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 est 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.

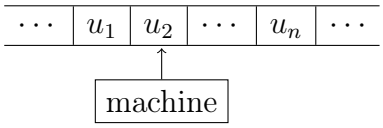
1.1.2 Modèles de calcul historiques (H.P)

• 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 implémentables 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.

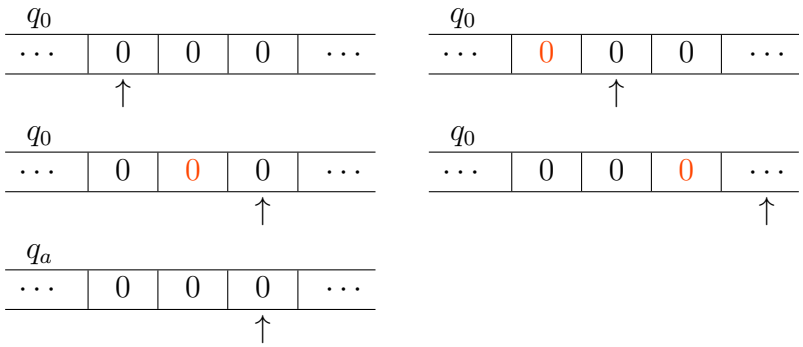
• Exemple :

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

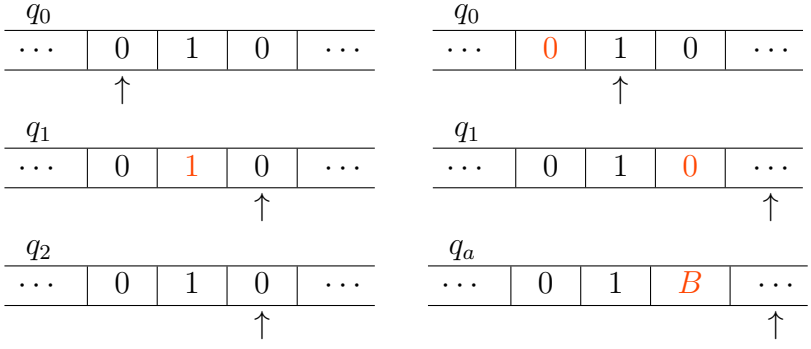
où δ est définie par la table :

	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	$(q_a, B, \rightarrow)$	$(q_a, B, \rightarrow)$	/

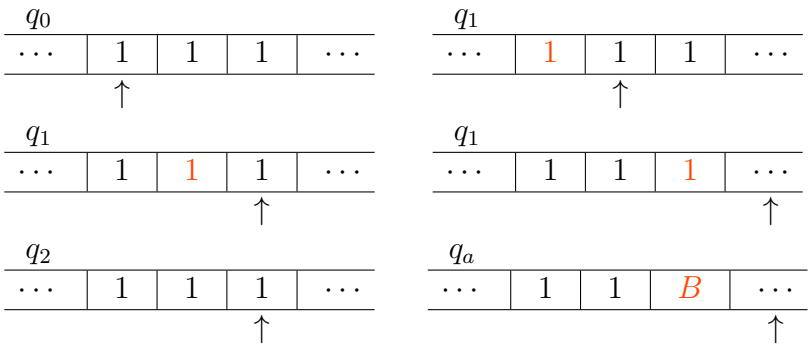
Avec $\langle 000 \rangle_2$ en entrée :



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 $2\mathbb{N}$.

• 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 $f(a)$.

• Remarques :

− 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 $\{0,1\}$:

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

On a $\forall n \in \mathbb{N},\ s_n = (s_{n,k})_{k\in\mathbb{N}} \in \{0,1\}^{\mathbb{N}}$, *i.e* $(s_{n,k})_{n,k\in\mathbb{N}} \in \left(\{0,1\}^{\mathbb{N}}\right)^{\mathbb{N}}$.

Soit $\forall n \in \mathbb{N},\ u_n = 1 - s_{n,n}$. 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 satisfiable ? » 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 co(P) le problème de décision associé au prédicat $\neg P$.
> Si P et 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}).

1.3 Réduction

1.3.1 Introduction

Pour montrer qu'un problème de décision P est décidable ou indécidable, une technique courante consiste à lier ce problème à un problème dont la décidabilité ou l'indécidabilité est déjà connue. En effet, si P' est décidable et si on peut ramener algorithmiquement la résolution de P à celle de P', alors P est décidable.
Inversement, si on peut ramener la résolution d'un problème P' indécidable à celle de P, alors P est indécidable.

