3 NP-complétude

Bien que la question « $\mathsf{P} = \mathsf{NP}$? » résiste depuis plusieurs décennies, la quantité d'efforts investis a permis d'avoir une compréhension bien meilleure du problème. La notion de NP-complétude fait partie de ces avancées majeures, une des premières : elle a d'abord montré l'importance incroyable de la classe NP avant de donner un moyen aux algorithmiciens de contourner le problème « $\mathsf{P} = \mathsf{NP}$? ».

En effet, nous allons voir qu'un très grand nombre de problèmes naturels et importants en pratique sont tous équivalents et sont les plus difficiles de la classe NP : cela implique qu'ils ne possèdent pas d'algorithme polynomial si $\mathsf{P} \neq \mathsf{NP}$. Puisque la croyance populaire soutient que $\mathsf{P} \neq \mathsf{NP}$, on se contente de ces résultats comme d'une espèce de « preuve » que le problème est difficile et qu'on ne peut pas espérer obtenir un algorithme efficace [1].

Pour affirmer que certains problèmes « sont les plus difficiles », il faut pouvoir comparer les problèmes entre eux. On définit pour cela la notion de *réduction* qui tire ses origines de la calculabilité. Il existe beaucoup de réductions différentes mais nous ne définirons dans ce chapitre que l'un des deux types les plus importants. Puis nous verrons plusieurs problèmes NP-complets avant de voir le théorème 3-AK de Ladner et celui de Mahaney 3-AQ.

3.1 Réductions

Les notions de réduction permettent de comparer deux problèmes A et B. Grosso-modo, on dira que B est plus facile que A si on peut décider *efficacement* l'appartenance à B dès lors qu'on possède *un moyen* de tester l'appartenance à A : en effet, dans ce cas, résoudre

1. En réalité on a vu que la classe P ne reflétait qu'imparfaitement l'ensemble des problèmes « faciles ». En conséquence les algorithmiciens ne s'arrêtent pas à un résultat de NP-complétude : ils peuvent chercher des solutions approchées, des algorithmes qui fonctionneront sur « la plupart » des instances rencontrées, programmer des méthodes efficaces pour les petites entrées, etc.

A permet de résoudre B et donc B n'est pas plus difficile que A. Les différentes notions de réduction varient par ce qu'on entend par « efficacement » et selon le « moyen » de résoudre A.

En d'autres termes, pour résoudre B on « se ramène à » A : c'est en ce sens que B est plus simple que A. Plutôt que « se ramener à », on utilise le terme *se réduire à* : on dira que B se réduit à A, ou encore qu'il y a une réduction de B à A.

3-A Définition (réductions many-one polynomiales)

Une *réduction many-one* en temps polynomial d'un problème B (sur l'alphabet Σ_B) à un problème A (sur l'alphabet Σ_A) est une fonction $f : \Sigma_B^\star \to \Sigma_A^\star$ calculable en temps polynomial telle que :

$$\forall x \in \Sigma_B^\star, \quad x \in B \iff f(x) \in A.$$

Si une telle fonction f existe, on dira que B *se réduit à* A (via f) et on notera $B \leqslant_m^p A$.

3-B Remarques Quelques commentaires sur cette définition :

- La résolution de B se ramène à la résolution de A par le calcul de f : il suffit d'un précalcul simple (polynomial), c'est en ce sens que B est plus simple que A. La réduction transforme une instance x du problème B en une instance x' du problème A telles que $x \in B$ ssi $x' \in A$.
- On notera que la définition est équivalente à $B = f^{-1}(A)$.
- Le nom *many-one* vient de ce que plusieurs mots $x \in \Sigma_B^\star$ peuvent avoir la même image.
- Nous avons défini la réduction many-one en temps polynomial : on pourrait définir d'autres réductions many-one selon l'efficacité du calcul de f. Lorsqu'on parlera de réductions many-one sans préciser la complexité de f, on désignera les réductions many-one en temps polynomial.
- **Attention**, le terme « se réduire à » peut être trompeur puisque habituellement on se ramène à une tâche plus simple. Ici, lorsque B se réduit à A, c'est bien B qui est plus simple !

La plupart des classes de complexité abordées dans cet ouvrage seront closes pour les réduction many-one (c'est-à-dire que $A \in \mathscr{C}$ et $B \leqslant_m^p A$ impliquent $B \in \mathscr{C}$), notamment P, NP, EXP et NEXP. Une exception notable sont les classes E et NE et c'est la raison pour laquelle elles sont moins étudiées.

3-C Proposition

Les classes P et NP sont closes pour $\leqslant_m^p$.

Idée de la démonstration Si $B \leqslant_m^p A$ via f, pour décider B il suffit de calculer $f(x)$ et de décider si $f(x) \in A$.

Démonstration Pour la classe P : soit $A \in \mathsf{P}$ et $B \leqslant_m^p A$ via f, montrons que $B \in \mathsf{P}$. Soit k un entier suffisamment grand pour que f soit calculable en temps n^k et A soit décidable en temps n^k. Voici un algorithme pour B sur l'entrée $x \in \Sigma_B^\star$:

- Calculer $f(x)$.
- Décider si $f(x) \in A$.

Cet algorithme décide B par définition de la réduction f de B à A, et il fonctionne en temps polynomial :

- le calcul de f prend un temps $\leqslant n^k$;
- puisque $|f(x)| \leqslant n^k$, décider si $f(x) \in A$ prend un temps $\leqslant (n^k)^k = n^{k^2}$.

Le temps total est $O(n^{k^2})$, donc $B \in \mathsf{P}$.

Pour la classe NP, le raisonnement est exactement le même. Si A est reconnu par une machine non déterministe polynomiale N_A, la machine non déterministe pour B fonctionne ainsi sur l'entrée x :

- Calculer $f(x)$.
- Exécuter $N_A(f(x))$.

□

3-D Remarque La clôture de la classe P pour $\leqslant_m^p$ signifie qu'un problème B plus simple qu'un problème A efficacement résoluble est lui-même efficacement résoluble, conformément à l'intuition.

✎ **3-E Exercice**

Montrer que les classes EXP et NEXP sont closes pour $\leqslant_m^p$.

✎ **3-F Exercice**

Montrer que les classes E et NE ne sont pas closes pour $\leqslant_m^p$.

Indication : par théorème de hiérarchie, on sait que $\mathsf{EXP} \neq \mathsf{E}$. *Utiliser un problème* $L \in \mathsf{EXP} \setminus \mathsf{E}$, *le « padder » (cf. proposition 2-AU) pour obtenir un langage* $L' \in \mathsf{E}$ *: alors* $L \leqslant_m^p L'$.

Il est immédiat de constater que $\leqslant_m^p$ est une relation de pré-ordre sur l'ensemble des langages, c'est-à-dire qu'elle est réflexive et transitive.

3-G Lemme

La relation $\leqslant_m^p$ est réflexive et transitive.

Démonstration Réflexivité : soit A un langage, alors $A \leqslant_m^p A$ via l'identité.

Transitivité : soit A, B et C des langages tels que $A \leqslant_m^p B$ via f et $B \leqslant_m^p C$ via g. Alors $x \in A$ ssi $f(x) \in B$ ssi $g(f(x)) \in C$, donc $A \leqslant_m^p C$ via $g \circ f$. □

On en déduit la relation d'équivalence suivante, signifiant que deux langages ont même difficulté.

3-H Définition

Si deux langages A et B vérifient $A \leqslant_m^p B$ et $B \leqslant_m^p A$, on notera $A \equiv_m^p B$ et on dira que A et B sont équivalents pour les réductions many-one polynomiales.

Voici un exemple très simple de réductions.

3-I Exemple Soit `ENSEMBLE INDÉPENDANT` le problème suivant :

– *entrée* : un graphe non orienté G et un entier k ;
– *question* : existe-t-il un ensemble de k sommets indépendants, c'est-à-dire tous non reliés deux à deux ?

Ce problème semble très proche du problème `CLIQUE` présenté au chapitre précédent (exemple 2-AM). Montrons en effet que `ENSEMBLE INDÉPENDANT` $\equiv_m^p$ `CLIQUE`.

Pour un graphe $G = (V, E)$, on notera $\bar{G}$ le graphe $(V, {}^cE)$ sur les mêmes sommets mais dont les arêtes sont complémentaires à celles de G : x et y sont reliés dans $\bar{G}$ ssi ils ne sont pas reliés dans G (voir figure 3.1). On remarque que le calcul de $\bar{G}$ à partir de G se fait en temps polynomial.

Pour réduire `CLIQUE` à `ENSEMBLE INDÉPENDANT`, il s'agit de transformer une instance (G, k) de `CLIQUE` en une instance (G', k') de `ENSEMBLE INDÉPENDANT`. La réduction f que nous proposons est définie par $f(G, k) = (\bar{G}, k)$, c'est-à-dire qu'on prend le complémentaire de G et qu'on garde la valeur de k. Cette fonction f est bien calculable en temps polynomial ; pour qu'il s'agisse d'une réduction, il reste à montrer que $(G, k) \in$ `CLIQUE` ssi $f(G, k) = (\bar{G}, k) \in$ `ENSEMBLE INDÉPENDANT`. Dans un sens, si $(G, k) \in$ `CLIQUE` alors G possède une clique de taille k, c'est-à-dire k sommets tous reliés : dans $\bar{G}$, ces k sommets sont tous non reliés et forment un ensemble indépendant de taille k, donc $(\bar{G}, k) \in$ `ENSEMBLE INDÉPENDANT`. L'autre sens est similaire.

Il s'avère que, par le même raisonnement, f est également une réduction de `ENSEMBLE INDÉPENDANT` à `CLIQUE` (ce qui est une propriété assez exceptionnelle et ne reflète pas

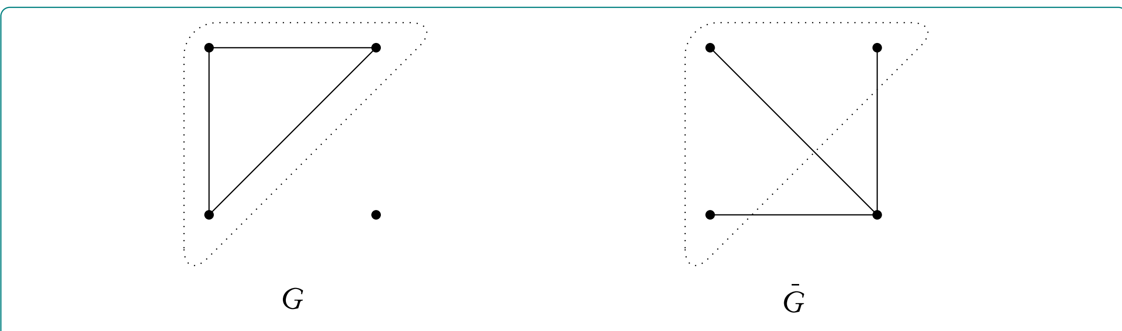

FIGURE 3.1 – Un graphe G avec une clique de taille 3 et son « complémentaire » $\bar{G}$ avec un ensemble indépendant de taille 3.

du tout le cas général des réductions entre problèmes). Donc les deux problèmes sont équivalents.

Il va sans dire que ce premier exemple de réductions est particulièrement simple : les réductions que nous verrons par la suite seront bien plus évoluées.

3.2 Complétude

Les réductions permettent de comparer les problèmes entre eux et donc de parler des problèmes *les plus difficiles* d'une classe de complexité. Nous introduisons pour cela les notions de difficulté et de complétude.

3.2.1 Définition et premières propriétés

3-J Définition (difficulté et complétude)

Soit $\leqslant$ une notion de réduction entre problèmes. Soit A un problème et $\mathscr{C}$ une classe de complexité.

– A est dit *$\mathscr{C}$-difficile* (ou $\mathscr{C}$-dur) pour les réductions $\leqslant$ si pour tout $B \in \mathscr{C}$, on a $B \leqslant A$.

