Chapitre III

Complexité temporelle

Introduction

— On se concentre maintenant sur des problèmes calculables.
— On s'intéresse au temps nécessaire pour résoudre ces problèmes.

On va donc maintenant introduire :

— la notion de complexité temporelle.
— les différentes classes : **P**, **NP** et **coNP**.
— exemples de problèmes **NP**.
— la vérifiabilité (réduction en temps polynomial).
— **NP**-complétude.

Dans ce cours, complexité = complexité temporelle.

1 Rappel sur les ordres asymptotiques

1.1 Bornes asymptotiques

Soit f et g des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$.

Définition 15 : grand O
définition : $f(n) = O(g(n))$ si $\exists c > 0, n_0 \in \mathbb{N}^* \mid \forall n \geq n_0, f(n) \leq c.g(n)$
se lit : g est une borne asymptotique supérieure pour f.
se comprend : f ne grandit pas plus vite que g.

Définition 16 : grand oméga Ω
définition : $f(n) = \Omega(g(n))$ si $\exists c > 0, n_0 \in \mathbb{N}^* \mid \forall n \geq n_0, c.g(n) \leq f(n)$
note : si $f(n) = O(g(n))$ alors $g(n) = \Omega(f(n))$.
se lit : g est une borne asymptotique inférieure pour f.
se comprend : f grandit au moins aussi vite que g.

Remarques:

- attention à l'ordre d'écriture.
- $f(n) = O(g(n)) = O(h(n))$ signifie $f(n) = O(g(n))$ et $g(n) = O(h(n))$
- sert à quantifier la vitesse de croissance d'une fonction le plus souvent croissante.

1.2 Domination asymptotiques

Soit f et g des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$.

Définition 17 : petit o
définition : $f(n) = o(g(n))$ si $\forall \epsilon > 0, \exists n_0 \in \mathbb{N}^* \mid \forall n \geq n_0, f(n) \leq \epsilon.g(n)$
se lit : f est dominée asymptotiquement par g.
se comprend : f n'est jamais plus grand que g

Définition 18 : petit ω
définition : $f(n) = \omega(g(n))$ si $\forall c > 0, \exists n_0 \in \mathbb{N}^* \mid \forall n \geq n_0, c.g(n) \leq f(n)$
note : $f(n) = o(g(n))$ alors $g(n) = \omega(f(n))$.
se lit : f domine g asymptotiquement.
se comprend : f n'est jamais plus petit que g

Remarques:
- attention à l'ordre d'écriture.
- sert à quantifier un terme (fonctionnel) que l'on cherche à négliger.

1.3 Ordres de grandeur

Ordres de grandeur : classés par vitesse de croissance

Grand O	vitesse de croissance	exemple
1	constante	10
$\log\log n$	loglogarithmique	
$\log n$	logarithmique	$\log(n^2)$
$\log(n)^c, c > 1$	polylogarithmique	$\log(n)^2$
$n^c, c \in (0,1)$	puissance fractionnaire	$n^{1/2}$
n	linéaire	$3n$
$n \log n$ $\log(n!)$	quasi-linéaire	
n^2	quadratique	$2n^2$
$n^c, c > 1$	polynomiale	$n^{3/2}$
e^n $c^n, c > 1$	exponentielle	2^n
$e^{n^c}, c > 1$ n^n	(poly)exponentielle	$2^{n^2}, n^n$ $n!$

Note : formule de Stirling $n! \sim \sqrt{2\pi n}(n/e)^n$

2 Introduction à la complexité temporelle

2.1 Complexité d'une machine de Turing

Les ordres asymptotiques permettent de quantifier la vitesse d'exécution d'un algorithme.
Exemple 10:
soit le langage $A = \{0^k 1^k | k \geq 0\}$
Ce langage est décidable.
Une MT qui décide A est :
$M_1(\langle w \rangle)$ = ① parcourir la bande
rejeter si on trouve un 0 à droite d'un 1.
② tant qu'il reste à la fois des 0 et des 1 sur la bande.
parcourir la bande et barrer un 0 et un 1
③ si tous les 0 et les 1 sont barrés alors accepter
sinon rejeter.

Exécutions :

w	`00001011`	w	`00011111`	w	`00001111`
①	rejeter	①	`00011111`	①	`00001111`
		②	`x00x1111`	②	`x000x111`
		②	`xx0xx111`	②	`xx00xx11`
		②	`xxxxxx11`	②	`xxx0xxx1`
		③	rejeter	②	`xxxxxxxx`
				③	accepter

Quel est le temps mis par la MT M_1 pour décider si son entrée $w \in A$?
Pour une MT,
- ce temps se mesure en nombre de transitions (ou de pas).
- il dépend de la longueur n de l'entrée w
  on note $n = |w|$.