1.3.2 Exemple

On considère le problème de la trivialité : étant donné un algorithme M à valeurs booléennes, M renvoie-t-il `true` sur toute entrée ?
On montre que ce problème est indécidable en le liant au problème de l'arrêt.
On suppose donc l'existence d'un algorithme M_T qui résout le problème de la trivialité et on construit l'algorithme suivant, d'entrées un algorithme M et une entrée e pour M :

Algorithm 3: Réduction de Arrêt à Trivialité

Input: M, e

1 Construire l'algorithme M' d'entrée e' et de code :

Algorithm 4: M'

Input: e'

2
1 Simuler M sur e;
2 Renvoyer `true`;

3 Simuler M_T sur M' et renvoyer son résultat;

Comme M_T réalise une fonction totale, M_T termine sur toute entrée, donc l'algorithme ci-dessus aussi.
On montre que cet algorithme résout le problème de l'arrêt : sur l'entrée (M, e), l'algorithme renvoie `true` $\Leftrightarrow$ si M_T renvoie `true` sur l'entrée M' $\Leftrightarrow$ M' renvoie `true` sur toute entrée, $\Leftrightarrow$ M termine sur e.

On a donc un algorithme qui résout le problème de l'arrêt, ce qui est absurde d'après 1.2.4 (page 9). Donc le problème de la trivialité est indécidable.

Remarque : on a écrit un algorithme qui transforme toute instance du problème de l'arrêt en une instance "équivalente" du problème de la trivialité. On dit donc que l'on a *réduit* le problème de l'arrêt au problème de la trivialité.

1.3.3 Réduction calculatoire

• Définition : soient deux prédicats $P_1 : E_1 \longrightarrow \texttt{bool}$ et $P_2 : E_2 \longrightarrow \texttt{bool}$.

On dit que le problème de décision associé à P_1 se *réduit calculatoirement* à celui associé à P_2 s'il existe une fonction $f : E_1 \longrightarrow E_2$ calculable telle que $\forall e \in E_1,\ P_1(e) \leftrightarrow P_2(f(e))$.

On note alors $P_1 \leqslant_m P_2$ et on dit que f est une *réduction* de P_1 à P_2.

• Remarques :

− On utiliser un symbole lié aux relations d'ordre car les réductions permettent "d'ordonner" les problèmes selon leur difficulté : si $P_1 \leqslant_m P_2$, alors P_2 est plus difficile à résoudre que P_1 car si l'on dispose des moyens nécessaires à la résolution de P_2, on sait également résoudre P_1 *via* la réduction.

− Le m du symbole $\leqslant_m$ vient du mot anglais de cette notion de réduction : on parle de *mapping reducibility*, ou de *many-to-one / many-one reducibility* (car f n'est pas forcément injective).

On parle aussi parfois de *computational reducibility*.

• Proposition :

> Soient P_1, P_2 deux problèmes de décision tels que $P_1 \leqslant_m P_2$.
> Alors
> (1) Si P_2 est décidable, alors P_1 aussi.
> (2) Si P_1 est indécidable, alors P_2 aussi.

□

(2) est la contraposée de (1), donc on démontre seulement (1).

Il existe un algorithme M_2 qui résout P_2.

Comme $P_1 \leqslant_m P_2$, il existe un algorithme M_f qui réalise une réduction de P_1 à P_2.

On construit l'algorithme M_1 d'entrée un instance e de P_1 :

Algorithm 5: M_1

1 Simuler M_f sur e et noter e' son résultat;
2 Simuler M_2 sur e' et renvoyer son résultat;

M_1 termine sur toute entrée car c'est le cas pour M_f et M_2.

Pour une instance e de P_1, M_1 renvoie vrai

$\Leftrightarrow$ M_2 renvoie vrai sur l'instance e' associée à e
$\Leftrightarrow$ M_2 renvoie vrai sur $f(e)$
$\Leftrightarrow$ $P_2(f(e))$ est vrai
$\Leftrightarrow$ $P_1(e)$ est vrai

Donc l'algorithme M_1 résout P_1. ■

• Attention à ne pas se tromper sur le sens de la réduction :

Exemple : on considère le problème de l'accessibilité dans un graphe : étant donné un graphe $G = (S, A)$ et $s, t \in S$, existe-t-il un chemin de s à t dans G ?

Ce problème est décidable : on lance un parcours de G depuis s et on revoie `true` si et seulement si t est atteint.

On montre que ACCESSIBILITÉ $\leqslant_m$ ARRÊT, ce qui n'apporte aucune information : le problème de l'arrêt est plus difficile que tout problème décidable puisqu'il est indécidable.