– A est dit *$\mathscr{C}$-complet* pour les réductions $\leqslant$ s'il est $\mathscr{C}$-difficile pour les réductions $\leqslant$ et si $A \in \mathscr{C}$.

Cette notion dépend du type de réductions $\leqslant$; par défaut, si l'on ne précise pas le type de réductions, il s'agira des réductions many-one en temps polynomial $\leqslant_m^p$.

Ainsi, un problème $\mathscr{C}$-difficile est un problème au moins aussi dur que tous ceux de $\mathscr{C}$; on le qualifie en outre de $\mathscr{C}$-complet s'il appartient à $\mathscr{C}$ (il fait donc partie des problèmes

3-K Remarques

- La condition est très forte : il faut que *tous* les problèmes de $\mathscr{C}$ se réduisent à A. A priori, rien ne dit qu'il existe des problèmes $\mathscr{C}$-difficiles ni (a fortiori) $\mathscr{C}$-complets.
- Attention, si un langage A est hors de $\mathscr{C}$, cela n'implique pas qu'il est $\mathscr{C}$-difficile ! Nous verrons un contre-exemple à l'exercice 3-AO.

Pour certaines classes usuelles, on ne sait pas si elles contiennent des problèmes complets. Cependant, la magie de cette notion de complétude est l'existence de nombreux problèmes naturels complets pour les classes vues jusqu'à présent, notamment NP. Mais commençons par voir que cette notion est inutile pour la classe P.

3-L Proposition

Tout problème A non trivial (c'est-à-dire $A \neq \emptyset$ et $A \neq \Sigma_A^\star$) est P-difficile pour les réductions many-one en temps polynomial.

Démonstration Soit A non trivial, $x_0 \notin A$ et $x_1 \in A$. Soit $B \in \mathsf{P}$: montrons que $B \leqslant_m^p A$.
La réduction f de B à A est alors définie comme suit :

$$f(x) = \begin{cases} x_1 & \text{si } x \in B \\ x_0 & \text{sinon.} \end{cases}$$

Puisque $B \in \mathsf{P}$, cette fonction f est calculable en temps polynomial. Par ailleurs, si $x \in B$ alors $f(x) = x_1 \in A$ et si $x \notin B$ alors $f(x) = x_0 \notin A$: ainsi, $x \in B$ ssi $f(x) \in A$, donc f est une réduction de B à A. □

Un peu de recul

Ce résultat vient du fait que la réduction est trop puissante pour la classe P : pour avoir une pertinence, il faudra donc réduire la puissance des réductions. Nous verrons par la suite que nous pouvons considérer pour cela les réduction many-one en espace logarithmique.

Il s'agit d'un phénomène général : plus la classe est petite, plus il faut utiliser des réductions faibles pour comparer les problèmes de la classe.

Les machines universelles vues précédemment permettent de définir un premier problème complet pour les classes vues jusqu'à présent, ce que nous illustrons sur la classe NP.

3-M Proposition

Le problème

$$A = \{(\langle N \rangle, x, 1^t) \mid N(x) \text{ accepte en temps } \leqslant t\},$$

où N est une machine non déterministe, x un mot et t un entier (donné ici en unaire), est NP-complet.

Idée de la démonstration $A \in \mathsf{NP}$ car on peut simuler $N(x)$ grâce à la machine universelle. Pour réduire tout problème $B \in \mathsf{NP}$ à A, il suffit de simuler via A la machine non déterministe reconnaissant B.

Démonstration Montrons d'abord que $A \in \mathsf{NP}$. Soit U_{nondet} la machine non déterministe universelle de la proposition 2-AB : voici un algorithme non déterministe pour décider si $(\langle N \rangle, x, 1^t) \in A$.

- Simuler $N(x)$ pendant t étapes en exécutant $U_{\text{nondet}}(\langle N \rangle, x)$.
- Si le chemin simulé a accepté, `accepter`, sinon `rejeter` (le chemin simulé n'a pas terminé ou a rejeté).

Par la proposition 2-AB, cet algorithme fonctionne en temps non déterministe polynomial et possède un chemin acceptant ssi $N(x)$ en possède un en au plus t étapes. Ainsi, $A \in \mathsf{NP}$.

Montrons maintenant que A est NP-difficile. Soit $B \in \mathsf{NP}$: B est reconnu par une machine non déterministe N_B fonctionnant en temps polynomial $p(n)$. La réduction de B à A est alors la fonction f définie par $f(x) = (\langle N_B \rangle, x, 1^{p(|x|)})$: $x \in B$ ssi $N_B(x)$ accepte en temps $\leqslant p(|x|)$ ssi $(\langle N_B \rangle, x, 1^{p(|x|)}) \in A$ ssi $f(x) \in A$. Puisque f est clairement calculable en temps polynomial, on a $B \leqslant_m^p A$. □

3-N Remarque Ce n'est pas un problème NP-complet très intéressant car c'est un problème ad-hoc calqué sur la définition de NP. La NP-complétude devient pertinente lorsqu'elle concerne des problèmes *naturels*.

✎ **3-O Exercice**

Montrer que le problème

$$A = \{(\langle M \rangle, x, t) \mid M(x) \text{ accepte en temps } \leqslant t\},$$

où M est une machine déterministe, x un mot et t un entier donné en binaire, est EXP-complet. De même, si M est non déterministe, montrer que le problème correspondant est NEXP-complet.

Un des intérêts des problèmes NP-complets réside dans le résultat suivant.

3-P Proposition

Les affirmations suivantes sont équivalentes :

1. $\mathsf{P} = \mathsf{NP}$;

2. tout problème NP-complet est dans P ;
3. il existe un problème NP-complet dans P.

Démonstration L'implication 1 $\Longrightarrow$ 2 est évidente, ainsi que 2 $\Longrightarrow$ 3 car il existe au moins un problème NP-complet (cf. proposition 3-M). Il reste seulement à montrer 3 $\Longrightarrow$ 1. Soit B un langage de NP, montrons que $B \in \mathsf{P}$. Soit A le langage NP-complet donné par 3 : d'une part, $A \in \mathsf{P}$ par 3 ; d'autre part, $B \leqslant_m^p A$ par complétude de A. Puisque P est clos par réduction many-one (proposition 3-C), on en déduit que $B \in \mathsf{P}$. □

3.2.2 Complétude du problème SAT

L'avancée majeure dans la compréhension de la classe NP a été la découverte que de nombreux problèmes naturels réputés difficiles sont NP-complets. Cook [Coo71] et, indépendamment, Levin [Lev73] ont donné de tels problèmes au début des années 1970, complétés ensuite par Karp [Kar72] notamment. Ici notre premier problème sera SAT, la satisfaisabilité de formules booléennes.

3-Q Définition (formules booléennes)

L'ensemble des *formules booléennes sans quantificateur* sur les variables $x_1, \dots, x_n$ est défini par induction comme suit.

- Une variable x_i est une formule.
- Si $\varphi_1(x_1, \dots, x_n)$ et $\varphi_2(x_1, \dots, x_n)$ sont des formules, alors

$$(\varphi_1(x_1, \dots, x_n) \wedge \varphi_2(x_1, \dots, x_n)),$$
$$(\varphi_1(x_1, \dots, x_n) \vee \varphi_2(x_1, \dots, x_n)) \quad \text{et}$$
$$(\neg \varphi_1(x_1, \dots, x_n))$$

  sont des formules.

Si $a_1, \dots, a_n \in \{V, F\}$ (V pour vrai et F pour faux) sont des valeurs pour les variables $x_1, \dots, x_n$ (on dit aussi une *assignation*), alors la *valeur de vérité* d'une formule $\varphi(x_1, \dots, x_n)$ en $a_1, \dots, a_n$, notée $\varphi(a_1, \dots, a_n)$, est définie comme suit.

- Si $\varphi(x_1, \dots, x_n) = x_i$ alors $\varphi(a_1, \dots, a_n) = a_i$.
- Si $\varphi = (\varphi_1 \wedge \varphi_2)$ (conjonction, ET) alors

$$\varphi(a_1, \dots, a_n) = \begin{cases} V & \text{si } \varphi_1(a_1, \dots, a_n) = V \textbf{ et } \varphi_2(a_1, \dots, a_n) = V \\ F & \text{sinon.} \end{cases}$$

- Si $\varphi = (\varphi_1 \vee \varphi_2)$ (disjonction, OU) alors

$$\varphi(a_1, \dots, a_n) = \begin{cases} V & \text{si } \varphi_1(a_1, \dots, a_n) = V \textbf{ ou } \varphi_2(a_1, \dots, a_n) = V \\ F & \text{sinon.} \end{cases}$$

– Si $\varphi = (\neg\varphi_1)$ (négation, NON) alors

$$\varphi(a_1, \ldots, a_n) = \begin{cases} V & \text{si } \varphi_1(a_1, \ldots, a_n) = F \\ F & \text{sinon.} \end{cases}$$

3-R Remarque On utilisera la plupart du temps 0 au lieu de F et 1 au lieu de V. On se permettra également les conventions usuelles concernant le parenthésage, ainsi que des abréviations utiles comme

– 1 pour $x_1 \vee \neg x_1$ et 0 pour $x_1 \wedge \neg x_1$;
– $\varphi \to \psi$ pour $\neg\varphi \vee \psi$;
– $\varphi \leftrightarrow \psi$ pour $(\varphi \to \psi) \wedge (\psi \to \varphi)$.

Enfin, un uple de variables $(x_1, \ldots, x_n)$ sera souvent désigné par x si le nombre d'éléments $|x|$ du uple n'a pas besoin d'être précisé.

3-S Exemple $\varphi(x_1, x_2, x_3) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge (x_2 \vee x_3))$ est une formule booléenne sans quantificateur qui satisfait $\varphi(0,0,0) = 0$ et $\varphi(0,0,1) = 1$.

3-T Remarques

– On peut coder très simplement une formule booléenne en binaire en codant les symboles à la suite, séparés par un délimiteur. Puisqu'il faut donner le numéro des variables, une formule sur n variables s'écrivant avec k symboles aura un codage de taille $O(k \log n)$. La taille du codage d'une formule φ sera noté $|\varphi|$.

– Une formule se représente également sous la forme d'un arbre où les feuilles sont les variables (apparaissant éventuellement plusieurs fois) et les nœuds les opérations $\vee$, $\wedge$ et $\neg$.

– Étant données une formule $\varphi(x_1, \ldots, x_n)$ et une assignation $(a_1, \ldots, a_n)$ pour les variables, on peut décider en temps polynomial si $\varphi(a_1, \ldots, a_n) = 1$. Il suffit pour cela d'évaluer l'arbre correspondant à la formule depuis les feuilles jusqu'à la racine.

Nous pouvons maintenant introduire le problème SAT.

3-U Définition (SAT)

- Une formule $\varphi(x_1,\ldots,x_n)$ est dite *satisfaisable* s'il existe une assignation des variables $(a_1,\ldots,a_n)\in\{0,1\}^n$ telle que $\varphi(a_1,\ldots,a_n)=1$.
- Le problème SAT est le problème suivant :
  - *entrée* : une formule booléenne sans quantificateur $\varphi(x_1,\ldots,x_n)$;
  - *question* : φ est-elle satisfaisable ?

3-V Théorème (Cook 1971, Levin 1973)

Le problème SAT est NP-complet.

Plus précisément, le fonctionnement en temps polynomial d'une machine non déterministe N sur une entrée x est décrit par une formule φ calculable en temps polynomial telle que le nombre d'assignations satisfaisant φ est égal au nombre de chemins acceptants de $N(x)$.

Idée de la démonstration SAT $\in$ NP car il suffit de deviner une assignation des variables et vérifier en temps polynomial qu'elle satisfait la formule.

