Chapitre 16 : Décidabilité et complexité

Table des matières

1 Décidabilité

1.1 Modèles de calcul et universalité

1.1.1 Introduction

L'objet de ce chapitre es l'étude de ce qu'il est possible de calculer avec un algorithme, avec ou sans contrainte de complexité temporelle.

Afin de pouvoir énoncer précisément des propriétés, il faut une définition formelle de la notion d'algorithme.

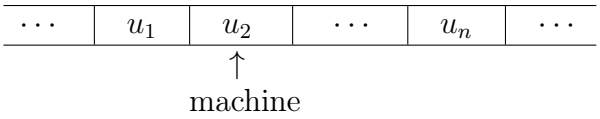
1.1.2 Modèles de calcul historiques (H.P)

• Nous avons vu dans le chapitre 14 plusieurs modèles de calcul, dont certains étaient équivalents (AFD, AFND, ε-AFND) et d'autres plus généraux (grammaires non contextuelle). On peut se demander s'il existe un modèle de calcul le plus général, capable de caractériser ce qu'il est possible de calculer mécaniquement.

• Plusieurs modèles de calcul ont étés proposés dans les années 1930, et se sont révélés équivalents et plus généraux que les modèles précédents : les machines de TURING, et le λ-calcul de CHURCH.

Les modèles les plus généraux conçus ultérieurement, comme les fonctions récursives, sont équivalents à ces deux modèles et on admet l'hypothèse, nommée thèse de CHURCH–TURING, que ces modèles caractérisent vraiment la notion d'algorithme. On dit aujourd'hui qu'un langage de programmation est *Turing-complet* s'il est capable d'écrire les mêmes algorithmes que ceux impémentables par machine de TURING.

• machine de TURING : informellement, une machine de TURING est une machine finie travaillant sur un ruban infini qui lui sert de mémoire. La machine dispose d'une tête de lecture lui permettant d'accéder à une case du ruban et de règles de transition décrivant les opérations réalisées sur les cases et les déplacements de la tête de lecture.



À la manière des automates, les machines de TURING ont un ensemble fini d'états, d'où la définition suivante :

> Une machine de TURING est un octuplet
>
> $$(\Sigma, \Gamma, B, Q, q_0, q_a, q_r, \delta)$$
>
> où
> – Σ est l'alphabet d'entrée
> .....
> – Γ est l'alphabet de ruban, ou de travail, tel que $\Sigma \subseteq \Gamma$;
> – $B \in \Gamma \setminus \Sigma$ est le symbole "blanc" représantant les cases vides ;
> – Q est un ensemble fini non vide d'états ;
> – $q_0 \in Q$ est l'état initial ;

– $q_a, q_r \in Q$ sont les états finaux de la machine : q_a est appelé l'état acceptant, q_r l'état rejetant ;
– δ est la fonction de transition :

$$\begin{array}{rccc} \delta & : & (Q \setminus \{q_a, q_r\}) \times \Gamma & \longrightarrow & Q \times \Gamma \times \{\leftarrow, \rightarrow\} \\ & & (q, a) & \longmapsto & (q', b, d) \end{array}$$

Si $\delta(q, a) = (q', b, d)$, alors lorsque la machine lit le symbole a sur le ruban en étant dans l'état q, elle écrit le symbole b à la place, déplace sa tête de lecture selon le déplacement d et passe dans l'état q'.

La machine accepte un mot $u \in \Sigma^*$ si et seulement si, en partant de l'état initial avec le ruban $B^\infty u B^\infty$ et la tête de lecture sur la première de u (si elle existe), l'exécution mène à l'état q_a. Elle rejette u si et seulement si elle atteint l'état q_r et elle peut également ne pas terminer.

Une machine de TURING peut aussi calculer une fonction : l'argument est placé sur le ruban et le contenu du ruban à la fin de l'exécution est le résultat de la fonction.