Voici un réduction, d'entrée G, s, t :

Algorithm 6: Réduction de ACCESSIBILITÉ à ARRÊT

Input: G, s, t

1 Définir l'algorithme $M_{G,s}$, d'entrée un sommet u :

Algorithm 7: $M_{G,s}$

Input: u

```
1 for chaque n ∈ ℕ do
2     for chaque séquence s_0 ··· s_n de sommets do
3         if s_0 ··· s_n est un chemin dans G de s à u then
4             return true;
```

5 **return** $(M_{G,s}, t)$;

On remarque qu'il existe un chemin de s à t dans G si et seulement si l'algorithme $M_{G,s}$ termine sur t.

On a donc bien construit une réduction calculatoire du problème de l'accessibilité au problème de l'arrêt.

1.3.4 Réduction TURING

• Remarque : la notion de réduction calculatoire est insuffisante pour capturer l'intuition de la notion de réduction. En effet, intuitivement, ARRÊT et co(ARRÊT) devraient être réductibles l'un à l'autre car si l'on sait résoudre l'un, on peut résoudre l'autre en inversant les réponses d'un algorithme de décision. Cependant, si on avait co(ARRÊT) $\leqslant_m$ ARRÊT, alors co(ARRÊT) serait semi-décidable Exo, ce qui est absurde d'après 1.2.5 (page 9).

Nous avons donc besoin d'une notion plus générale de réduction.

• Définition (*réduction* TURING) : Soient $P_1 : E_1 \longrightarrow \texttt{bool}$ et $P_2 : E_2 \longrightarrow \texttt{bool}$ deux problèmes de décision.

On dit que P_1 est TURING-*réductible* à P_2 si, en supposant l'existence d'un algorithme qui résout P_2, il existe un algorithme qui résout P_1.

On note alors $P_1 \leqslant_T P_2$, et on dit que P_1 est décidable relativement à P_2.

• Remarque : cette notion est liée aux machines de TURING à oracle (H.P), qui sont des machines pouvant effectuer des requêtes à un oracle, *i.e* à un outil qui peut "magiquement" résoudre le problème associé, même si ce problème est indécidable.

• Proposition :

> Soient P_1, P_2 deux problèmes de décision tels que $P_1 \leqslant_T P_2$.
> Alors :
> (1) Si P_2 est décidable, alors P_1 aussi.
> (2) Si P_1 est indécidable, alors P_2 aussi.

□

Comme en 1.3.3 (page 12), il suffit de montrer (1).

Il existe un algorithme M_2 qui résout P_2.

Par définition de $P_1 \leqslant_T P_2$, il existe donc un algorithme M_1 qui résout P_1.

En pratique, cet algorithme est construit ainsi : on utilise la machine à oracle qui résout P_1 à l'aide d'un oracle pour P_2 et on remplace cet oracle par l'algorithme M_2, ce qui donne un algorithme sans oracle qui résout P_1. ■

• Exemple : co(ARRÊT) $\leqslant_T$ ARRÊT.

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

On construit alors l'algorithme $M_{\neg H}$, d'entrée un algorithme M et une entrée e pour M :

Algorithm 8:

1 Simuler M_H sur (M, e) et noter b son résultat;
2 Renvoyer $\neg b$;

Cet algorithme termine sur toute entrée car c'est le cas pour M_H.

Sur l'entrée (M, e), $M_{\neg H}$ renvoie `true` $\Leftrightarrow$ M_H renvoie `false` sur (M, e) $\Leftrightarrow$ M ne termine pas sur e.

Donc $M_{\neg H}$ résout co(ARRÊT) et co(ARRÊT) $\leqslant_T$ ARRÊT.

• Remarque : cette réduction ne pose aucun problème du point de vue de la semi-décidabilité : une machine à oracle pour ARRÊT faisant appel à l'algorithme de semi-décision pour ARRÊT sur une instance sur laquelle il ne termine pas (donc une entrée (M, e) telle que M ne termine pas sur e), alors la machine ne peut s'arrêter pour renvoyer la réponse "vrai" pour co(ARRÊT), ce qui empêche d'avoir un algorithme de

semi-décision pour co(ARRÊT).

La différence entre réduction calculatoire et réduction TURING est la suivante : dans le cas d'une réduction calculatoire, in ne peut faire qu'un appel à l'oracle et on doit renvoyer directement son résultat alors que dans le cas d'une réduction TURING on peut effectuer des calculs supplémentaires qui dépendent de la réponse de l'oracle.

Autrement dit, si $P_1 \leqslant_m P_2$, alors $P_1 \leqslant_T P_2$, mais la réciproque est fausse.

