_j$ est défini de la même façon.

Nous considérons le programme Booléen qui commence de la façon suivante :
— $Instr_1 := true$, et $Instr_i := false$ pour $i \neq 1$ :
— $L_{\pi'(n)} :\equiv B; \dots L_1 :\equiv B;\ R_1 :\equiv X_1; \dots R_n :\equiv X_n;\ R_n :\equiv B; \dots R_{\pi'(n)} :\equiv B$
Nous souhations ensuite simuler une à une les instructions de la machine de Turing.
— Nous ajoutons alors les instructions
  — $UneEtape$
  — $UneEtape$
  — ...
  — $UneEtape$
  où $UneEtape$ apparait $\pi(n)$ fois,
où $UneEtape$ est l'instruction

$$\begin{array}{l} if\ Instr_1\ then\ \overline{I_1}\ else \\ \quad if\ Instr_2\ then\ \overline{I_2}\ else \\ \quad\quad if\ Instr_3\ then\ \overline{I_3}\ else \\ \quad\quad\quad \dots \\ \quad\quad\quad\quad if\ Instr_{m+1}\ then\ \overline{I_{m+1}} \end{array}$$

avec chaque $\overline{I}_\ell$ obtenue à partir du programme de $V$ selon la correspondance suivante :

| Instruction $I_\ell$ | Instruction $\overline{I_\ell}$ Booléen correspondante |
|---|---|
| aller vers la droite | $L_{\pi'(n)} :\equiv L_{\pi'(n)-1}; \dots; L_2 :\equiv L_1; L_1 :\equiv R_1;$<br>$R_1 :\equiv R_2; R_2 :\equiv R_3; \dots; R_{\pi'(n)} :\equiv$ B;<br>$Instr_\ell :=$ false ; $Instr_{\ell+1} :=$ true |
| aller vers la gauche | $R_{\pi'(n)} :\equiv R_{\pi'(n)-1}; \dots; R_2 :\equiv R_1; R_1 :\equiv L_1;$<br>$L_1 :\equiv L_2; L_2 :\equiv L_3; \dots; L_{\pi'(n)-1} :\equiv L_{\pi'(n)}; L_{\pi'(n)} :\equiv B;$<br>$Instr_\ell :=$ false ; $Instr_{\ell+1} :=$ true |
| écrire S | $R_1 :\equiv a; Instr_\ell := false; Instr_{\ell+1} := true$ |
| if $S$ goto $\ell'$<br>else $\ell''$ | $Instr_\ell := false;$<br>$Instr_{\ell'} := R_1^S; Instr_{\ell''} := \neg R_1^S$ |

Ces règles ne font qu'essentiellement reprendre la description de la sémantique de chacune des instructions.

Nous affirmons que ce programme est construit de telle sorte que les propriétés suivantes soient vérifiées : a chaque $1 \leq t \leq \pi(n)$, considérons l'état du programme Booléen juste avant d'effectuer l'instruction $UneEtape$. Alors $Instr_\ell$ sera 1 pour exactement un $\ell$, et pour chaque $1 \leq i \leq \pi'(n)$, la variable booléenne $L_i^a$ et la variable booléenne $R_i^a$ seront vraies pour exactement un $a$.

Cela découle du fait que c'est vrai pour $t = 1$ de la façon dont nous avons construit le programme, et que l'examen des cas du tableau ci-dessus préserve cette propriété.

Et par ailleurs, par construction, par récurrence sur le nombre d'étapes effectuées par la machine de Turing, nous obtenons que le programme simule correctement la machine de Turing.

On peut bien déterminer la description de ce programme Booléen en temps polynomial en $n$, selon les règles ci-essus. □

**Question 9.** *Prouver que le problème* SAT-PROGRAMME-BOOL *est NP-complet, sans utiliser la NP-complétude d'un autre problème connu pour être NP-complet.*

*Solution :* SAT-PROGRAMME-BOOL est dans NP car la donnée des valeurs initiales des variables constitue un certificat vérifiable en temps polynomial.

Il reste à montrer qu'il est plus difficile que tous les problèmes dans NP. Supposons en effet que l'on a un problème $A$ dans NP. Par définition, il admet un vérificateur polynomial : il y a un polynôme $q(n)$ et un vérificateur $V$ polynomial tel un mot $w$ est dans $A$ si et seulement si $V$ accepte $\langle w, u\rangle$ pour un certain certificat $u$ de taille $q(|w|)$.