La complétude vient du fait qu'on peut décrire par une formule de taille polynomiale le diagramme espace-temps d'une exécution d'une machine non déterministe polynomiale car celui-ci répond à des règles *locales*. En d'autres termes, on décrit par une formule $\varphi(y)$ le fonctionnement de la machine le long du chemin (découlant du choix des transitions) décrit par y. Pour savoir s'il existe un chemin acceptant dans le calcul de la machine, il suffit alors de savoir s'il existe une affectation des variables y de la formule pour laquelle l'état final du diagramme décrit est acceptant, ce qui est un problème de type SAT.

Démonstration Voici un algorithme non déterministe fonctionnant en temps polynomial pour SAT sur l'entrée $\varphi(x_1,\ldots,x_n)$:

- deviner $(a_1,\ldots,a_n)\in\{0,1\}^n$;
- vérifier que $\varphi(a_1,\ldots,a_n)=1$.

Comme on l'a vu, la vérification prend un temps polynomial : ainsi, SAT $\in$ NP.

Il reste donc à montrer que SAT est NP-difficile. Soit $B\in$ NP : à toute instance x du problème B, il s'agit d'associer une formule φ_x telle que $x\in B$ ssi φ_x est satisfaisable. Les variables de φ_x désigneront en quelque sorte le chemin de calcul à suivre. Soit N une machine non déterministe polynomiale à k rubans pour B (ensemble d'états Q, alphabet de travail Γ) : nous allons « simuler » le fonctionnement de $N(x)$ le long d'un chemin arbitraire par φ_x. Pour cela, nous allons considérer le *diagramme espace-temps*

de $N(x)$, c'est-à-dire le diagramme représentant l'évolution du contenu des rubans au cours du temps. Par commodité, sur la figure 3.2 nous représentons un seul ruban de travail. La case sur laquelle se situe la tête de lecture est indiquée par le symbole $\star$.

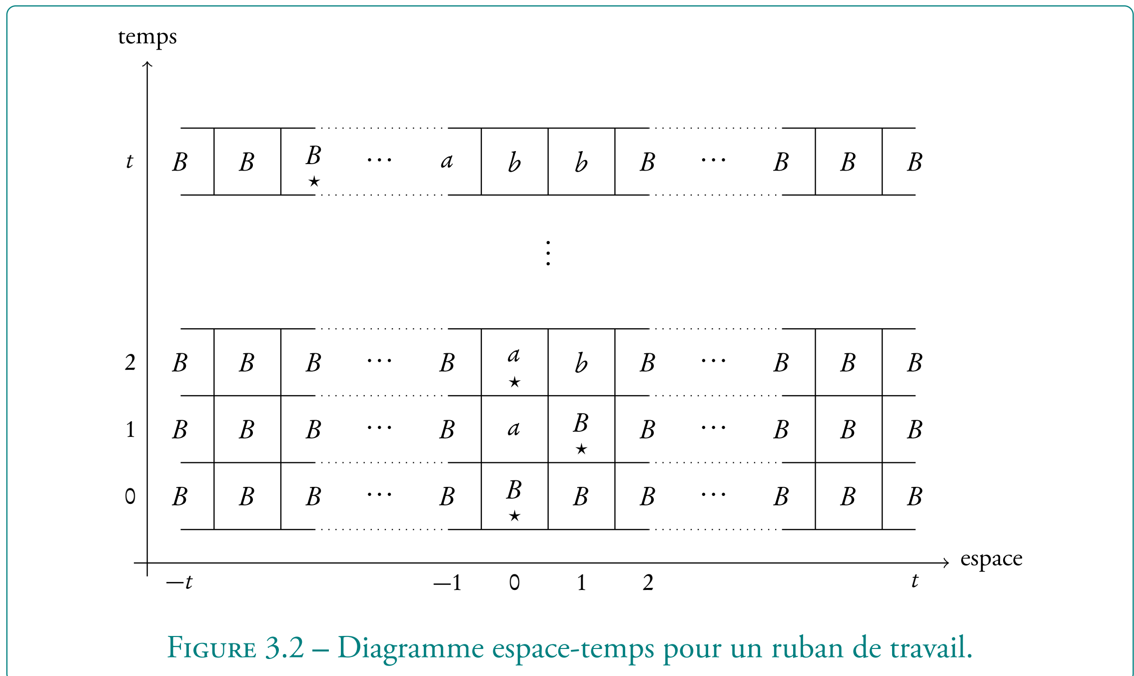

FIGURE 3.2 – Diagramme espace-temps pour un ruban de travail.

Soit $t(n)$ un polynôme majorant le temps d'exécution de N : si le calcul de $N(x)$ se termine en moins de $t(|x|)$ étapes, alors sur le diagramme espace-temps on suppose que N reste dans le même état jusqu'au temps $t(|x|)$.
Lors des t étapes de calcul de $N(x)$ (t est un polynôme en $|x|$, donc constructible en temps), les cases visitées du ruban se situent entre $-t$ et t. Nous considérons donc seulement les $(2t+1)(t+1)$ cases du diagramme représentées sur la figure.
Notre formule φ_x va affirmer que le calcul commence dans la configuration initiale, que le diagramme espace-temps de la machine est cohérent avec la relation de transition et qu'on termine dans un état acceptant. Elle sera composée de quatre parties :

– `cohérence` signifiant que deux valeurs ne sont pas assignées à la même case, deux positions à la même tête, deux états à la même étape ;
– $\texttt{début}_x$ signifiant que la configuration initiale est la bonne : on est dans l'état initial, les têtes sont en position initiale, x est écrit sur le ruban de lecture et les autres rubans sont vides ;
– pour chaque étape j, $\texttt{transition}_j$ signifiant que la j-ème transition est valide : déplacements des têtes et symboles écrits cohérents avec la relation de transition, pas de changement pour les cellules non concernées par les têtes ;
– `accepte` signifiant qu'on arrive dans un état acceptant à un temps $\leqslant t$.

Voici maintenant les variables qui seront utilisées dans notre formule.

– Pour chacune des étapes, chaque case des rubans va être décrite par plusieurs variables : pour chaque symbole $\gamma \in \Gamma$ et pour chaque ruban r, on utilise une variable $c^r_{\gamma,i,j}$ qui vaut *vrai* ssi la i-ème case du ruban r contient le symbole γ au temps j (ici, $1 \leqslant r \leqslant k$, $-t \leqslant i \leqslant t$ et $0 \leqslant j \leqslant t$).
– La position de la tête au temps j sur chaque ruban r est décrite par $2t+1$ variables : la variable $p^r_{i,j}$ (où $1 \leqslant r \leqslant k$, $-t \leqslant i \leqslant t$ et $0 \leqslant j \leqslant t$) vaut *vrai* ssi la tête du ruban r est à la position i au temps j.
– Pour chaque état q et à chaque instant j, la variable $e_{q,j}$ vaut *vrai* ssi l'état des têtes est q à l'instant j (ici, $q \in Q$ et $0 \leqslant j \leqslant t$).

En tout, le nombre de variables est donc $(t+1)(2t+1)|\Gamma|k+(t+1)(2t+1)k+(t+1)|Q|$, qui est polynomial en la taille de x puisque t est polynomial et Γ, k et Q sont constants (dépendant seulement de la machine N et non de x).
Dans un premier temps, il faut s'assurer qu'à chaque étape, chaque case contient exactement une valeur, chaque tête est exactement à un endroit et on est dans exactement un état. On peut exprimer cela par la conjonction des formules suivantes (ici, $r \in \{1,\dots,k\}$ désigne un ruban, $i \in \{-t,\dots,t\}$ une position, $j \in \{0,\dots,t\}$ une étape et $\gamma \in \Gamma$ un symbole) :

– à tout moment chaque case contient exactement un symbole :

$$\bigwedge_{r,i,j} \bigvee_{\gamma} (c^r_{\gamma,i,j} \wedge \bigwedge_{\gamma' \neq \gamma} \neg c^r_{\gamma',i,j});$$

– à tout moment chaque tête a exactement une position :

$$\bigwedge_{r,j} \bigvee_{i} (p^r_{i,j} \wedge \bigwedge_{i' \neq i} \neg p^r_{i',j});$$

– et à tout moment on est dans exactement un état :

$$\bigwedge_{j} \bigvee_{q} (e_{q,j} \wedge \bigwedge_{q' \neq q} \neg e_{q',j}).$$

La conjonction de ces trois formules est de taille polynomiale et donne la formule `cohérence` mentionnée ci-dessus.
Pour exprimer que la configuration initiale est correcte, il s'agit de tester si l'état au temps 0 est q_0, si tous les rubans sont vides à part le ruban de lecture qui contient $x = x_1 \dots x_n$ à partir de la case 1, et si toutes les têtes sont en position 1. On peut écrire cela sous la forme d'une conjonction des quatre formules suivantes :

– $e_{q_0,0}$ (l'état de départ est l'état initial q_0) ;
– $\bigwedge_{r>1,i} c^r_{B,i,0}$ (au départ toutes les cases des rubans $r > 1$ contiennent le symbole B) ;
– le ruban de lecture contient x entre les positions 1 et n et B ailleurs :

$$(\bigwedge_{i \leqslant 0 \vee i > n} c^1_{B,i,0}) \wedge (\bigwedge_{1 \leqslant i \leqslant n} c^1_{x_i,i,0});$$

– la tête de chaque ruban r est en position 1 au temps 0 :

$$\bigwedge_r (p^r_{1,0}).$$

La conjonction de ces quatre formules est de taille polynomiale et donne la formule $\texttt{début}_x$ mentionnée précédemment.

Pour alléger les notations qui suivent et par abus de notation, pour tout $\gamma \in \Gamma^{k-1}$ on notera $\delta(q_a,\gamma) = \{(q_a,(B,\ldots,B),(S,\ldots,S))\}$ et $\delta(q_r,\gamma) = \{(q_r,(B,\ldots,B),(S,\ldots,S))\}$ (alors qu'en réalité, par définition $\delta(q_a,\gamma) = \delta(q_r,\gamma) = \emptyset$ puisque le calcul s'arrête). Ce n'est qu'une notation pour la suite, mais intuitivement cela revient à considérer que le calcul ne s'arrête pas à partir du moment où on atteint un état terminal, et qu'alors la configuration ne change plus.

Dans ce cas, la formule `accepte` est simple à écrire, il s'agit d'exprimer qu'au temps t on est dans l'état acceptant q_a : cela s'écrit $e_{q_a,t}$.

Il reste le cœur de la simulation : spécifier que le comportement de la machine correspond à la relation de transition. Pour la j-ème transition (où $1 \leqslant j \leqslant t$), il s'agit de s'assurer que :

– pour chaque ruban les cases ne contenant pas la tête n'ont pas changé de contenu :

$$\psi_{\text{contenu}} \equiv \bigwedge_{r,i} \big(\neg p^r_{i,j-1} \to \bigwedge_{\gamma} (c^r_{\gamma,i,j} \leftrightarrow c^r_{\gamma,i,j-1})\big);$$

– et la valeur de la case contenant la tête, la position de la tête et l'état ont évolué selon une transition valide : pour tout état q, tout uple de positions i et tout uple de symboles γ, si à l'étape $j-1$, l'état est q, les positions des têtes sont i et les symboles lus sont γ, alors l'évolution doit se dérouler selon une transition $(q',\gamma',d') \in \delta(q,\gamma)$ (nouvel état, nouveau contenu des cases et déplacement des têtes). En symboles :

$$\psi_{\text{trans}} \equiv \bigwedge_{q,i,\gamma} \Big(\big(e_{q,j-1} \wedge \bigwedge_r (p^r_{i_r,j-1} \wedge c^r_{\gamma_r,i_r,j-1})\big) \to$$

$$\bigvee_{(q',\gamma',d') \in \delta(q,\gamma)} \big(e_{q',j} \wedge \bigwedge_r (c^r_{\gamma'_r,i_r,j} \wedge p^r_{i_r+d'_r,j})\big)\Big).$$

Ici, i désigne $(i_1,\ldots,i_k)$; γ désigne $(\gamma_1,\ldots,\gamma_k)$ mais, utilisé comme argument de δ, seules les $k-1$ premières composantes sont concernées (puisque le k-ème ruban est en écriture seule) ; de même, γ' désigne $(\gamma'_1,\ldots,\gamma'_k)$ mais, utilisé comme « image » de δ, seules les $k-1$ dernières composantes sont concernées (puisque le premier ruban est en lecture seule) ; d' désigne $(d'_1,\ldots,d'_k)$; enfin, $i_r + d'_r$ signifie $i_r - 1$ si $d'_r = G$, i_r si $d'_r = S$ et $i_r + 1$ si $d'_r = D$.