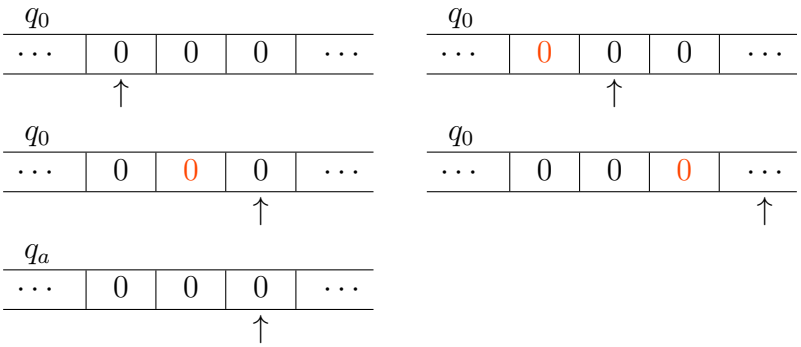
• Exemple :

$$M = (\{0,1\}, \{0,1,B\}, B, \{q_0, q_1, q_2, q_a, q_r\}, q_0, q_a, q_r, \delta)$$

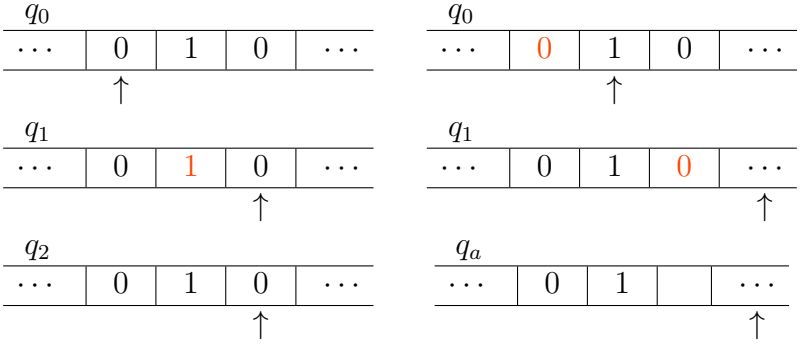
où δ est définie par la table :

	0	1	B
q_0	$(q_0, 0, \rightarrow)$	$(q_1, 1, \rightarrow)$	$(q_a, B, \leftarrow)$
q_1	$(q_1, 0, \rightarrow)$	$(q_1, 1, \rightarrow)$	$(q_2, B, \leftarrow)$
q_2	$(q_a, B, \rightarrow)$	$(q_a, B, \rightarrow)$	/

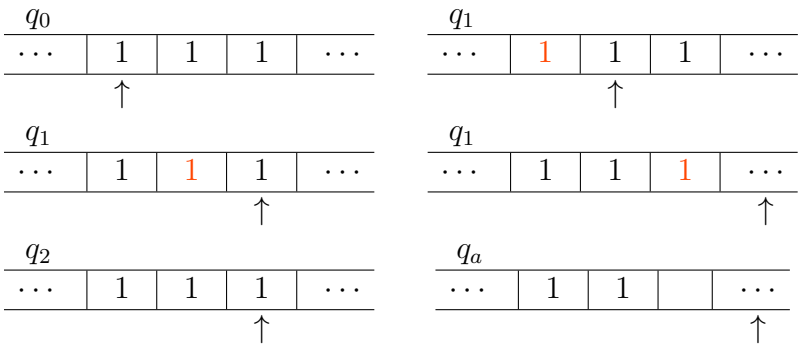
Avec $\langle 000 \rangle_2$ en entrée :



Avec $\langle 010 \rangle_2$ en entrée :



Avec $\langle 111 \rangle_2$ en entrée :



Cette machine de TURING calcule la fonction

$$\begin{array}{ccc} \{0,1\}^* & \longrightarrow & \{0,1\}^* \\ x & \longmapsto & y \end{array}$$

avec $\langle y \rangle_2 = \left\lfloor \dfrac{\langle x \rangle_2}{2} \right\rfloor$

On dit plus généralement que la machine calcule

$$\begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \longmapsto & \left\lfloor \dfrac{n}{2} \right\rfloor \end{array}$$

De même, si on supprime q_2 et on remplace δ par la table

	0	1	B
q_0	$(q_0, 0, \rightarrow)$	$(q_0, 1, \rightarrow)$	$(q_1, B, \leftarrow)$
q_1	$(q_a, 0, \rightarrow)$	$(q_r, 1, \rightarrow)$	/

On obtient une machine qui reconnaît le langage $\{u \in \{0,1\}^* \mid \langle u \rangle_2 \equiv 0 \ [2]\}$.