2 Classes de complexité

2.1 Introduction

2.1.1 Motivation

Jusqu'ici, nous n'avons évoqué la que la complexité d'un algorithme donné. Un problème de décision donné peut être résolu par plusieurs algorithme : comment définir la complexité d'un problème de décision ?

Idée : on pourrait considérer la complexité du meilleur algorithme qui résout le problème. Pas clair : existe-il un algorithme de complexité meilleure que celle de tous les autres ?

On va donc inverser le point de vue : au lieu d'associer une complexité à un problème, on associe un ensemble de problème à chaque complexité. Un problème de décision appartient à une classe de complexité s'il peut être résolu avec un algorithme de la complexité associé à la classe.

2.1.2 Complexité

On rappelle que le modèle de calcul est celui d'un programme C ou OCaml s'exécutant sur une machine à mémoire infinie.

• Définition (*opération élémentaire*) : On appelle *valeur atomique* toute valeur qui s'écrit avec un nombre fini fixé à l'avance de bits (par exemple : un mot machine).

Les opérations élémentaires sont :

- Les opérations arithmétiques sur les valeurs atomiques ;
- Les comparaisons de valeurs atomiques ;
- Les lectures et écritures en mémoire de valeurs atomiques.

• Définition (*complexité d'un algorithme*) : Soit M un algorithme.

On appelle *complexité* (temporelle dans le pire cas) de M la fonction de $\mathbb{N} \longrightarrow \mathbb{N}$ qui à tout $n \in \mathbb{N}$ associe le maximum du nombre d'opérations élémentaires exécutées par M sur une entrée de taille n, où la taille d'une entrée est le nombre de valeurs atomiques nécessaires à sa description.

• Remarque : plutôt que d'exprimer la complexité d'un algorithme M, on a l'habitude d'en donner l'ordre de grandeur (symbole Θ) ou un majorant (symbole $\mathcal{O}$).

2.1.3 Classes de complexité

On se limite aux classes de complexité temporelles (les classes de complexité spatiales sont H.P).

• Définition : Soit $t : \mathbb{N} \longrightarrow \mathbb{N}$.

On note DTIME($t(n)$) l'ensemble des problèmes de décision résolubles avec un algorithme de complexité $\mathcal{O}(t(n))$.

• Remarque : le D de DTIME signifie *déterministe* : le modèle de calcul est celui des algorithmes déterministes : pas d'algorithmes randomisés / probabilistes (*cf* chapitre 17).

• Exemple : On considère le problème P_{sort} : étant donné deux tableaux de taille t_1 et t_2, t_2 est-il une version tirée de t_1 ?

En testant l'égalité entre t_2 et toutes les permutations de t_1, on montre que $P_{\text{sort}} \in \text{DTIME}(n! \cdot n)$.

En triant t_1 et en comparant avec t_2, on montre que $P_{\text{sort}} \in \text{DTIME}(n \log n)$.

• Proposition :

> Soient $f, g \in \mathbb{N}^{\mathbb{N}} \mid f(n) = \mathcal{O}(g(n))$.
> Alors
> $$\text{DTIME}(f(n)) \subseteq \text{DTIME}(g(n))$$

□

$f(n) = \mathcal{O}(g(n))$, donc

$$\exists C > 0,\ \exists N \in \mathbb{N} \mid \forall n \geqslant N,\ f(n) \leqslant Cg(n)$$

Soit $P \in \text{DTIME}(f(n))$.

Il existe M qui résout P avec une complexité $\mathcal{O}(f(n))$, *i.e* $\exists C' > 0,\ \exists N' \in \mathbb{N} \mid \forall n \geqslant N',\ \forall e$ entrée de taille n, l'exécution de M sur e effectue $\leqslant C'Cg(n)$ opérations élémentaires

Donc M est de complexité $\mathcal{O}(g(n))$.

Donc $P \in \text{DTIME}(g(n))$. ■

2.2 Les classes P et NP

2.2.1 La classe P

• Définition : la classe **P** est

$$\bigcup_{k \in \mathbb{N}} \text{DTIME}(n^k)$$

i.e la classe des problèmes de décision résoluble en temps polynomial.

• Exemples :

– Le problème de l'accessibilité (*cf* 1.3.3, page 12) appartient à **P** : on le résout à l'aide d'un parcours du graphe, donc en temps linéaire en la taille du graphe.

– Le meilleurs algorithme de tri par comparaison est de complexité $\mathcal{O}(n \log n) = \mathcal{O}(n^2)$, donc polynomiale, mais le problème « trier un tableau donné » n'est pas un problème de décision.