La formule $\texttt{transition}_j$ vaut alors $(\psi_{\text{contenu}} \wedge \psi_{\text{trans}})$. On remarquera une nouvelle fois que cette formule est de taille polynomiale.

On peut enfin écrire la formule φ_x :

$$\varphi_x = \texttt{cohérence} \wedge \texttt{début}_x \wedge \bigwedge_{1 \leqslant j \leqslant t} \texttt{transition}_j \wedge \texttt{accepte}.$$

La formule φ_x est de taille polynomiale et est satisfaisable si et seulement s'il existe un chemin acceptant dans le calcul de $N(x)$. Par ailleurs, le code de N étant fixé par le choix de la machine décidant le langage B, à partir de l'instance x il est aisé de construire en temps polynomial le code de φ_x puisque nous venons d'écrire cette formule explicitement. Ainsi la fonction $f : x \mapsto \varphi_x$ est une réduction many-one polynomiale de B à SAT.
Enfin, par construction (notamment grâce à la formule cohérence), il y a une bijection entre les assignations satisfaisant φ_x et les chemins acceptants de $N(x)$, donc le nombre de celles-ci est égal au nombre de chemins acceptants de $N(x)$. □

Ce problème est notre premier problème NP-complet naturel et servira de base pour montrer la complétude d'une multitude d'autres problèmes. En effet, pour montrer la NP-difficulté d'un problème A, il suffit maintenant de donner une réduction de SAT à A comme le montre le lemme suivant.

3-W Lemme

Soit C un problème NP-complet et $A \in \mathsf{NP}$. Si $C \leqslant_m^p A$ alors A est NP-complet.

Démonstration Soit $B \in \mathsf{NP}$. Par complétude de C, $B \leqslant_m^p C$. Or par hypothèse $C \leqslant_m^p A$: la transitivité de $\leqslant_m^p$ (lemme 3-G) permet de conclure que $B \leqslant_m^p A$ et donc que A est NP-difficile. □

3.2.3 Autres problèmes NP-complets

Notre prochain problème NP-complet est une variante importante de SAT appelée 3SAT : il s'agit de décider la satisfaisabilité de formules données en 3-CNF, forme que nous définissons maintenant.

3-X Définition (forme normale conjonctive)

- Un *littéral* est soit une variable (x_i) soit la négation d'une variable ($\neg x_i$).
- Une *clause* est une disjonction de littéraux : $C = \bigvee_i l_i$ où l_i est un littéral.
- Une formule φ est dite sous *forme normale conjonctive* (CNF) si elle est la conjonction de clauses (c'est-à-dire la conjonction de disjonctions de littéraux) :

$$\varphi = \bigwedge_i C_i \qquad \text{où} \qquad C_i = \bigvee_j l_{i,j} \text{ est une clause.}$$

– Enfin, φ est en 3-CNF si elle est en CNF avec au plus trois littéraux par clause.

3-Y **Exemple** $\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$ est en 3-CNF.

3SAT est alors le problème suivant :

– *entrée* : une formule $\varphi(x_1, \ldots, x_n)$ en 3-CNF ;
– *question* : φ est-elle satisfaisable ?

3-Z **Proposition**

Le problème 3SAT est NP-complet.

De plus, la réduction de SAT à 3SAT préserve le nombre d'assignations satisfaisant la formule.

Idée de la démonstration On voit une instance $\varphi(x)$ de SAT sous la forme d'un arbre : tester sa satisfaisabilité revient alors à deviner les valeurs de x et celles des nœuds de l'arbre et tester la cohérence de ces choix. Ce test est local à chaque nœud de l'arbre et nécessite seulement des clauses contenant trois littéraux car le degré d'un sommet de l'arbre est au plus 3 (un père et au plus deux fils).

Démonstration Que 3SAT soit dans NP est une évidence puisqu'il suffit de tester (en temps polynomial) si l'entrée est en 3-CNF et d'appliquer l'algorithme non déterministe pour SAT.

Afin de montrer sa NP-difficulté, nous donnons une réduction de SAT à 3SAT. Soit $\varphi(x)$ une instance de SAT : nous allons construire une instance $\psi(x, y)$ de 3SAT de sorte que φ soit satisfaisable ssi ψ l'est.

Pour cela, nous considérons l'arbre représentant la formule φ, dont les feuilles (entrées) sont les variables x_i (apparaissant éventuellement plusieurs fois) et les nœuds les opérations $\neg$, $\vee$ et $\wedge$: chaque nœud a donc au plus deux fils. On peut voir l'exemple d'un tel arbre à la figure 3.3.

L'objectif est de décrire la valeur de tous les nœuds de l'arbre afin de connaître la valeur de la racine. On associe donc une nouvelle variable y_s à chaque nœud s de l'arbre. Les clauses de notre formule $\psi(x, y)$ sont les suivantes :

– si s est une feuille correspondant à la variable x_i, il s'agit d'exprimer que $y_s = x_i$, ce qui donne les deux clauses $C_s = (x_i \vee \neg y_s) \wedge (\neg x_i \vee y_s)$;
– si s est une porte $\neg$ ayant pour fils la porte s', il s'agit d'exprimer que $y_s = \neg y_{s'}$, ce qui donne les deux clauses $C_s = (y_s \vee y_{s'}) \wedge (\neg y_s \vee \neg y_{s'})$;
– si s est une porte $\vee$ ayant pour fils les portes s_1 et s_2, on exprime que $y_s = y_{s_1} \vee y_{s_2}$, ce qui donne les trois clauses $C_s = (\neg y_s \vee y_{s_1} \vee y_{s_2}) \wedge (y_s \vee \neg y_{s_1}) \wedge (y_s \vee \neg y_{s_2})$;

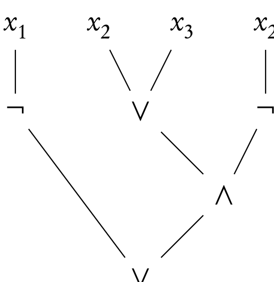

Figure 3.3 – Arbre correspondant à la formule $(\neg x_1) \vee ((x_2 \vee x_3) \wedge (\neg x_2))$.

– enfin, si s est une porte $\wedge$ ayant pour fils les portes s_1 et s_2, il s'agit d'exprimer que $y_s = y_{s_1} \wedge y_{s_2}$, ce qui donne les trois clauses

$$C_s = (y_s \vee \neg y_{s_1} \vee \neg y_{s_2}) \wedge (\neg y_s \vee y_{s_1}) \wedge (\neg y_s \vee y_{s_2}).$$

De cette façon, nous avons exprimé le fait que toutes les variables y_s prennent la valeur de la porte s correspondante. Il ne reste plus qu'à vérifier que la porte de sortie, appelons-la t, a la valeur 1, ce qui donne au final la formule

$$\psi(x,y) = y_t \wedge \bigwedge_s C_s,$$

qui est bien en 3-CNF. Ainsi, $\varphi(x)$ est satisfaisable ssi il existe une assignation des variables x telle que la porte t vaille 1, ssi pour cette assignation des variables x, il existe une (unique) assignation des variables y telle que $\psi(x,y)$ vaille 1. De plus, la fonction $\varphi \mapsto \psi$ est bien sûr calculable en temps polynomial puisqu'il suffit d'écrire les clauses que nous avons spécifiées ci-dessus. Au final, nous venons de décrire une réduction de `SAT` à `3SAT` qui conserve le nombre de solutions. □

3-AA **Remarque** Attention, le problème de la satisfaisabilité des formules en 2-CNF (deux littéraux par clause) est résoluble en temps polynomial, bien que l'algorithme ne soit pas trivial.

De même pour les formules en forme normale disjonctive (DNF), où les clauses sont des conjonctions de littéraux et la formule est une disjonction de clauses (cf. exercice ci-dessous).

✎ **3-AB** **Exercice**

Montrer que le problème de décider la satisfaisabilité d'une formule en DNF (forme normale disjonctive où les clauses sont des conjonctions de littéraux et la formule est une disjonction de clauses) est dans P.

À partir de cette variante de `SAT`, nous pouvons montrer la complétude de plusieurs problèmes dans des domaines variés : graphes, nombres, etc. Nous verrons seulement un petit nombre d'exemples mais la liste des problèmes `NP`-complets est grande. De nombreuses variantes de `SAT` sont `NP`-complètes et nous commençons par le problème `1Lit3SAT` suivant :

– *entrée* : une formule $\varphi(x_1, \ldots, x_n)$ en 3-CNF ;
– *question* : existe-t-il une affectation des variables rendant exactement un littéral vrai par clause ?

Il s'agit bien sûr d'un sous-ensemble de `3SAT` dont nous montrons qu'il est également `NP`-complet.

3-AC Proposition

Le problème `1Lit3SAT` est `NP`-complet.

Idée de la démonstration L'appartenance à `NP` est facile. La `NP`-difficulté vient d'une réduction de `3SAT` où l'on sépare les littéraux de chaque clause en ajoutant des variables pour garantir une solution avec exactement un littéral vrai par clause.

Démonstration Le problème est dans `NP` grâce à la machine non déterministe suivante, fonctionnant en temps polynomial sur une formule $\varphi(x_1, \ldots, x_n)$ en 3-CNF :

– `deviner` une affectation des variables ;
– vérifier que chaque clause contient exactement un littéral vrai.

Nous donnons maintenant une réduction du problème `3SAT` au problème `1Lit3SAT`. Soit $\varphi(x_1, \ldots, x_n)$ une instance de `3SAT` (c'est-à-dire une formule en 3-CNF) ayant m clauses.

Si C est une clause $(x \vee y \vee z)$ de φ, où x, y, z sont des littéraux (variables ou négation de variables), alors on remplace C par les quatre clauses

$$\psi_C = (\neg x \vee a \vee b) \wedge (\neg y \vee c \vee d) \wedge (\neg z \vee e \vee f) \wedge (a \vee c \vee e),$$

où $a, \ldots, f$ sont des nouvelles variables.

Si C est satisfaite par une affectation de x, y, z, alors il existe une affectation avec les mêmes valeurs de x, y, z rendant exactement un littéral vrai par clause de ψ_C. Pour montrer cela, raisonnons sur le nombre de littéraux vrais dans C :

– si un seul littéral de C est vrai, par exemple x (les autres cas sont symétriques), on fixe $a = 1$ et $b, c, d, e, f = 0$;
– si deux littéraux de C sont vrais, par exemple x et y (les autres cas sont symétriques), alors on fixe $a, d = 1$ et $b, c, e, f = 0$;
– si les trois littéraux de C sont vrais, on fixe $a, d, f = 1$ et $b, c, e = 0$.

Réciproquement, si ψ_C est satisfaite par une affectation rendant exactement un littéral vrai par clause, alors la même affectation de x, y, z satisfait C. En effet, exactement l'une des variables a, c, e doit être vraie, par exemple a (les autres cas sont symétriques), donc x doit être vrai sinon la première clause aurait deux littéraux vrais.