Fixons un mot $w = w_1 \dots w_n$ de longueur $n$. Considérons le programme Booléen $V_w$ qui correspond au programme $V$ sur $\langle w, u\rangle$ (on voit cette paire comme le mot $wu$) donné par la question précédente : Il travaille sur certaines variables booléennes $X_1, \dots X_n$ qui correspondent aux lettres de $w$ et d'autres $X_{n+1}, \dots X_{n+q(n)}$ qui correspondent aux lettres de $u$.

Intéressons nous maintenant à comment ce programme $V_w$ dépend de $w$ : la fonction $f$ qui à $w$ associe ce programme $f(w) = V_w$ avec ses $n = |w|$ fixées aux lettres de $w$ est bien calculable en temps polynomial par la question précédente.

Par définition, nous avons que $w \in A$ si et seulement s'il existe une valeur pour les variables booléennes qui correspondent à $u$ (les variables $X_{n+1}, \dots X_{n+q(n)}$) tel que le programme $V_w$ accepte avec $X_1 = 1$. Autrement dit, $f$ réalise une réduction du problème $A$ vers le problème SAT-PROGRAMME-BOOL, et donc $A \leq$ SAT-PROGRAMME-BOOL, ce qui prouve le théorème.

□

On dira que deux instructions $I$ et $I'$ sont strictement équivalentes, si leur exécution modifie exactement les mêmes variables et de la même façon.

**Question 10.** *On considère l'instruction $\overline{I}$ définie par $X := (E \wedge Go) \vee (X \wedge \neg Go)$. Donner une instruction $I$ strictement équivalente à $\overline{I}$ lorsque la variable booléenne $Go$ est vraie. Même chose lorsque la variable booléenne $Go$ est fausse.*

*Solution :* C'est $X := E$ lorsque $Go$ est vraie, et c'est $X := X$ soit aucun effet, lorsque $Go$ est fausse. □

Plus généralement, considérons une instruction $I$, et une instruction $J$ avec possiblement plus de variables. On dira que $I$ est équivalente à $J$ si leur exécution modifie exactement les mêmes variables communes et de la même façon, et ce quel que soit la valeur initiale des variables supplémentaires de $J$. Par exemple $X := 0$ est équivalente à $S := Go; X := 0; Go := S$.

**Question 11.** *Donner une instruction équivalente à l'instruction "$if\ U\ then\ J\ else\ K$" sans utiliser d'instruction "$if\ then\ else$", construite à partir de $\overline{J}$ et $\overline{K}$.*

*Solution :* Par exemple, on peut faire $S := Go; Go := S \wedge U; \overline{J}; Go := S \wedge \neg U; \overline{K}; Go := S$. □

**Question 12.** *Prouver que l'on peut transformer un programme Booléen en un programme équivalent (mais avec possiblement plus de variables) qui ne contient pas d'instruction du type "$if\ then\ else$". Montrer que cette transformation se calcule en temps polynomial.*

*Solution :* Le programme Booléen $P$ peut contenir plusieurs instructions $if\ then\ else$ : numérotons les de 1 à $k$.

Nous introduisons une variable $S_j$ pour $1 \leq j \leq k$, et une variable $Go$. Nous fixons la valeur initiale de $Go$ à 1 avec une toute première instruction $Go := 1$. Nous ajoutons ensuite à cette instruction des instructions obtenues de la façon suivante :

Nous remplaçons alors chaque instruction $I$ de $P$ par l'instruction $\overline{I}$ obtenue de façon inductive de la façon suivante :

— si $I$ est de la forme $X := E$ pour une certaine expression $E$, nous prenons $\overline{I}$ comme

$$X := (E \wedge Go) \vee (X \wedge \neg Go)$$

La motivation est que l'effet d'une telle instruction $\overline{I}$ lorsque $Go$ vaut 0 est de ne rien faire (enfin a essentiellement pour effet de faire $X := X$), et lorsque $Go$ vaut 1 d'avoir le même effet que $X := E$, à savoir celui de $I$.

— si $I$ est de la forme $J; K$, nous prenons $\overline{I}$ comme $\overline{J}; \overline{K}$

Cela permet de préserver cette propriété inductivement : lorsque $Go$ vaut 0, l'effet et de ne rien faire, et lorsque $Go$ vaut 1, cela a le même effet que $I$.

— si $I$ est de la forme $if\ U\ then\ J\ else\ K$ pour l'instruction $if\ then\ else$ de numéro $j$, nous prenons $\overline{I}$ comme

$$S_j := Go; Go := S_j \wedge U; \overline{J}; Go := S_j \wedge \neg U; \overline{K} \wedge Go := S_j.$$

Cela permet de préserver encore cette propriété inductivement : lorsque $Go$ vaut 0, l'effet et de ne rien faire, et lorsque $Go$ vaut 1, cela a le même effet que $I$.