Autrement dit, la machine calcule $1/2\mathbb{N}$

• Les modèles du λ-calcul et des machines de TURING sont pertinents pour l'étude théorique de la calculabilité grâce à leur "simplicité", mais ne sont pas pratiques pour l'écriture d'algorithmes concrets. C'est pourquoi on utilise en général un modèle différent pour l'écriture des algorithmes.

1.1.3 Modèle de calcul au programme de la MPI

Le modèle de calcul considéré est celui d'un programme C ou OCaml qui s'exécute sur une machine à mémoire infinie.

En particulier, il n'y a jamais de dépassement de capacité de la pile d'exécution et on peut toujours allouer de la mémoire sur le tas.

Il est aisé de simuler une machine de TURING avec un programme C ou OCaml. Le sens réciproque est beaucoup plus difficile.

Nous appelons *algorithme* tout objet qui est un programme C ou OCaml, une machine de TURING, ou un λ-calcul.

On s'autorisera l'usage du pseudo-code pour l'écriture d'algorithmes.

1.1.4 Caclulabilité

Le terme *fonction* étant ambigu (fonction mathématique, fonctions dans un programme), on utilisera ce terme uniquement pour les fonctions mathématiques. Un algorithme peut être vu comme une réalisation d'une fonction mathématique partielle : celle qui à chaque entrée de l'algorithme sur la pile d'exécution se termine associe la valeur de retour de l'algorithme.

• Définition (*fonction calculable*) : une fonction $f : A \longrightarrow B$ est dite *calculable* s'il existe un algorithme M tel que $\forall a \in A$, l'exécution de M sur a termine en temps fini, et renvoie $f(a)$.

• Remarque :

− Il est "facile" de montrer qu'une fonction est calculable : il "suffit" d'exhiber un algorithme qui convient.

Il est en revanche beaucoup plus difficile de montrer qu'une fonction n'est pas calculable : il faut montrer qu'aucun algorithme ne peut convenir.

− On va se limiter aux fonctions de $\mathbb{N} \longrightarrow \mathbb{N}$. En effet :

+ si A est fini, on peut se contenter de tabuler les valeurs de f et d'écrire un algorithme qui va chercher dans la table la bonne valeur.

+ Si A est indémontrable, on a un problème de représentation de l'entrée : on a besoin de représentation infinies, ce qui est peu pertinent dans l'optique d'étudier ce qu'une machine réelle peut calculer.

+ On utilise en général des encodages pour représenter les données manipulées et un encodage binaire peut être vu comme un entier non signé (donc un entier naturel).

• Proposition :

> Il existe une infinité de fonctions non calculables.

□

Il suffit de montrer que $\mathbb{N}^{\mathbb{N}}$ est indénombrable car l'ensemble des algorithmes est dénombrable (l'ensemble des codes sources s'injecte dans l'ensemble des chaînes de caractères, dénombrable).

$\mathbb{N}^{\mathbb{N}}$ est indénombrable car $\{0,1\}^{\mathbb{N}}$ l'est déjà d'après le théorème de Cantor (on voit une fonction $\mathbb{N} \longrightarrow \{0,1\}$ comme la fonction indicatrice d'une partie de $\mathbb{N}$).