En revanche, le problème P_{sort} vu en 2.1.3 (page 16) appartient à **P**.

– Le problème suivant : « étant donné une grammaire algébrique sous forme normale de CHOMSKY G et un mot u, a-t-on $u \in \mathcal{L}(G)$? » appartient à **P** (*cf* TP_{22})

2.2.2 Vérification

• Exemple : on considère le problème CHEMIN_HAMILTONIEN : « étant donné un graphe G et deux sommets s et t, existe-il dans G un chemin hamiltonien de s à t ? »
(un chemin hamiltonien est un chemin passant exactement une fois par chaque sommet).

On peut résoudre ce problème en temps $\mathcal{O}(n!)$: il suffit de tester toutes les permutations de l'ensemble des sommets qui commencent par s et finissent par t.

On ne sait pas si CHEMIN_HAMILTONIEN $\in$ **P**, en revanche on peut facilement vérifier si une solution à ce problème est correcte : étant donné un chemin, on vérifie en temps linéaire si c'est un chemin hamiltonien de s à t. On dit que le chemin à vérifier est un certificat pour ce problème.

• Définition (*vérificateur*) : Soit P un problème de décision sur un domaine E.
Un *vérificateur* pour P est un algorithme V d'entrées une instance e $e \in E$ de P et une valeur c appelée *certificat* et à valeur dans les booléens tel que

$$\{e \in E |\ P(e)\} = \{e \in E \mid V \text{ termine et renvoie vrai sur l'entrée } (e, c)\}$$

• Exemple : pour le problème de coloration de graphe vu en 1.2.2, on écrit le vérificateur suivant, d'entrées une instance (G, k), où G est un graphe, et k un entier, et c une coloration :

Algorithm 9: Vérificateur du problème de coloration

Input: $(G, k), c$

```
Parcourir S en comptant le nombre de couleurs distinctes;
if ce nombre est > k then
    return false;
for tout {s, s'} ∈ A do
    if c(s) = c(s') then
        return false;
return true;
```

Ce vérificateur est de complexité $\mathcal{O}\left(|S|^2\right)$ (si on l'implémente naïvement).

• Remarque : en général, on exprime la complexité d'un vérificateur uniquement en fonction de la taille de l'instance.

2.2.3 La classe NP

• Définition : La classe **NP** est celle des problèmes de décision admettant un vérificateur de complexité polynomiale en la taille de l'instance.

• La contrainte de complexité justifie le fait d'ignorer la taille du certificat : un certificat trop grand ne pourra de toute façon pas être lu dans son intégralité par un algorithme de complexité polynomiale en la taille de l'instance.

On pourrait imposer au certificat d'être de taille polynomiale en celle de l'instance et considérer des vérificateurs de complexité polynomiale en la taille de leurs deux entrées.

Le N de **NP** signifie *non déterministe*, en raison du modèle H.P des machines de TURING non déterministes qui généralisent les machines de TURING déterministes de la même façon que les AFND généralisent les AFD. Un problème appartient à **NP** s'il existe une machine de TURING non déterministe qui le résout en temps polynomial.

• Théorème :

> $$\mathbf{P} \subset \mathbf{NP}$$

□

Soit D un problème de décision qui appartient à **P**.

Il existe donc un algorithme M qui résout D en temps polynomial.

On construit le vérificateur V d'entrée une instance e de D et un certificat que l'on ignore (chaîne de caractères quelconque), et dont le code est :

Algorithm 10: V

Input: e, c

```
1 Simuler M sur e;
2 return le résultat de M;
```

V est de complexité polynomiale en la taille de e car c'est le cas pour M.

$$\begin{array}{lcl} \exists c \mid V(e,c) \text{ renvoie } \texttt{true} & \Leftrightarrow & \forall c,\ V(e,c) \text{ renvoie } \texttt{true} \\ & \Leftrightarrow & M(e) \text{renvoie } \texttt{true} \\ & \Leftrightarrow & D(e) \end{array}$$

Donc V est un vérificateur pour D de complexité polynomiale, donc $D \in \mathbf{NP}$. ■

• Exemples :

– Les vérificateurs en 2.2.2 (page 17) montrent que le problème de coloration et CHEMIN_HAMILTONIEN appartiennent à **NP**.

– CLIQUE : étant donné un graphe $G = (S, A)$ et un entier $k \in \mathbb{N}$, existe-t-il une clique de taille k dans G, *i.e* un ensemble $S' \subseteq S$ de taille k telle que $G_{S'}$ est un graphe complet.