Le remplacement de chaque clause de φ de cette façon, en utilisant pour chacune de nouvelles variables $a, \ldots, f$, donne une formule $\psi = \bigwedge_C \psi_C$ contenant $4m$ clauses et $n+6m$ variables. La formule ψ est évidemment calculable en temps polynomial à partir de φ. Par ce qui précède, puisque les valeurs des littéraux x, y, z sont préservées (ce qui est important si plusieurs clauses utilisent les mêmes littéraux), si φ est satisfaisable alors il existe une affectation rendant exactement un littéral vrai par clause de ψ ; et réciproquement, si ψ possède une affectation rendant exactement un littéral vrai par clause, alors la même affectation des variables de φ satisfait φ. □

Nous proposons en exercice de montrer qu'une autre variante de `SAT` est `NP`-complète.

✎ 3-AD Exercice

Soit `Half-3SAT` le problème suivant :

– *entrée :* formule $\varphi(x_1, \ldots, x_n)$ en 3-CNF ;

– *question :* existe-t-il une affectation des variables $(x_1, \ldots, x_n)$ satisfaisant exactement la moitié des clauses ?

Montrer que `Half-3SAT` est `NP`-complet.

Indication : on peut utiliser plusieurs fois une même clause si nécessaire.

Les graphes sont aussi de grands pourvoyeurs de problèmes `NP`-complets. Par exemple, le problème `ENSEMBLE INDÉPENDANT` défini précédemment (exemple 3-I), pour lequel il s'agit de décider s'il existe k sommets non reliés deux à deux dans un graphe non orienté.

3-AE Proposition

Le problème `ENSEMBLE INDÉPENDANT` est `NP`-complet.

Idée de la démonstration Le problème est dans `NP` car vérifier que k sommets sont indépendants se fait en temps polynomial. Pour la `NP`-difficulté, on réduit `3SAT` : la réduction transforme une formule φ en 3-CNF à i clauses en un graphe contenant i triangles (un pour chaque clause). Chaque sommet d'un triangle correspond à un littéral de la clause. Entre les triangles, on relie ensuite x et $\neg x$. Si φ est satisfaisable, alors les littéraux valant 1 dans une solution forment un ensemble indépendant du graphe, et réciproquement.

Démonstration Voici une machine non déterministe pour `ENSEMBLE INDÉPENDANT`, sur l'entrée (G, k) :

- `deviner` un sous-ensemble de k sommets $x_1, \dots, x_k$;
- vérifier que $x_1, \dots, x_k$ sont distincts et sont tous deux à deux non reliés.

Il est clair que cette machine fonctionne en temps polynomial et qu'il existe un chemin acceptant si et seulement s'il existe un ensemble indépendant de taille k. Donc `ENSEMBLE INDÉPENDANT` est dans `NP`.

Pour montrer la `NP`-difficulté, nous réduisons `3SAT` à `ENSEMBLE INDÉPENDANT`. Soit $\varphi(x_1, \dots, x_n)$ une formule en 3-CNF (une instance de `3SAT`) : le but est d'associer à φ en temps polynomial un graphe G et un entier k tel que φ est satisfaisable ssi G a un ensemble indépendant de taille k. La formule φ est constituée de m clauses contenant chacune trois littéraux (si une clause contient moins de trois littéraux, on ajoute l'un des littéraux pour en obtenir exactement trois, par exemple $(x_1 \vee \neg x_2) \mapsto (x_1 \vee x_1 \vee \neg x_2)$). On pose alors $k = m$ (le nombre de clauses). Chaque clause donne lieu à un triangle dans G reliant trois nouveaux sommets correspondant aux trois littéraux apparaissant dans la clause (voir la figure 3.4).

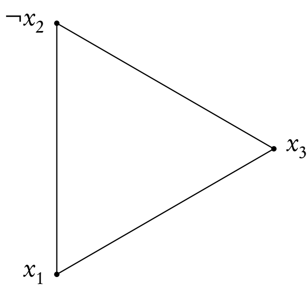

FIGURE 3.4 – Une clause est représentée par un triangle dont les sommets sont les littéraux. Ici, la clause est $(x_1 \vee \neg x_2 \vee x_3)$.

Ainsi, les sommets du graphe G seront tous les littéraux apparaissant dans φ avec leur multiplicité, et il y a des arêtes entre tous les littéraux correspondant à une même clause, formant donc $k = m$ triangles dans G. Mais ce ne sont pas les seules arêtes du graphe : on relie également chaque littéral à sa négation, par exemple on relie toutes les occurences de x_1 à toutes les occurences de $\neg x_1$. On obtient ainsi le graphe G associé à la formule φ, comme illustré à la figure 3.5.

Cette construction de (G, k) à partir de φ est clairement calculable en temps polynomial. Il reste à montrer que φ est satisfaisable ssi G a un ensemble indépendant de taille $k = m$.

Si φ est satisfaisable par une certaine affectation des variables, soit a_i un littéral égal à 1 dans la clause i selon cette affectation (a_i est de la forme x_j ou $\neg x_j$). Par construction, un des sommets de G correspond à ce littéral. Puisque pour $i \neq j$, a_i et a_j sont dans des clauses différentes, ils ne sont pas dans le même triangle de G, et par ailleurs ils ne sont pas la négation l'un de l'autre car il existe une affectation qui les rend tous

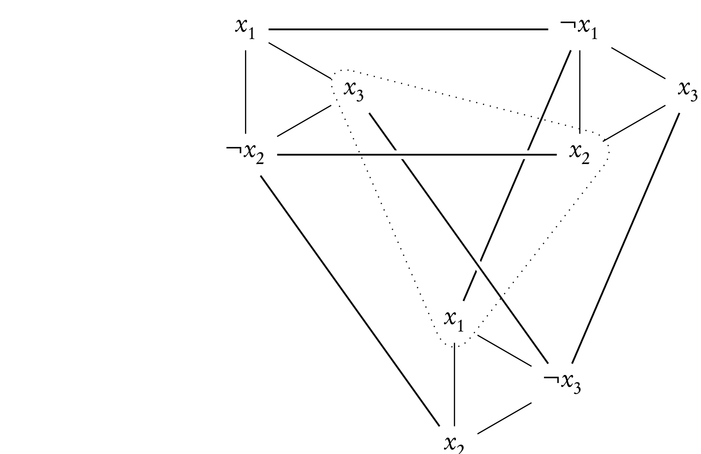

FIGURE 3.5 – Graphe correspondant à la formule $\varphi(x_1,x_2,x_3)=(x_1\vee x_2\vee\neg x_3)\wedge(x_1\vee\neg x_2\vee x_3)\wedge(\neg x_1\vee x_2\vee x_3)$ et ensemble indépendant de taille 3 (en pointillés) montrant que $x_1=x_2=x_3=1$ satisfait φ.

deux vrais. Ainsi, il n'y a pas d'arête entre eux dans G et $a_1,\dots,a_m$ est un ensemble indépendant de taille $k=m$.

Réciproquement, supposons que G ait un ensemble indépendant $\{a_1,\dots,a_m\}$ de taille $k=m$. Si $i\neq j$, a_i et a_j ne peuvent pas être la négation l'un de l'autre sinon ils seraient reliés dans G. Ainsi, il existe une affectation des variables de φ telle que tous les a_i ont valeur 1. De même, il ne peuvent pas correspondre à deux littéraux de la même clause car ceux-ci sont reliés dans G. Cette affectation rend donc au moins un littéral vrai par clause, et φ est satisfaisable. □

3-AF **Remarque** Si k est fixé (et non donné dans l'entrée), le problème se résout en temps polynomial puisqu'il suffit de tester tous les sous-ensembles de k sommets (il y en a moins de n^k).

Par les réductions entre `ENSEMBLE INDÉPENDANT` et `CLIQUE` vues à l'exemple 3-I, nous avons le corollaire suivant.

3-AG **Corollaire**

Le problème `CLIQUE` est NP-complet.

Nous allons maintenant montrer que le problème `SOMME PARTIELLE` vu au chapitre précédent (exemple 2-AM) est NP-complet.

3-AH Proposition

Le problème `SOMME PARTIELLE` est NP-complet.

Idée de la démonstration Il suffit de deviner un sous-ensemble sommant à t pour résoudre ce problème dans NP. Pour la NP-difficulté, on réduit `3SAT` en imposant que chaque clause soit satisfaite grâce à des entiers correspondant aux littéraux et aux clauses.

Démonstration La machine non déterministe polynomiale suivante montre que le problème `SOMME PARTIELLE` est dans NP :

- `deviner` un sous-ensemble $S \subseteq \{1, \dots, m\}$;
- vérifier que $\sum_{i \in S} a_i = t$.

Nous réduisons maintenant `3SAT` au problème `SOMME PARTIELLE` pour montrer la NP-difficulté de ce dernier. Soit $\varphi(x_1, \dots, x_n)$ une instance de `3SAT` (c'est-à-dire une formule en 3-CNF) ayant m clauses $C_0, \dots, C_{m-1}$.

À chacun des deux littéraux $l \in \{x_i, \neg x_i\}$ sur la variable x_i, on associe l'entier

$$a_l = 6^{i+m-1} + \sum_{j \mid l \in C_j} 6^j.$$

En d'autres termes, en base 6 le chiffre de j-ème position est 1 si l apparaît dans la clause C_j, et le chiffre de position $i + m - 1$ est fixé à 1.

Par ailleurs, à chaque clause C_i ($0 \leqslant i < m$), on associe les deux entiers $b_i = b'_i = 6^i$. En d'autres termes, en base 6 le chiffre de i-ème position est fixé à 1.

Enfin, l'entier cible est défini par $t = 3 \sum_{i=0}^{m-1} 6^i + \sum_{i=m}^{m+n-1} 6^i$, c'est-à-dire le nombre dont l'écriture en base 6 est

$$\underbrace{1 \dots 1}_{n \text{ fois}} \underbrace{3 \dots 3}_{m \text{ fois}}.$$

Un exemple est donné à la figure 3.6.

Comme sur l'exemple, on visualisera les nombres a, b, b' et t en base 6, les n premières colonnes représentant les variables $x_n, \dots, x_1$ et les m suivantes représentant les clauses $C_{m-1}, \dots, C_0$. On notera que l'ensemble des nombres a ne peut pas apporter plus de trois chiffres 1 par colonne C_i car chaque clause contient trois littéraux. Ainsi, chaque colonne a au plus cinq chiffres 1 et il n'y a donc jamais de retenue qui peut se propager en base 6.

Cette instance de `SOMME PARTIELLE` est bien sûr calculable en temps polynomial à partir de φ. Supposons que φ soit satisfaisable par l'affectation $(\alpha_1, \dots, \alpha_n)$ de ses variables. Alors on choisit comme solution de `SOMME PARTIELLE` l'ensemble A des entiers a_l tels que l est vrai (c'est-à-dire $l = x_i$ si $\alpha_i = 1$ ou $l = \neg x_i$ si $\alpha_i = 0$). Dans ce cas, $\sum_{a_l \in A} a_l$ s'écrit en base 6 avec n chiffres 1 en tête (sur les colonnes des variables) et les m chiffres suivants sont compris entre 1 et 3. En effet, puisque chaque clause est

Instance de 3SAT

$$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_3).$$

Représentation des entiers en base 6 :

	x_4	x_3	x_2	x_1	C_2	C_1	C_0
a_{x_1}				1	1	0	1
$a_{\neg x_1}$				1	0	1	0
a_{x_2}			1	0	0	0	0
$a_{\neg x_2}$			1	0	1	1	1
a_{x_3}		1	0	0	0	0	1
$a_{\neg x_3}$		1	0	0	1	0	0
a_{x_4}	1	0	0	0	0	1	0
$a_{\neg x_4}$	1	0	0	0	0	0	0
b_1							1
b'_1							1
b_2						1	0
b'_2						1	0
b_3					1	0	0
b'_3					1	0	0
t	1	1	1	1	3	3	3

FIGURE 3.6 – Illustration de la réduction de 3SAT à SOMME PARTIELLE.

satisfaite, il y a au moins un 1 par colonne de clause (et au plus trois par la remarque ci-dessus). Il suffit alors de compléter la somme par les variables b_i et b'_i nécessaires pour amener chaque colonne de clause à la valeur 3. Ainsi, cette instance de SOMME PARTIELLE a une solution.