En effet, dans le cas où $Go$ vaut 0, $Go$ restera 0 à l'issue de $Go := S_i \wedge U$ et $Go := S_i \wedge \neg U$, et donc l'effet d'exécuter $\overline{J}$ et $\overline{K}$ est de ne rien faire.

Dans le cas où $Go$ vaut 1, le programme va exécuter successivement $\overline{J}$ puis $\overline{K}$. Mais cela sera fait une fois avec $Go$ à 1 et l'autre fois avec $Go$ à 0 ou le contraire, selon la valeur de $U$. Par la propriété inductive, quand $U$ est 1, cela a l'effet de $J$, soit exactement le même effet que $J$ (l'effet de $\overline{K}$ étant nul), et quand $U$ est 0, cela a exactement (aucun effet pour $\overline{J}$ puis) l'effet de $\overline{I}$, soit exactement l'effet de $I$. Bref, le programme a exactement au final, le même effet que s'il faisait $I$.

Observons que la première et dernière instruction $S_i := Go$, et $Go := S_i$ permet de sauvegarder la valeur de la variable $Go$, pour ne pas perturber les instructions suivantes. □

**Question 13.** *Prouver que l'on peut transformer un programme Booléen sans instruction du type "if then else" en un programme équivalent (mais avec possiblement plus de variables) qui ne contient pas d'instruction du type "if then else" et où chaque variable apparaît au plus une fois*[6] *à gauche du signe :=. Montrer que cette transformation se calcule en temps polynomial.*

*Solution :* Un programme Booléen sans *if then else* est essentiellement une suite d'affectations : techniquement, il peut contenir des instructions composées de la forme $I; I'$ mais on peut inductivement les remplacer par faire $I$, puis faire $I'$ de façon à ce qu'il ne contiennent effectivement que des affectations. Cela ne change pas la sémantique (l'effet) du programme Booléen.

Le principe est de remplacer alors chaque affectation $X_i := E_i$, pour $1 \leq i \leq m$ par $X^i := E^i$ où $X^1, \ldots, X^k$ sont des nouvelles variables, et $E^i$ est identique à $E_i$ si ce n'est que toute référence à chaque variable $Y$ dans $E_i$ est remplacée comme suit :

— on recherche en arrière dans les instructions précédentes la première fois (donc la dernière fois en terme de l'exécution) où on a une instruction $I_\ell$ de la forme $Y := \ldots$, ou éventuellement on détecte qu'il n'y en a pas.

— si on en a trouvée une, en remplace $Y$ par $X^\ell$, sinon on laisse $Y$ inchangée.

Tout cela ne change rien à l'effet du programme (à sa sémantique), si ce n'est qu'un renommage de variables. Et nous avons bien garantit que chaque variable est affectée doérnavant au plus une fois (la variable $X^i$ ne peut être affectée que à la ligne numéro $i$).

La transformation du programme $P$ en le programme $P'$ décrite ci-dessus est bien une transformation qui se fait en temps polynomial (en la taille du programme $P$ : en effet, cela reste

6. Autrement dit, on a jamais $X := \ldots$ quelque part, et $X := \ldots$ autre part dans le programme, et ce pour toute variable $X$.

polynomial en son nombre d'instructions et en le nombre de ses variables, qui reste polynomial en sa taille). □

**Question 14.** *En déduire*[7] *la NP-complétude de* $\leq$ 3-SAT

— ***Donnée****: Une formule propositionnelle $\varphi$ en forme normale conjonctive avec au plus 3 littéraux par clause*

— ***Réponse****: Déterminer si $\varphi$ est satisfiable.*

*Ainsi que de :* 3-SAT

— ***Donnée****: Une formule propositionnelle $\varphi$ en forme normale conjonctive avec exactement 3 littéraux par clause*

— ***Réponse****: Déterminer si $\varphi$ est satisfiable.*

*Solution :*

Les problème $\leq$ 3-SAT et 3-SAT sont clairement dans NP, car la donnée des valeurs de vérité des variables constitue un certificat de taille polynomiale vérifiable en temps polynomial.

Il reste à montrer qu'on peut réduire SAT-PROGRAMME-BOOL à $\leq$ 3-SAT. Considérons un programme Booléen $P$. On peut le transformer en temps polynomial en un programme Booléen sans aucun *if then else* , et avec affectations uniques, dont les variables sont $X_1, X_2, \ldots, X_v$ par les questions précédentes.