Argument diagonal (pour montrer que $\{0,1\}^{\mathbb{N}}$ n'est pas dénombrable) : on suppose que $\{0,1\}^{\mathbb{N}}$ est dénombrable. Alors on peut numéroter les suites de $\{0,1\}$:

$$\begin{array}{llllll} s_0 & 0 & 0 & 0 & \cdots \\ s_1 & 1 & 0 & 0 & \cdots \\ s_2 & 1 & 1 & 1 & \cdots \\ \vdots & & & & \end{array}$$

On a $(s_n)_{n\in\mathbb{N}} \in \left(\{0,1\}^{\mathbb{N}}\right)^{\mathbb{N}}$. Soit $\forall n \in \mathbb{N},\ u_n = 1 - s_n$. Alors :

$$(u_n)_{n\in\mathbb{N}} \in \{0,1\}^{\mathbb{N}}$$

Mais $\forall n \in \mathbb{N},\ s_n \neq (u_k)_{k\in\mathbb{N}}$: absurde. ■

1.1.5 Universalité

• On utilise souvent un argument diagonal pour montrer qu'une fonction n'est pas calculable : on procède par l'absurde en supposant l'existence d'un algorithme M convenable et en construisant un algorithme qui utilise M, souvent en faisant référence à son propre code source, pour aboutir à une absurdité (*cf* 1.2.4, page 9).

Ce type de démonstration nécessite deux propriétés essentielles :

– L'autoréférence, *i.e* la possibilité de faire référence à son propre code source. C'est possible car l'ensemble des algorithmes est dénombrable : on peut faire référence à un algorithme par son numéro.

– La simulation : il faut pouvoir simuler l'exécution d'un algorithme afin d'exploiter le résultat.

• Théorème :

> Il existe un algorithme, appelé machine universelle, d'entrée un algorithme M et une entrée x pour M, qui simule l'exécution de M sur x.

□

(démonstration informelle)

On passe par les machines de Turing.

Comme les machines à plusieurs rubans sont équivalentes aux machines à un ruban, on construit une machine qui a le code de M sur son ruban d'entrée, l'état courant de M dans l'exécution sur x, dans un deuxième ruban et le ruban de travail de M dans un troisième ruban. ■

1.2 Décidabilité

1.2.1 Introduction

On s'intéresse maintenant à des fonctions particulières : les *prédicats*, *i.e* les fonctions à valeur dans les booléens (ou $\{0,1\}$). Ces fonctions sont importantes car elles expriment des propriétés des éléments de l'ensemble qui constitue le domaine du prédicat : on veut pouvoir déterminer si un objet satisfait une propriété à l'aide d'un algorithme.

1.2.2 Problèmes de décision, d'optimisation

• Définition (*problème de décision*) : Un problème de décision sur un domaine A est défini par une fonction totale P de A dans l'ensemble des booléens.

Un élément $a \in A$ est appelé une *instance* du problème P et un algorithme M résout P si et seulement si $\forall a \in A,\ M$ appliqué à a termine et revoie $P(a)$.

• Remarques :

– On utilise rarement la définition d'un prédicat pour caractériser un problème de décision mais on préfère utiliser un énoncé en langue naturelle.

Exemple : SAT : « étant donné une formule propositionnelle A, A est-elle satisfiable ? » plutôt que

$$\begin{array}{rcccl} \text{SAT} & : & \text{Formules\_prop} & \longrightarrow & \{0,1\} \\ & & A & \longmapsto & \begin{cases} 0 & \text{si } \vDash \neg A \\ 1 & \text{sinon} \end{cases} \end{array}$$

Attention lors de l'expression d'un problème de décision, ne pas se limiter à une instance : par exemple, l'énoncé « la formule $X \vee \neg X$ est-elle satisfait ? » n'est pas un problème de décision car la réponse à cette question est « oui » ou « non », mais pas un algorithme, même s'il est possible d'utiliser un algorithme pour déterminer cette réponse.

Remarque : on pourrait reformuler cette question pour écrire un problème de décision (de domaine $\{X \vee \neg X\}$) mais comme dans le cas de la calculabilité, les problèmes de décision de domaine fini sont peu intéressants car on peut tabuler les réponses et écrire un algorithme allant chercher dans la table la réponse à l'instance considérée.