On montre que CLIQUE $\in$ **NP** grâce au vérificateur suivant :

Algorithm 11: Vérificateur de CLIQUE

Input: $G = (S, A), k, S'$

```
if |S'| ≠ k then
    return false;
for chaque s ∈ S' do
    for chaque s' ∈ S' \ {s} do
        if {s, s'} ∉ A then
            return false;
return true;
```

Ce vérificateur est de complexité $\mathcal{O}\big(|S'|^2\big) = \mathcal{O}\big(|S|^2\big)$ donc polynomiale en la taille de (G, k).

Attention, la taille de k est $\mathcal{O}(\log k)$.

Exo Montrer que le problème suivant appartient à **NP** :

SUBSET_SUM : étant donné un ensemble (multi-ensemble : les répétitions sont autorisées) fini et un entier S, existe-t-il

$$E' \subseteq E \mid \sum_{n \in E'} n = S$$

?

2.3 Réductions et complétude

2.3.1 Réduction polynomiale

• Définition : Soient P_1, P_2 des problèmes de décision.

On dit que P_1 se réduit en temps polynomial à P_2 si et seulement si il existe un réduction de P_1 à P_2 calculable en temps polynomial, *i.e* s'il existe une fonction f du domaine de P_1 vers celui de P_2 telle que $\forall e$ instance de P_1, $P_1(e) \Leftrightarrow P_2(f(e))$ et un algorithme M réalisant f et de complexité polynomiale.

On note alors $P_1 \leqslant_{\mathrm{p}} P_2$.

• Remarques :

– Si on sait résoudre P_2, alors il y a un surcoût polynomial pour résoudre P_1.

– On peut aussi généraliser aux réductions TURING la notion de réduction polynomiale.

• Exemple : on a vu que SAT $\leqslant_{\mathrm{p}}$ 3-SAT *via* la transformation de TSEITIN (*cf* chapitre 8, 3.3.5).

Rappel : Si φ est une formule propositionnelle, on ajoute des variables propositionnelles p_ψ, $\forall \psi \in SF(\varphi)$ et on construit des 3-FNC f_ψ qui représentent l'équivalence $\psi \leftrightarrow p_\psi$:

$$\begin{array}{rcl} f_x & = & (\neg x \vee p_x) \wedge (\neg p_x \vee x) \\ f_{\neg\psi} & = & (\neg p_{\neg\psi} \vee \neg p_\psi) \wedge (p_\psi \vee p_{\neg\psi}) \\ f_{\psi_1 \vee \psi_2} & = & (\neg p_{\psi_1 \vee \psi_2} \vee p_{\psi_1} \vee p_{\psi_2}) \wedge (\neg p_{\psi_1} \vee p_{\psi_1 \vee \psi_2}) \wedge (\neg p_{\psi_2} \vee p_{\psi_2 \vee p_{\psi_1 \vee \psi_2}}) \\ f_{\psi_1 \wedge \psi_2} & = & (\neg p_{\psi_1 \wedge \psi_2} \vee p_{\psi_1}) \wedge (\neg p_{\psi_1 \wedge \psi_2} \vee p_{\psi_2}) \wedge (\neg p_{\psi_1} \vee \neg p_{\psi_2} \vee p_{\psi_1 \wedge \psi_2}) \end{array}$$

On réécrit les sous-formules de la forme $\psi_1 \to \psi_2$ en $\neg\psi_1 \vee \psi_2$, ce qui ne fait que doubler la taille de la formule dans le pire cas.

On considère alors la 3-FNC

$$p_\varphi \wedge \bigwedge_{\psi \in SF(\varphi)} f_\psi$$

Cette formule est équisatisfiable à φ Exo, et calculable en temps linéaire en la taille de φ.

Donc SAT $\leqslant_{\mathrm{p}}$ 3-SAT.

2.3.2 NP-complétude

• Introduction : L'idée de la complétude est d'exploiter la notion de réduction afin de se focaliser sur des problèmes qui sont plus difficiles que tous les problèmes de leur classe de complexité.

En effet, si la complexité associée est super-polynomiale (polynomiale ou plus), alors le coût de la réduction est absorbé, donc on a un algorithme de même complexité pour chaque problème de la classe concernée, *i.e* la classe de complexité est stable par réduction polynomiale.

• Proposition : Soient P_1, P_2 des problèmes de décision tels que $P_1 \leqslant_{\mathrm{p}} P_2$.

Si $P_2 \in \mathbf{P}$ (resp. $P_2 \in \mathbf{NP}$), alors $P_1 \in \mathbf{P}$ (resp. $P_1 \in \mathbf{NP}$).