Le programme obtenu s'écrit avec les instructions $I_1; I_2; \ldots; I_m$, nous pouvons considérer la formule $\varphi = Init \wedge \underline{I_1} \wedge \underline{I_2} \wedge \ldots \underline{I_m} \wedge X_1$ avec $\underline{I}$ donnée par le tableau de correspondance suivant :

| instruction programme Booléen | Formule |
|---|---|
| $X := true$ | $X$ |
| $X := false$ | $\neg X$ |
| $X := Y$ | $(X \vee \neg Y) \wedge (\neg X \vee Y)$ |
| $X := \neg Y$ | $(X \vee Y) \wedge (\neg X \vee \neg Y)$ |
| $X := Y \vee Z$ | $(X \vee \neg Y) \wedge (X \vee \neg Z) \wedge (\neg X \vee Y \vee Z)$ |
| $X := Y \wedge Z$ | $(X \vee \neg Y \vee \neg Z) \wedge (\neg X \vee \neg Y) \wedge (\neg X \vee \neg Z)$ |

Chaque ligne de ce tableau est construite pour avoir la propriété suivante : si on note $I$ l'instruction programme Booléen et $F$ la formule correspondante, alors effectuer l'instruction $I$ va rendre la formule $F$ vraie. Et par ailleurs, si la formule $F$ est satisfaite, alors c'est que la valeur de $X$ est nécessairement celle que l'on obtient si l'on exécute l'instruction $I$.

La formule $\varphi$ possède les mêmes variables que le programme $P$, à savoir $X_1, \ldots, X_v$.

La transformation du programme initial en cette formule est bien calculable en temps polynomial.

Par construction, on peut affecter les variables $X_1, \ldots, X_v$ telles qu'à la fin de l'exécution du programme la variable $X_1$ vaut 1 si et seulement si l'on peut satisfaire la formule $\varphi$.

En effet, s'il existe une telle affectation, en prenant comme valeur pour chaque variable de $\varphi$ la valeur finale de cette variable à la fin de l'exécution du programme, par construction, nous aurons satisfait chacune des sous-formules $\underline{I_1}$, $\ldots \underline{I_m}$, et $X_1$, et donc nous savons que $\varphi$ est satisfiable.

Ce raisonnement est correct, car le programme est à affectations uniques, donc la valeur d'une variable qui n'est pas le membre gauche d'aucune affectation est celle qu'elle avait initialement, et d'une variable qui est le membre gauche d'une affectation le résultat d'une unique affectation, et la sous-formule correspondante est bien satisfaite : on peut par exemple faire une récurrence sur $n$, le nombre d'instructions effectuées, en raisonnant sur l'effet des $n$ premières instructions, pour $1 \leq n \leq m$, si l'on en doute.

Réciproquement, supposons que $\varphi$ est satisfiable, pour une certaine valeur de vérité de ses variables. Cela signifie que chacune de ses sous-formules $\underline{I_1}$, $\ldots \underline{I_m}$, et $X_1$ l'est. En raison de

7. En utilisant uniquement les questions 8 à 13, c'est-à-dire sans utiliser le théorème de Cook-Levin.

la remarque ci-dessus, à propos des propriétés des lignes du tableau, cela signifie pour chaque affectation $X := \ldots$ présente dans le programme, la valeur de vérité de $X$ est nécessairement celle qui lui serait donnée en exécutant le programme avec initialement ce choix de valeur de vérité.

Nous pouvons là aussi par exemple faire une récurrence sur la conjonction des formules $\underline{I_1} \wedge \cdots \wedge \underline{I_n}$ pour $1 \leq n \leq m$, si l'on en doute.

En particulier, c'est le cas pour la variable $X_1$. Donc, puisque $\varphi$ contient la sous-formule $X_1$, nous savons que $X_1$ est 1 à la fin de l'exécution.

Autrement dit, nous avons prouvé que SAT-PROGRAMME-BOOL$\leq$ $\leq$ 3-SAT.

Pour $3SAT$, il suffit de remplacer les lignes du tableau plus haut, par des formules avec 3 littéraux par clauses. Par exemple en dupliquant, un littéral dans une clause lorsqu'on a moins que 3 littéraux dans une clause du tableau ci-dessus. □

# 4 Axiomatisation et Axiomatisation finie

On fixe une signature.

On dit qu'une classe [8] de structures $K$ est axiomatisable si l'on peut trouver une théorie $T$ qui la caractérise : les structures de la classe sont exactement les modèles de $T$. On dit qu'une classe de structures $K$ est finimement axiomatisable s'il on peut trouver une théorie $T$ avec un nombre fini d'axiomes qui la caractérise.