Réciproquement, si cette instance de SOMME PARTIELLE a une solution, on remarque d'abord que pour chaque variable x_i, soit a_{x_i} soit $a_{\neg x_i}$ a été choisi pour obtenir 1 dans la colonne de x_i. Si a_{x_i} est choisi, on affectera $x_i = 1$; si c'est $a_{\neg x_i}$ qui est choisi, on affectera $x_i = 0$. Puisque chaque colonne de clause vaut 3 et que parmi les nombres b, b', seuls deux peuvent y contribuer, cela signifie qu'il y a au moins un a_l choisi apportant un 1 dans cette colonne. La clause est donc satisfaite par le littéral l. Au total, chaque clause est satisfaite par l'affectation des variables, donc φ est satisfaisable.

La fonction qui à φ associe cette instance de SOMME PARTIELLE est donc une réduction de 3SAT à SOMME PARTIELLE, ce qui montre la NP-difficulté de SOMME PARTIELLE.

□

Un peu de recul

Nous avons vu quelques problèmes NP-complets issus de domaines différents (logique, graphes, nombres) mais il existe des centaines d'autres problèmes NP-complets importants en pratique. C'est ce qui fait de la NP-complétude une notion centrale en complexité et en algorithmique.

On ne connaît d'algorithme polynomial pour aucun d'entre eux. Rappelons que résoudre l'un d'entre eux en temps polynomial revient à tous les résoudre en temps polynomial. Un résultat de NP-complétude est donc considéré comme une « preuve » que le problème est difficile (mais ce n'est pas une vraie preuve tant qu'on n'a pas montré $\mathsf{P} \neq \mathsf{NP}$!).

On pourra trouver de nombreux autres exemples de problèmes NP-complets dans le livre de Garey et Johnson [GJ79] par exemple.

3.2.4 Complémentaire

Nous avons défini la classe coNP à la section 2.2.8. Cette classe aussi admet des problèmes complets. En réalité, tout le travail fait ici se transpose aisément. Nous avons en effet le résultat suivant.

3-AI Lemme

Si A est NP-complet, alors ${}^cA$ est coNP-complet.

Démonstration Comme $A \in \mathsf{NP}$, par définition ${}^cA$ est dans coNP. Il reste à montrer qu'il est coNP-difficile. Soit $B \in \mathsf{coNP}$: par définition, ${}^cB \in \mathsf{NP}$, donc ${}^cB$ se réduit à A via une réduction f calculable en temps polynomial. En d'autres termes, $x \in {}^cB$ ssi $f(x) \in A$. Cela implique que $x \in B$ ssi $f(x) \in {}^cA$, c'est-à-dire que B se réduit à ${}^cA$. Donc ${}^cA$ est coNP-complet. □

La remarque 2-AZ ne remet bien sûr pas en cause ce résultat. On en déduit que les problèmes coSAT, coCLIQUE, etc., sont coNP-complets.

✎ **3-AJ Exercice**

Montrer que le problème TAUTOLOGIE suivant est coNP-complet :

- *entrée* : une formule booléenne $\varphi(x_1, \ldots, x_n)$;
- *question* : φ est-elle une tautologie, c'est-à-dire $\varphi(a_1, \ldots, a_n)$ est-elle vraie pour toute affectation $(a_1, \ldots, a_n)$?

Indication : on pourra faire une réduction très simple de coSAT.

3.2.5 Théorème de Ladner

Dans NP, nous avons donc d'un côté des problèmes NP-complets qui sont difficiles si $\mathsf{P} \neq \mathsf{NP}$, et de l'autre des problèmes P qui sont faciles. Si $\mathsf{P} \neq \mathsf{NP}$, y a-t-il quelque chose entre les deux ? En d'autres termes, si $\mathsf{P} \neq \mathsf{NP}$ existe-t-il des problèmes de NP qui soient hors de P mais pas NP-complet (langages « intermédiaires ») ?

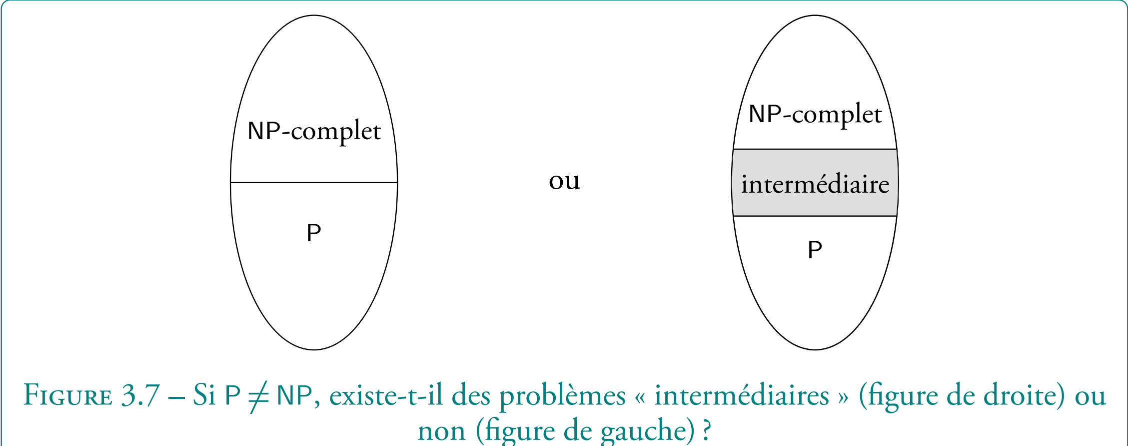

FIGURE 3.7 – Si $\mathsf{P} \neq \mathsf{NP}$, existe-t-il des problèmes « intermédiaires » (figure de droite) ou non (figure de gauche) ?

En s'inspirant de méthodes issues de calculabilité, Ladner [Lad75a] répond à cette question dès 1975.

3-AK Théorème (Ladner, 1975)

Si $\mathsf{P} \neq \mathsf{NP}$ alors il existe un problème $A \in \mathsf{NP}$ tel que :

- $A \notin \mathsf{P}$;
- A n'est pas NP-complet.

Idée de la démonstration Le langage A sera SAT avec des « trous ». Plus précisément, pour certains intervalles de longueur de mots, A sera vide (« trous ») tandis que pour les autres intervalles de longueur de mots, A sera égal à SAT (voir figure 3.8). S'ils sont assez grands, les premiers intervalles garantissent que A n'est pas NP-complet, tandis que les seconds garantissent qu'il n'est pas dans P.

3-AL Remarque Avant de voir la démonstration détaillée, il nous faut formaliser le concept *d'énumération des machines de Turing fonctionnant en temps polynomial*. Le code d'une machine de Turing n'est qu'un mot sur un alphabet fini, ce n'est donc pas difficile d'énumérer le code de toutes les machines, $M_1, M_2, \ldots$ On remarquera que dans cette liste, pour chaque i il y a une infinité de machines équivalentes à M_i puisqu'il suffit d'ajouter des instructions inutiles pour augmenter la taille du code.

En revanche, pour énumérer seulement les machines fonctionnant en temps polynomial il faut une astuce. Il s'agit d'ajouter un ruban à la machine M_i permettant de compter le nombre d'étapes de calcul. Dès que le compteur dépasse $i + n^i$ (où n est la taille de l'entrée), on arrête le calcul en rejetant. On obtient ainsi une nouvelle énumération (M'_i) de certaines machines de Turing, dans laquelle M'_i fonctionne en temps polynomial $(i + n^i)$ mais est éventuellement interrompue au cours de son calcul.

Si M est une machine fonctionnant en temps polynomial $p(n)$, alors il existe k tel que $p(n) \leqslant k + n^k$ pour tout n. Soit M_i l'une des machines équivalentes à M et telle que $i \geqslant k$. Alors M'_i fonctionne en temps $i + n^i \geqslant p(n)$, donc son calcul n'est pas interrompu et elle est aussi équivalente à M. Ainsi, (M'_i) est une énumération de toutes les machines fonctionnant en temps polynomial.

Démonstration (th. 3-AK) Si $h : \mathbb{N} \to \mathbb{N}$ est une fonction strictement croissante, on définit

$$\texttt{SAT}_h = \{\varphi \mid \varphi \in \texttt{SAT} \text{ et } \exists i \; h(2i) \leqslant |\varphi| < h(2i+1)\}.$$

Figure 3.8 – Langage $\texttt{SAT}_h$.

Si, sur l'entrée 1^n, on sait calculer en temps polynomial $h(0), h(1), \ldots, h(i)$ où i est le plus grand entier tel que $h(i) \leqslant n$, alors $\texttt{SAT}_h \in \mathsf{NP}$. En effet, l'algorithme suivant convient pour $\texttt{SAT}_h$ sur l'entrée φ :

- calculer $h(0), h(1), \ldots, h(i)$ où i est le plus grand entier tel que $h(i) \leqslant |\varphi|$;
- si i est impair, `rejeter` ;
- décider (de manière non déterministe) si $\varphi \in \texttt{SAT}$.

L'objectif est maintenant de définir une fonction h ayant cette propriété, et telle que $\texttt{SAT}_h$ ne soit ni dans P ni NP-complet. Pour cela, nous allons utiliser les deux observations suivantes.

1. Puisque $\mathsf{P} \neq \mathsf{NP}$ (par hypothèse) et que $\texttt{SAT}$ est NP-complet, $\texttt{SAT} \notin \mathsf{P}$.
   En d'autres termes, pour toute machine déterministe polynomiale M et tout entier n, il existe φ telle que
   - $|\varphi| \geqslant n$;

- $M(\varphi)$ accepte ssi $\varphi \notin \texttt{SAT}$
  (c'est-à-dire [$\varphi \in \texttt{SAT}$ et $M(\varphi)$ rejette] ou [$\varphi \notin \texttt{SAT}$ et $M(\varphi)$ accepte] ; en d'autres termes, M se trompe sur φ).

2. Si A est un langage fini (donc dans P), puisque $\mathsf{P} \neq \mathsf{NP}$ par hypothèse, SAT ne se réduit pas à A.
   En d'autres termes, pour toute réduction polynomiale f et tout entier n, il existe φ telle que
   - $|\varphi| \geqslant n$;
   - $f(\varphi) \in A$ ssi $\varphi \notin \texttt{SAT}$
     (c'est-à-dire [$\varphi \in \texttt{SAT}$ et $f(\varphi) \notin A$] ou [$\varphi \notin \texttt{SAT}$ et $f(\varphi) \in A$]).

On définit arbitrairement $h(0) = 0$. Les valeurs suivantes de h vont être définies grâce au temps de fonctionnement d'une certaine machine M, qui prendra en entrée des entiers i et $m_0, \dots, m_{i-1}$ (plus tard on prendra $m_j = h(j)$).

Soit (M_i) une énumération des machines fonctionnant en temps polynomial (voir remarque 3-AL) et qui calculent une fonction de $\{0,1\}^\star$ dans $\{0,1\}^\star$: on verra tantôt M_i comme calculant une réduction, tantôt comme acceptant un langage (dans ce mode « acceptation », le mot est accepté ssi le résultat est 1).

Soit M la machine suivante, sur l'entrée $(i, m_0, \dots, m_{i-1})$:

- si i est impair, $i = 2k+1$, énumérer par taille croissante toutes les formules φ de taille $> m_{i-1}$ et simuler $M_k(\varphi)$, jusqu'à trouver φ telle que [$M_k(\varphi) = 1$ ssi $\varphi \notin \texttt{SAT}$] ;
- si i est pair, $i = 2k$, énumérer par taille croissante toutes les formules φ de taille $> m_{i-1}$ et simuler $M_k(\varphi)$, jusqu'à trouver φ telle que :
  - soit $m_j \leqslant |M_k(\varphi)| < m_{j+1}$ pour j pair, et [$M_k(\varphi) \in \texttt{SAT}$ ssi $\varphi \notin \texttt{SAT}$] ;
  - soit $m_j \leqslant |M_k(\varphi)| < m_{j+1}$ pour j impair, et $\varphi \in \texttt{SAT}$;
  - soit $|M_k(\varphi)| \geqslant m_{i-1}$ et $\varphi \in \texttt{SAT}$.