□

Il existe une réduction f de P_1 à P_2 et un algorithme M_f réalisant f et de complexité $\mathcal{O}(p_f(|e|))$ où p_f est un polynôme et e une instance de P_1.

– On suppose $P_2 \in \mathbf{P}$. Il existe donc un algorithme M_2 résolvant P_2, de complexité $\mathcal{O}(p_2(|e_2|))$, où p_2 est un polynôme, et e_2 une instance de P_2.

On construit l'algorithme M_1, d'entrée une instance e de P_1 et de code :

Algorithm 12: M_1

Input: e

1 Simuler M_f sur e pour obtenir $f(e)$;
2 Simuler M_2 sur $f(e)$ et renvoyer son résultat;

M_1 résout P_1 et est de complexité $\mathcal{O}(p_f(|e|) + p_2(|f(e)|))$.

Or $|f(e)| = \mathcal{O}(p_f(|e|))$, donc M_1 est de complexité $\mathcal{O}((\underbrace{p_f + p_2 \circ p_f}_{\text{polynôme}})(|e|))$.

Donc $P_1 \in \mathbf{P}$.

– On suppose $P_2 \in \mathbf{NP}$. Il existe donc un vérificateur V_2 pour P_2, de complexité polynomiale en la taille de l'instance de P_2 passé en entrée.

On montre de même que $P_1 \in \mathbf{NP}$ grâce au vérificateur V_1, d'entrées une instance e de P_1 et un certificat c (pour P_2), et de code :

Algorithm 13: V_1

Input: e, c

1 Simuler M_f sur e pour obtenir $f(e)$;
2 Simuler V_2 sur l'entrée $(f(e), c)$ et renvoyer son résultat;

■

• Définition (***NP**-complétude*) : Soit P_0 un problème de décision.

– On dit que P_0 est ***NP**-difficile* si et seulement si

$$\forall P_1 \in \mathbf{NP},\ P_1 \leqslant \mathrm{p} P_0$$

– On dit que P_0 est ***NP**-complet* si et seulement si

$$\begin{cases} P_0 \in \mathbf{NP} \\ P_0 \text{ est } \mathbf{NP}\text{-difficile} \end{cases}$$

• Proposition : Soit P_0 un problème **NP**-complet.

Si $P_0 \in \mathbf{P}$, alors $\mathbf{P} = \mathbf{NP}$.

□

On sait déjà que $\mathbf{P} \subseteq \mathbf{NP}$ (*i.e* 2.2.3, page 18).

Soit $P_1 \in \mathbf{NP}$.

Comme P_0 est **NP**-difficile, $P_1 \leqslant_\mathrm{p} P_0$, donc par la proposition précédente, comme $P_0 \in \mathbf{P},\ P_1 \in \mathbf{P}$, donc $\mathbf{NP} \subseteq \mathbf{P}$ et $\mathbf{P} = \mathbf{NP}$. ■

• Théorème (Cook-Levin) : SAT est **NP**-complet.

□

(démonstration informelle, H.P).

– SAT $\in \mathbf{NP}$: on considère comme certificat une valuation des variables de l'instance (donc de taille linéaire en la taille de l'instance) et on évalue la formule avec cette valuation (en temps linéaire) puis on vérifie que le résultat vaut V.

– SAT est **NP**-difficile : on considère $P_0 \in \mathbf{NP}$ et un vérificateur V pour P_0.

On veut montrer que $P_0 \leqslant_\mathrm{p}$ SAT. On voit une exécution de V comme une suite de configurations (plus facile avec les machines de Turing) et on construit une FNC exprimant que l'enchaînement des configurations est valide et dépendent de variables représentant la valeur du certificat. La formule sera satisfiable si et seulement si il existe un valeur du certificat menant à une exécution valide et acceptante de V.

On peut borner la taille du certificat et le nombre de configuration à enchaîner grâce à un polynôme qui borne la complexité de V, d'où une réduction polynômiale. ■

2.3.3 Preuves de NP-complétude

• Remarques : comme pour la décidabilité, l'usage de réductions (polynomiales) est un outil important.

• Prop : Soit P_0 un problème de décision.

Alors

$$P_0 \text{ est } \mathbf{NP}\text{-complet} \Leftrightarrow \begin{cases} P_0 \in \mathbf{NP} \\ \exists P_1 \ \mathbf{NP}\text{-complet} \mid P_1 \leqslant_{\mathrm{p}} P_0 \end{cases}$$

□

$\boxed{\Rightarrow}$ $P_0 \in \mathbf{NP}$ par définition.