Attention, même si ces problèmes sont triviaux du point de vue de la décidabilité, ils peuvent être beaucoup plus complexes d'un point de vue algorithmique car il peut être en pratique impossible de tabuler les réponses.

Exemple : étant donné une position au jeu d'échecs, le joueur au trait est-il gagnant ?

– De nombreux problèmes de décision intéressants découlent de problèmes d'optimisation.

• Définition (*problème d'optimisation*) : Un problème d'optimisation sur un domaine d'entrées A et un domaine de solutions B est caractérisé par une relation $\mathcal{R} \subseteq A \times B$ qui lie les instances $a \in A$ aux solutions $b \in B$ possibles pour cette instance, et une fonction de coût $c : B \longrightarrow \mathbb{R}_+$.

Une *solution* à un problème d'optimisation est un algorithme M qui termine et renvoie une solution $b_{\min} \in B$ telle que

$$\begin{cases} a\mathcal{R}b_{\min} \\ c(b_{\min}) = \min\{c(b) \mid a\mathcal{R}b\} \end{cases}$$

• Exemple : étant donné un graphe $G = (S, A')$, trouver une coloration de G ayant un

nombre minimal de couleurs.

$$\begin{array}{rcl} A & = & \text{ensemble des graphes} \\ B & = & \text{ensemble des colorations} \\ G\mathcal{R}f & \Leftrightarrow & \begin{cases} \text{dom}(f) = S \\ \forall a = \{s, s'\} \in A',\ f(s) \neq f(s') \end{cases} \\ & \Leftrightarrow & f \text{ est une coloration valide de } G \\ c & = & \text{fonction qui à chaque coloration associe le nombre de couleurs utilisées} \end{array}$$

• On transforme un problème d'optimisation en problème de décision en introduisant un plafond sur les coûts $c_{\max}$ et en considérant le prédicat

$$\begin{array}{rcccl} P_{c_{\max}} & : & A & \longrightarrow & \{0,1\} \\ & & a & \longmapsto & \begin{cases} 1 & \text{si } \exists b \in B \,\Big|\, \begin{array}{l} a\mathcal{R}b \\ c(b) \leqslant c_{\max} \end{array} \\ 0 & \text{sinon} \end{cases} \end{array}$$

Exemple : un graphe G est-il 3-coloriable ? ou 4-coloriable ?

On peut également inclure le plafond dans le domaine du problème de décision :

$$\begin{array}{rcccl} P & : & A \times \mathbb{R}^+ & \longrightarrow & \{0,1\} \\ & & (a, c_{\max}) & \longmapsto & \begin{cases} 1 & \text{si } \exists b \in B \,\Big|\, \begin{array}{l} a\mathcal{R}b \\ c(b) \leqslant c_{\max} \end{array} \\ 0 & \text{sinon} \end{cases} \end{array}$$

Exemple : étant donné un graphe G et un entier k, G est-il k-coloriable ?

1.2.3 Problèmes décidables / indécidables

• Définition : un problème de décision caractérisé par un prédicat P est dit *décidable* si et seulement si il existe un algorithme qui le résout, ou de façon équivalente, si et seulement si P est calculable.

Dans le cas contraire, P est dit *indécidable*.

• Exemples :

– Une liste d'entiers est-elle triée ? (domaine : ensemble des listes d'entiers)

– Un entier est-il premier ? (domaine : $\mathbb{N}$)

– Un graphe est-il acyclique ? (domaine : ensemble des graphes)

– Un mot m est-il accepté par un AFD M ? (domaine : produit cartésien de Σ^* et de l'ensemble des automates finis déterministes sur Σ, pour Σ fixé).

• Proposition

> Il existe une infinité de problèmes indécidables

□

Cf 1.1.4, page 5. ■

1.2.4 Problème de l'arrêt

• Définition : le *problème de l'arrêt* est le problème de décision suivant : étant donné un algorithme M et une entrée e pour M, l'exécution de M sur e termine-t-elle ?