Par les observations 1 et 2 précédentes, de telles formules existent forcément.

On définit alors récursivement h comme suit : $h(i)$ est égal à $h(i-1)$ auquel on ajoute le temps de calcul de $M(i, h(0), \dots, h(i-1))$. On remarquera notamment que la formule φ recherchée par M vérifie $h(i-1) \leqslant |\varphi| < h(i)$ puisque le temps de calcul de M est évidemment supérieur à la taille de φ. De plus, sur l'entrée 1^n, on sait calculer en temps polynomial $h(0), h(1), \dots, h(i)$ où i est le plus grand entier tel que $h(i) \leqslant n$, puisqu'il suffit de simuler les fonctionnements successifs de M jusqu'à ce que le nombre d'étapes total de la simulation dépasse n. Ainsi, $\texttt{SAT}_h \in \mathsf{NP}$ par ce qui précède.

Nous prétendons que $\texttt{SAT}_h$ n'est pas dans P et qu'il n'est pas NP-complet. Pour montrer cela, nous raisonnons par l'absurde.

Si $\mathtt{SAT}_h \in \mathsf{P}$ alors il est reconnu par une machine polynomiale M_k. Par définition de $\mathtt{SAT}_h$, pour tout φ si $h(2k) \leqslant |\varphi| < h(2k+1)$ alors $[\varphi \in \mathtt{SAT}_h$ ssi $\varphi \in \mathtt{SAT}]$. Or par définition de h, il existe une formule φ de taille comprise entre $h(2k)$ et $h(2k+1)$ telle que $[M_k(\varphi) = 1$ ssi $\varphi \notin \mathtt{SAT}]$. Ainsi, M_k donne la mauvaise réponse pour φ, une contradiction.

Si $\mathtt{SAT}_h$ est NP-complet, alors il existe une réduction polynomiale de $\mathtt{SAT}$ à $\mathtt{SAT}_h$ calculée par une machine M_k, c'est-à-dire $\varphi \in \mathtt{SAT} \iff M_k(\varphi) \in \mathtt{SAT}_h$. Or par définition de h, il existe φ de taille comprise entre $h(2k-1)$ et $h(2k)$ telle que :

- soit $h(j) \leqslant |M_k(\varphi)| < h(j+1)$ pour j impair et $\varphi \in \mathtt{SAT}$, dans ce cas $M_k(\varphi) \notin \mathtt{SAT}_h$ et donc M_k n'est pas une réduction de $\mathtt{SAT}$ à $\mathtt{SAT}_h$;
- soit $h(j) \leqslant |M_k(\varphi)| < h(j+1)$ pour j pair et $[M_k(\varphi) \in \mathtt{SAT}$ ssi $\varphi \notin \mathtt{SAT}]$. Or la taille de $M_k(\varphi)$ implique que $[M_k(\varphi) \in \mathtt{SAT}$ ssi $M_k(\varphi) \in \mathtt{SAT}_h]$, donc M_k n'est pas une réduction de $\mathtt{SAT}$ à $\mathtt{SAT}_h$.

Ainsi, aucune machine polynomiale ne décide $\mathtt{SAT}_h$ ni ne réduit $\mathtt{SAT}$ à $\mathtt{SAT}_h$, ce qui conclut la preuve. □

3-AM Remarques

- Le résultat original de Ladner est en fait sensiblement plus fort. On peut en effet faire la même construction à partir d'un langage $A \notin \mathsf{P}$ décidable quelconque, plutôt que de partir de $\mathtt{SAT}$. On obtient donc l'énoncé suivant : si $A \notin \mathsf{P}$ est décidable, alors il existe $B \notin \mathsf{P}$ tel que $B \leqslant_m^{\mathsf{p}} A$ et $A \nleqslant_m^{\mathsf{p}} B$.
  En d'autres termes, sous l'hypothèse $\mathsf{P} \neq \mathsf{NP}$ il existe une infinité de langages
  $$\mathtt{SAT} >_m^p A_1 >_m^p A_2 >_m^p \dots$$
  tels que A_{i+1} se réduit à A_i mais A_i ne se réduit pas à A_{i+1}.
- On ne connaît pas de langage qui soit intermédiaire sous l'hypothèse $\mathsf{P} \neq \mathsf{NP}$ et qui soit plus « naturel » que la construction de Ladner ci-dessus (bien que le problème de la factorisation d'entiers et celui de l'isomorphisme de graphes que l'on verra au chapitre 10 soient des candidats).
- Il découle du théorème de Ladner que la NP-complétude *ne peut pas* être définie ainsi : « les langages $A \in \mathsf{NP}$ tels que $A \in \mathsf{P} \implies \mathsf{P} = \mathsf{NP}$ ». En effet, si $\mathsf{P} \neq \mathsf{NP}$ alors tout langage intermédiaire (un langage de NP qui ne soit ni complet ni dans P) serait complet (car « faux implique faux »), une absurdité.

✎ **3-AN Exercice**

Montrer le premier point de la remarque précédente.

✎ **3-AO Exercice**

Montrer qu'il existe un langage $A \notin \mathsf{EXP}$ qui ne soit pas EXP-difficile.

Indication : grâce à une stratégie de preuve similaire à celle pour le théorème de Ladner, transformer un problème $L \notin \mathsf{EXP}$ en un problème $A \notin \mathsf{EXP}$ qui ne soit pas EXP-difficile.

3.2.6 Théorème de Mahaney

En guise d'échauffement à cette partie, on pourra s'entraîner à résoudre l'exercice B-F.

Dans l'objectif de mieux comprendre les problèmes NP-complet, nous allons maintenant voir qu'un problème NP-difficile doit posséder beaucoup d'éléments. Plus précisément, si $\mathsf{P} \neq \mathsf{NP}$ alors il ne peut pas être « creux » au sens de la définition suivante.

3-AP Définition

Un langage A est dit *creux* s'il existe un polynôme $p(n)$ tel que pour tout n, le nombre de mots de A de taille n est au plus $p(n)$.

De manière équivalente, A est creux si le nombre de mots de A de taille $\leqslant n$ est majoré par un polynôme $q(n)$ (prendre $q(n) = \sum_{i=0}^{n} p(i)$).

On rappelle que $A^{=n}$ et $A^{\leqslant n}$ désignent respectivement l'ensemble des mots de A de taille n et de taille $\leqslant n$. On a donc $|A^{=n}| \leqslant p(n)$ et $|A^{\leqslant n}| \leqslant q(n)$ pour tout n.

Mahaney [Mah82] a en effet montré le résultat suivant.

3-AQ Théorème (Mahaney, 1982)

S'il existe un langage creux NP-difficile, alors $\mathsf{P} = \mathsf{NP}$.

Idée de la démonstration Sous l'hypothèse d'un langage creux NP-difficile, nous donnons un algorithme polynomial pour SAT. Celui-ci recherche la solution maximale (si elle existe) d'une formule φ. Si S est un langage creux NP-difficile et f une réduction de SAT à S, l'idée est de parcourir l'arbre des solutions possibles en se servant de f et du caractère « creux » de S pour élaguer des branches de l'arbre et n'explorer qu'un nombre polynomial de nœuds à chaque niveau.

Démonstration Supposons que S soit un langage creux NP-difficile possédant $\leqslant p(n)$ mots de taille $\leqslant n$, et que SAT se réduise à S via f (calculable en temps polynomial) : $[\varphi \in \mathtt{SAT}$ ssi $f(\varphi) \in S]$. Soit q un polynôme tel que $|f(\varphi)| \leqslant q(|\varphi|)$. Nous allons nous servir de f et du caractère « creux » de S pour donner un algorithme polynomial pour SAT. Sur l'entrée φ, celui-ci va rechercher la solution de φ maximale pour l'ordre lexicographique (si elle existe).

Si $\varphi(x)$ est une instance de SAT (où $x = x_1 \ldots x_n$), pour $y = y_1 \ldots y_n$ on note $\varphi_y(x)$ la formule $[(x \geqslant y) \wedge \varphi(x)]$, où $x \geqslant y$ signifie que x est après y dans l'ordre lexicographique et peut s'écrire simplement avec les symboles autorisés habituels en disant que soit $x = y$, soit il existe un préfixe identique de taille $k-1$ et que le k-ème bit de x est 1 tandis que celui de y est 0 :

$$(\bigwedge_{i\in[1,n]} x_i \leftrightarrow y_i) \vee \bigvee_{k\in[1,n]} \left(\left(\bigwedge_{i<k} x_i \leftrightarrow y_i\right) \wedge (x_k \wedge \neg y_k)\right).$$

Ainsi, $\varphi_y \in \texttt{SAT}$ ssi φ possède une solution x supérieure à y (pour l'ordre lexicographique). Le code de φ_y est bien sûr calculable en temps polynomial à partir de y et de φ. On notera m la taille d'une formule φ_y.

L'avantage de considérer les formules φ_y est que l'ensemble $\{y \mid \varphi_y \in \texttt{SAT}\}$ est clos vers la gauche, c'est-à-dire que si $\varphi_{y'} \in \texttt{SAT}$ et $y \leqslant y'$ alors $\varphi_y \in \texttt{SAT}$ (cf. illustration figure 3.9). Cela va faciliter la recherche de la solution maximale de φ.

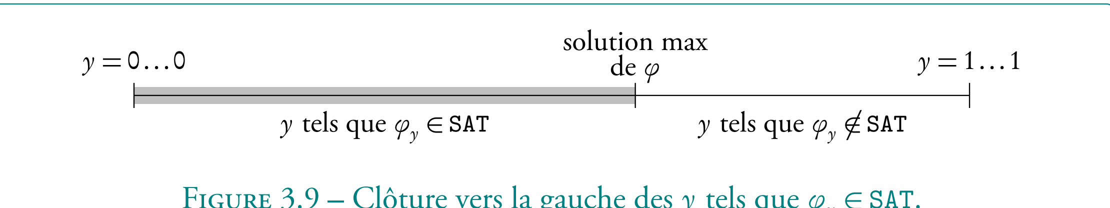

FIGURE 3.9 – Clôture vers la gauche des y tels que $\varphi_y \in \texttt{SAT}$.

Notre algorithme pour rechercher la solution maximale de φ va développer l'arbre binaire complet des φ_y pour $|y| = n$, dans lequel :

- les 2^i nœuds à la profondeur i sont tous les $\varphi_{z0^{n-i}}$ pour $|z| = i$;
- les fils de $\varphi_{z0^{n-i}}$ sont $\varphi_{z'0^{n-i-1}}$ et $\varphi_{z''0^{n-i-1}}$ où $z' = z0$ et $z'' = z1$.

La branche descendant d'un nœud $z0\ldots0$ est donc l'ensemble des y ayant z comme préfixe. Grâce à f et au caractère « creux » de S, nous pourrons garder seulement $p(q(n))$ nœuds à chaque niveau (cf. figure 3.10). Il faut toutefois maintenir l'invariant suivant :

à chaque niveau, l'une des branches restantes contient la solution maximale de φ
(de sorte que l'une des feuilles est égale à cette solution maximale).

Il y a deux raisons grâce auxquelles on peut supprimer des nœuds dans l'arbre tout en préservant l'invariant :

- si $f(\varphi_y) = f(\varphi_{y'})$ et $y < y'$, alors par définition de f soit φ_y et $\varphi_{y'}$ sont tous deux dans SAT, soit ils sont tous deux hors de SAT. Donc la solution x maximale de φ vérifie soit $x \geqslant y'$, soit $x < y$. Dans les deux cas, il n'est pas nécessaire d'explorer la branche de y. Cela permet de se restreindre aux nœuds ayant des images par f toutes différentes.