Par le théorème de Cook-Levin, SAT est **NP**-complet, donc comme P_0 est **NP**-difficile, SAT $\leqslant_{\mathrm{p}} P_0$.

$\boxed{\Leftarrow}$ $P_0 \in \mathbf{NP}$ par hypothèse.

P_0 est **NP**-difficile : si $P_2 \in \mathbf{NP}$, alors comme P_1 est **NP**-complet, on a $P_2 \leqslant_{\mathrm{p}} P_1 \leqslant_{\mathrm{p}} P_0$.

Donc, comme $\leqslant_{\mathrm{p}}$ est transitive (car la composition de polynômes est un polynôme),

$$P_2 \leqslant_{\mathrm{p}} P_0$$

■

• Exemple : 3-SAT est **NP**-complet.

– 3-SAT $\in \mathbf{NP}$: on utilise le même vérificateur que celui vu pour SAT en 2.3.3 (page 22).

– SAT est **NP**-complet (*cf* 2.3.3, page 22) et SAT $\leqslant_{\mathrm{p}}$ 3-SAT (*cf* 2.3.1, page 19).

De même, $\forall k \geqslant 3$, k-SAT est **NP**-complet :

– k-SAT $\in \mathbf{NP}$ pour la même raison que 3-SAT ;

– 3-SAT est **NP**-complet et la fonction identité est une réduction polynomiale de 3-SAT à k-SAT car $k \geqslant 3$.

2.3.4 Clique

• Définition : Soit $G = (S, A)$ un GNO.

Une clique dans G est une ensemble $S' \subseteq S \mid \forall s \neq s' \in S', \ \{s, s'\} \in A$.

On dit que $|S'|$ est la taille de la clique.

On s'intéresse au problème Clique suivant : étant donné un GNO G et un entier k, existe-t-il une clique de taille k dans G ?

• Proposition :

> CLIQUE est **NP**-complet.

□

− CLIQUE $\in$ **NP** : on utilise S' come certificat : on vérifie en $\mathcal{O}(|S'|) = \mathcal{O}(|S|)$ que $S' \subseteq S$ et $|S'| = k$.

On vérifie en $\mathcal{O}\big(|S'|^2\big) = \mathcal{O}\big(|S|^2\big)$ que $\forall s \neq s' \in S',\ \{s, s'\} \in A$.

Donc on a bien un vérificateur de complexité $\mathcal{O}\big(|S|^2\big)$, donc polynomiale en la taille de (G, k).

− On montre que 3-SAT $\leqslant_{\mathrm{p}}$ CLIQUE :

Soit φ une instance de 3-SAT. On veut construire une instance (G, k) de CLIQUE en temps polynomial en $|\varphi|$ telle que G contient une clique de taille k si et seulement si φ est satisfiable.

$G = (S, A)$ est construit ainsi :

+ S contient chaque littéral de φ avec multiplicité ;

+ Deux littéraux l, l' sont liés dans A si et seulement si l et l' ne sont pas la négation l'un de l'autre et l, l' appartiennent à des clauses différentes.

k est le nombre de clauses de φ.

(G, k) est calculable en temps quadratique (donc polynomial) en $|\varphi|$.

Il reste à montrer que φ est satisfiable si et seulement si G contient une clique de taille k :

$\boxed{\Rightarrow}$ Il existe une valuation v telle que $[\![\varphi]\!]_v = V$, donc chaque clause de φ (numérotées de 1 à k), il existe un littéral l_i de la clause tel que $[\![l_i]\!]_v = V$.

On note alors $S' = \{l_i \mid i \in [\![1\ ;\ k]\!]\}$ et on montre que S' est une clique de taille k dans G.

$S' \subseteq S$, $|S'| = k$ et $\forall i \neq i \in [\![1\ ;\ k]\!]$, l_i et k_j appartiennent à des clauses différentes et l_i, l_j ne peuvent être négation l'un de l'autre car $[\![l_i]\!]_v = V = [\![l_j]\!]_v$.

Donc S' est une clique de taille k dans G.

$\boxed{\Leftarrow}$ Par définition de A, les littéraux de S' proviennent tous de clauses différentes donc on a exactement un littéral par clause.

Par définition de A, les littéraux de S' ne se contredisent pas, donc on peut construire une valuation qui rend vrai chaque littéral de S' et prolonger de manière quelconque cette valuation aux variables de φ qui n'apparaissent pas dans les littéraux de S'.

Par ce qui précède, cette valuation rend vrai un littéral de chaque clause de φ, donc c'est un modèle de φ. ■