• Remarque : en pratique, le domaine de ce problème de décision est contraint à partir de l'ensemble des codes sources des algorithmes car il faut une représentation manipulable par un algorithme (comme en 1.1.4, page 5, on s'intéresse aux algorithmes d'entrées des chaînes de caractère ou des écritures binaires).

• Théorème

> Le problème de l'arrêt est indécidable.

□

Par l'absurde, avec un argument diagonal comme évoqué en 1.1.5, page 6.

On suppose qu'il existe un algorithme H qui résout le problème de l'arrêt.

On construit l'algorithme D d'entrées un algorithme M et de pseudo-code :

Algorithm 1: D

```
1 Simuler H sur l'entrée (M, M);
2 if résultat est vrai then
3     Boucler indéfiniment;
4 else
5     Terminer;
```

On observe alors l'exécution de D sur l'entrée D.

– Si H renvoie vrai pour l'entrée (D, D), c'est que D termine sur son propre code. Mais dans ce cas, D boucle indéfiniment : absurde.

– Si H revoie faux pour l'entrée (D, D), c'est que D ne termine pas sur son propre code.

Mais dans ce cas, D termine bien d'après la suite du code : absurde.

H réalisant une fonction totale, il termine toujours, donc on a traité tous les cas.
■

• Remarque : c'est une réécriture du paradoxe du barbier : un algorithme qui ne termine pas sur le code de tout algorithme qui termine sur son propre code termine-t-il sur son propre code ?

1.2.5 Problèmes semi-décidables (H.P)

• Définition : un problème de décision caractérisé par un prédicat P est dit semi-décidable si et seulement si il existe un algorithme M tel que $\forall a \in A$,

– Si $P(a)$, alors M termine sur a, et renvoie vrai ;

– Si $\neg P(a)$, alors soit M termine sur a et renvoie faux, soit M ne termine pas sur a.

• Proposition :

> Le problème de l'arrêt est semi-décidable.

□

Sur l'entrée (M, e), il suffit de simuler M sur e puis de renvoyer vrai.
Si M termine sur e, l'algorithme renvoie bien vrai, sinon il ne termine pas. ■

• Proposition

> Soit P un problème de décision.
> On appelle $\text{co}(P)$ le problème de décision associé au prédicat $\neg P$.
> Si P et $\text{co}(P)$ sont semi-décidables, alors P est décidable.

□

On se donne des algorithmes M_P et $M_{\text{co}(P)}$ tels que $\forall A \in \{P, \text{co}(P)\}$, M_A termine et revoie vrai sur toute instance a telle que $A(a)$ et ne termine pas ou renvoie faux sinon.

On construit l'algorithme suivant, qui résout P :

Algorithm 2:

Input: une instance a

```
1 Simuler en parallèle M_P et M_co(P) sur a;
2 if M_P termine then
3 |   Renvoyer son résultat;
4 if co(P) termine then
5 |   Renvoyer la négation de son résultat;
```

On sait que cet algorithme termine sur toute entrée a car soit $P(a)$, soit $\neg P(a)$ (tiers exclus). ■

• Corollaire :

> co Arrêt n'est pas semi-décidable.

Étant donné un algorithme M et une entrée e, l'exécution de M sur e est-elle finie ?

• Remarque : la réciproque de la proposition est vraie Exo.

• Exemple : un autre problème non semi-décidable : « étant donné un algorithme M, est-ce que M ne renvoie pas vrai sur son propre code source, *i.e* est-ce que M appliqué à son code renvoie faux ou ne termine pas ? »

Par un argument diagonal : si D termine et renvoie vrai sur tout M respectant la propriété et renvoie faux ou ne termine pas sur les autres.

Si l'exécution de D sur D :

– ne termine pas, alors par définition de la semi-décidabilité, D doit renvoyer vrai sur son code : absurde

– termine avec le résultat faux, de même : absurde
– termine avec le résultat vrai, alors D ne renvoie pas vrai sur son code : absurde

Remarque : le co-problème de ce problème de décision est semi-décidable et indécidable.
Il existe des problèmes non semi-décidables dont le co-problème n'est pas semi-décidable (cf TD_{45})