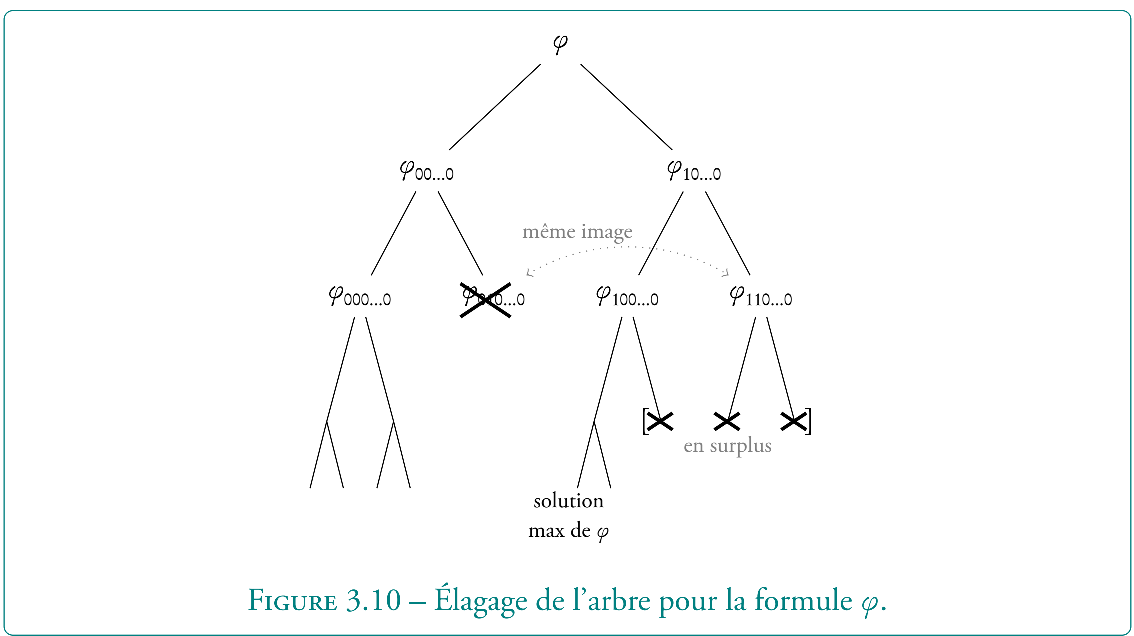

FIGURE 3.10 – Élagage de l'arbre pour la formule φ.

– Si après application de l'élagage précédent, un niveau contient encore un nombre $N > p(q(m))$ nœuds, on peut supprimer les $(N - p(q(m)))$ plus grands nœuds, car ils sont « en surplus ». En effet, les images des N nœuds sont toutes différentes (par le point précédent), donc au moins $N - p(q(m))$ nœuds ont leur image hors de S (puisque $|S^{\leqslant q(m)}| \leqslant p(q(m))$), c'est-à-dire qu'ils sont eux-mêmes hors de SAT. Et par clôture à gauche, seuls les $p(q(m))$ premiers peuvent éventuellement appartenir à SAT.

L'algorithme parcourt l'arbre niveau par niveau en élaguant les branches inutiles, de sorte qu'il ne conserve qu'un nombre polynomial de nœuds à chaque niveau (au plus $p(q(m))$). Au dernier niveau de l'arbre (le niveau n), il y a $\leqslant p(q(m))$ feuilles $y_1, \ldots, y_k$. Par l'invariant, l'une d'entre elles correspond à la solution maximale de φ si elle existe. Il suffit alors de vérifier si $\varphi(y_i)$ est vrai pour l'une des feuilles y_i. Puisqu'il y a n niveaux, l'algorithme est polynomial.
Plus précisément, on maintient l'ensemble Y_i des préfixes y indexant les nœuds du niveau i. Au départ, $Y_1 \leftarrow \{0,1\}$. Voici la procédure pour calculer Y_{i+1}, où U est un ensemble permettant de conserver les images par f des nœuds vus jusqu'à présent. On énumère les nœuds du niveau précédent par ordre décroissant afin de pouvoir supprimer le plus petit nœud lorsque deux nœuds ont la même image.

- $Y_{i+1} \leftarrow \emptyset$;
- $U \leftarrow \emptyset$;
- `Pour chaque` $y \in Y_i$ par ordre décroissant `faire`
  - $u \leftarrow f(\varphi_{y10\ldots0})$,

◦ `si` $u \notin U$ `alors`

· $Y_{i+1} \leftarrow Y_{i+1} \cup \{y1\}$

· $U \leftarrow U \cup \{u\}$,

◦ $u \leftarrow f(\varphi_{y00\ldots0})$,

◦ `si` $u \notin U$ `alors`

· $Y_{i+1} \leftarrow Y_{i+1} \cup \{y0\}$

· $U \leftarrow U \cup \{u\}$;

– si Y_{i+1} contient $N > p(q(m))$ éléments, alors supprimer les $(N - p(q(m)))$ plus grands.

Une fois que Y_n est calculé, il suffit de tester si $\varphi(y)$ est vrai pour l'un des $y \in Y_n$. Si c'est le cas, alors $\varphi \in$ SAT, sinon $\varphi \notin$ SAT puisque par l'invariant, Y_n contient la solution maximale de φ si elle existe. Cet algorithme pour SAT fonctionne en temps polynomial, donc P = NP. □

3.2.7 Algorithme polynomial pour SAT si P = NP

Pour compléter ce chapitre sur NP, nous allons voir que la question « P = NP ? » se ramène à l'analyse d'un unique algorithme. Il nous faut tout d'abord un lemme simple qui montre comment transformer un problème d'évaluation en un problème de décision.

3-AR Lemme

Si P = NP alors il existe une machine déterministe fonctionnant en temps polynomial qui, sur l'entrée φ (une instance de SAT), accepte et renvoie une affectation satisfaisant φ si φ est satisfaisable, ou rejette sinon.

Idée de la démonstration Il s'agit de réaliser une recherche préfixe de la plus petite solution de φ.

Démonstration Soit L le langage $L = \{(\varphi, a) \mid \exists b, \varphi(ab) = 1\}$, où a est le début d'une affectation (partielle) de φ et b est la fin de cette affectation. En d'autres termes, L permet de savoir si l'on peut compléter a de manière à obtenir une solution de φ. C'est bien sûr un langage de NP donc, par hypothèse, de P. Voici l'algorithme pour trouver la plus petite solution de $\varphi(x_1, \ldots, x_n)$:

– $a \leftarrow \epsilon$ (affectation vide) ;

– `pour` i `de` 1 `à` n `faire`

    - si $(\varphi, a0) \in L$ alors $a \leftarrow a0$,
    - sinon $a \leftarrow a1$;
- si $\varphi(a) = 1$ alors `accepter` et renvoyer a,
- sinon `rejeter`.

C'est ainsi une recherche préfixe de la plus petite solution de φ, qui se déroule en temps polynomial puisque le test $(\varphi, a0) \in L$ se fait en temps polynomial. □

Si $\mathsf{P} = \mathsf{NP}$ alors `SAT` possède un algorithme polynomial, mais on ne sait pas lequel. Avec un peu d'astuce on peut tout de même donner un algorithme A fixé tel que A résout « presque » `SAT` et fonctionne en temps polynomial si $\mathsf{P} = \mathsf{NP}$. Il s'agit d'une variante de l'algorithme optimal de recherche universelle de Levin [Lev73]. L'énoncé précis est donné à la proposition 3-AT.

Pour décrire l'algorithme, on aura besoin comme à la remarque 3-AL d'une énumération (M_i) des machines fonctionnant en temps polynomial. Ici, ces machines sont censées renvoyer une affectation des variables de la formule φ donnée en entrée (comme au lemme 3-AR).

Voici cet algorithme M sur une entrée φ de taille n (une formule booléenne, c'est-à-dire une instance de `SAT`).

- $i \leftarrow 1$;
- pour i de 1 à n `faire`
    - simuler $M_i(\psi)$ pour toute formule ψ de taille $\leqslant \log n$,
    - si pour tout ψ, M_i donne toujours une affectation valide lorsque $\psi \in$ `SAT`, sortir de la boucle ;
- simuler $M_i(\varphi)$, produisant un résultat a (une affectation des variables de φ) ;
- si $\varphi(a) = 1$ `accepter`, sinon `rejeter`.

3-AS **Remarque** Comme dans tout langage de programmation, la valeur de i après la boucle pour est égale soit à la valeur de i au moment de la sortie de boucle si on sort prématurément, soit à $n+1$ si la boucle s'est déroulée complètement.

3-AT **Proposition**

Si $\mathsf{P} = \mathsf{NP}$ alors la machine M ci-dessus vérifie :

- M fonctionne en temps polynomial ;
- M décide $\mathtt{SAT}$ pour toute entrée suffisamment grande, c'est-à-dire qu'il existe m tel que pour tout φ, si $|\varphi| \geqslant m$ alors $[M(\varphi) = 1$ ssi $\varphi \in \mathtt{SAT}]$.

Idée de la démonstration M simule les machines M_i les unes après les autres. Si $\mathsf{P} = \mathsf{NP}$ alors l'une d'entre elles, M_{i_0}, résout $\mathtt{SAT}$. Pour toutes les entrées suffisamment grandes, on atteint M_{i_0} et notre algorithme donne alors la bonne réponse. Le temps de calcul est alors celui de M_{i_0}, qui est polynomial.

Démonstration Si $\mathsf{P} = \mathsf{NP}$, soit M_{i_0} la première machine de l'énumération qui renvoie une affectation valide pour toute formule $\varphi \in \mathtt{SAT}$. Toute machine M_i pour $i < i_0$ se trompe donc sur au moins une formule (c'est-à-dire qu'il existe $\varphi \in \mathtt{SAT}$ et $M_i(\varphi)$ n'est pas une affectation valide) : soit N la taille minimum telle que $\forall i < i_0$, M_i se trompe sur une formule de taille $< N$.

Alors pour $|\varphi| \geqslant \max\{i_0, 2^N\}$, la boucle $\mathtt{pour}$ trouve une erreur pour toutes les machines M_i telles que $i < i_0$, donc on sort de la boucle pour $i = i_0$. On simule alors la machine M_{i_0} qui donne une affectation valide si une telle affectation existe. Ainsi, $M(\varphi) = 1$ si $\varphi \in \mathtt{SAT}$, et bien sûr, par conception de l'algorithme, pour tout $\varphi \notin \mathtt{SAT}$, $M(\varphi) = 0$.

Pour le temps d'exécution, rappelons que M_i fonctionne en temps $O(n^i)$, donc on simule M_i en temps $O(n^{2i})$ par la machine universelle de la proposition 1-Q. On exécute au plus i_0 fois la boucle $\mathtt{pour}$, toutes les machines simulées fonctionnent en temps $\leqslant n^{i_0}$ et sont simulées sur $2^{\log n} = n$ formules de taille $\log n$. Donc la boucle $\mathtt{pour}$ prend un temps $\leqslant i_0 n (\log n)^{2i_0}$. Puis la simulation de $M_{i_0}(\varphi)$ prend un temps $O(n^{2i_0})$: au total, M fonctionne donc en temps polynomial. □

3-AU **Remarques**

- Si $\mathsf{P} \neq \mathsf{NP}$ alors la machine M ci-dessus ne fonctionne pas en temps polynomial car elle va alors simuler des machines M_i pour i de plus en plus grand. Or M_i fonctionne en temps n^i, ce qui fait que M n'est pas polynomiale. De plus, il n'y a pas de raison qu'elle reconnaisse correctement $\mathtt{SAT}$, même sur des instances suffisamment grandes.
- On ne sait pas construire une telle machine M qui fonctionnerait sur toute entrée φ (et non seulement sur des entrées suffisamment grandes).

Pour compléter les réflexions sur la NP-complétude, les exercices B-F et B-C en annexe abordent des thèmes traités dans ce chapitre.