*Remarque : En particulier, si $K$ est finiment axiomatisable, alors $K$ est modèle d'une unique formule : la conjonction des formules de $T$.*

**Question 15.** *Supposons que $T$ axiomatise $K$. On suppose que $K$ est finiment axiomatisable (possiblement par une autre théorie finie $T'$). Montrer que l'on peut trouver un nombre fini d'axiomes de $T$ qui axiomatise aussi $K$.*
Indication : on pourra utiliser le théorème de compacité.

*Solution :* Supposons que $K$ corresponde aux modèles de $T'$ pour $T'$ fini. Alors $K$ correpond aussi à la conjonction $\sigma$ des formules de $T'$. Cette formule $\sigma$ se prouve à partir de $T$. Par le théorème de compacité, elle se prouve à partir d'un nombre fini d'axiomes de $T$, que l'on peut appeler $\psi_1, \ldots, \psi_k$. Alors $K$ s'axiomatise aussi par $\psi_1, \ldots, \psi_k$ :

— un modèle de $T$ satisfait en particulier $\psi_1, \ldots, \psi_k$, et donc tout élément de $K$ fait partie des modèles de $\psi_1, \ldots, \psi_k$.

— chaque modèle de $\psi_1, \ldots, \psi_k$ est un modèle de $\sigma$, puisque $\sigma$ se prouve à partir d'eux. Donc fait partie de $K$.

□

**Question 16.** *Montrer que si $K$ est finiment axiomatisable alors $K$ et son complémentaire sont axiomatisables.*

*Solution :* Si on est finiment axiomatisable, alors on est axiomatisable. $K$ est finiment axiomatisable, donc axiomatisé par la conjonction des formules de cette axiomatisation finie. Son complémentaire est donc axiomatisé par la négation de cette formule. □

**Question 17.** *Montrer que la réciproque est vraie : si $K$ et son complémentaire sont axiomatisables alors $K$ est finiment axiomatisable.*
Indication : on pourra utiliser le théorème de compacité, et utiliser le fait que l'union des deux théories doit être contradictoire.

8. On peut prendre le mot "classe" dans cet énoncé comme un synonyme de "ensemble".

*Solution :* L'union de la théorie $T_1$ qui axiomatise $K$ et de la théorie $T_2$ qui axiomatise son complémentaire ne doit pas posséder de modèle. Donc il y a un sous-ensemble fini $S$ de cette union qui ne possède pas de modèle par le théorème de compacité. Si on prend les axiomes dans ce sous-ensemble fini, le sous-ensemble $S_1$ défini comme $S \cap T_1$ et $S_2$ défini comme $S \cap T_2$ donne une axiomatisation finie de $K$ et de son complémentaire. □

**Question 18.** *Nous avons vu dans le cours que la classe des corps de caractéristique 0 est axiomiatisable. Montrer que la classe des corps de caractéristique 0 n'est pas finiment axiomatisable.*

*Solution :* Nous avons vu que $\Delta \cup \{\overline{2} \neq 0, \overline{3} \neq 0, \ldots, \bar{p} \neq 0, \ldots\}$ axiomatise la classe des corps de caractéristique 0 pour un certain ensemble de formules $\Delta$. Suppons que $\mathcal{F}_0$ soit finiment axiomatisable. Alors par la question 15, cela le serait par $\Gamma = \Delta \cup \{\bar{p}_1 \neq 0, \ldots, \bar{p}_k \neq 0\}$, avec $p_1, \ldots, p_k$ des nombres premiers (pas nécessairement les $k$ premiers). On peut prendre $q$ un nombre premier plus grand que tous les $p_i$. Alors $\mathbb{Z}/q\mathbb{Z}$ est un modèle de $\Gamma$, mais n'est pas de caractéristique 0. On obtient une contradiction. □

**Question 19.** *Montrer que la classe des corps de caractéristique $\neq 0$ n'est pas finiment axiomatisable.*

*Solution :* Découle de la question 17. □

**Question 20.** *Nous avons vu dans le cours que la classe des corps algébriquement clos est axiomiatisable. Montrer que la classe des corps algébriquement clos n'est pas finiment axiomatisable.*

*Solution :* Soit $\sigma_n = \forall y_1 \ldots y_n \exists x \left(x^n + y_1 x^{n-1} + \ldots + y_{n-1}x + y_n = 0\right)$.

Alors $\Gamma = \Delta \cup \{\sigma_n \mid n \geq 1\}$ ($\Delta$ axiomatiserait la classe des corps algébriquement clos. On peut montrer la non-axiomatisation finie en utilisant la question 15, et en trouvant un corps dans lequel un certain polynôme ne se factorise pas. □