Analyse de M_1 :
① prend au plus $O(n)$ pas.
② répétition au plus $n/2$ fois de $O(n)$ pas $\Rightarrow O(n^2)$.
③ prend $O(n)$ pas.
Donc, M_1 prend $O(n^2)$ pas pour décider si $w \in A$.

3 Classe de complexité

Rappel : Une MT M déterministe est un décideur si elle s'arrête sur toute ses entrées.
$\Rightarrow$ M décide $\mathcal{L}(M)$.

Définition 19 : Temps d'exécution
Le **temps d'exécution** de M est la fonction $f : \mathbb{N} \to \mathbb{N}$ où $f(n)$ est le nombre maximum de pas nécessaires à M pour traiter une chaîne d'entrée de longueur n.

Remarques :
- on dit alors que :
  - M s'exécute en temps $f(n)$.
  - M est une MT à temps $f(n)$.
- défini ici en nombre de transitions sur une MT.
  en nombre de cycles ou d'instructions sur un processeur.

Définition 20 : classe de complexité temporelle
Soit $t : \mathbb{N} \to \mathbb{R}^+$ une fonction.
La classe de complexité temporelle $\texttt{TIME}(t(n))$ est l'ensemble de tous les langages décidables par une MT à temps $O(t(n))$.

Exemple 11: (version optimisée de l'exemple 10)
Pour le langage $A = \{0^k 1^k | k \geq 0\}$.
La MT M_1 qui décide A s'exécute en temps $O(n^2)$.
Donc, $A \in \texttt{TIME}(n^2)$.
Peut-on trouver une autre MT qui s'exécute plus rapidement ?

$M_2(\langle w \rangle)$ **=** ① parcourir la bande
rejeter si on trouve un 0 à droite d'un 1.
② tant qu'il reste à la fois des 0 et des 1 sur la bande.
si le nombre total de 0 et de 1 est impair, alors rejeter.
barrer un 0 sur deux, puis un 1 sur deux.
③ si tous les 0 et les 1 sont barrés alors accepter,
sinon rejeter.

Exécutions :

w	00001111	w	000000111111	w	00111111
①	00001111	①	000000111111	①	00111111
②	x0x0x1x1	②	x0x0x0x1x1x1	②	x0x1x1x1
②	xxxxxxxx	②	xxx0xxxxx1xx	②	xxxxx1x1
③	accepter	②	xxxxxxxxxxxx	③	rejeter
		③	accepter		

Pour 0^k1^k, à chaque exécution de la boucle ②,
- pour M_1, un seul 0 et un seul 1 sont barrés.
  $\Rightarrow$ la boucle ② est exécutée k fois.
- pour M_2, le nombre de 0 et de 1 est divisé par 2.
  $\Rightarrow$ la boucle ② est exécutée $\log_2 k$ fois.
  Rappel : doubler k fois $\Rightarrow 2^k$ fois plus grand.

Analyse de M_2 :
① prend au plus $O(n)$ pas.
② répétition au plus $O(\log n)$ fois de $n/2$ pas $\Rightarrow O(n \log n)$.
③ prend $O(n)$ pas.
Le temps d'exécution de M_2 est $O(n \log n)$.
Par conséquent, $A \in \texttt{TIME}(n \log n)$.

Questions :
- Contradiction entre $A \in \texttt{TIME}(n^2)$ et $A \in \texttt{TIME}(n \log n)$?
  A appartient aux deux.
  Il est toujours possible de faire plus lent (par exemple en faisant autre chose).
- Existe-t-il encore plus rapide ?
  Sur des MTs à une bande, non.
  On peut d'ailleurs montrer le théorème suivant.

Théorème 41 : complexité des langages réguliers (non prouvé)

Un langage régulier peut être décidé par une MT simple bande à temps $n \log n$.

Montrons que l'on peut faire mieux avec une MT à 2 bandes.

3.1 Cas des machines de Turing multi-bandes

Exemple 12: (version multibande de l'exemple 11)

Avec une MT à deux bandes :

$M_3(\langle w \rangle)$ **=** ① parcourir la bande
  rejeter si on trouve un 0 à droite d'un 1
② parcourir les 0 de la bande 1
  copier les 0 de la bande 1 vers la bande 2
③ parcourir les 1 de la bande 1
  barrer en même temps, les 1 sur la bande 1
  et les 0 sur la bande 2
④ si tous les 0 et les 1 sont barrés alors accepter
  Sinon rejeter.

Exécutions :

	bande 1	bande 2
w	`00001111`	`________`
①	`00001111`	`________`
②	`____1111`	`0000____`
②	`____xxxx`	`xxxx____`
③	accepter	

	bande 1	bande 2
w	`00000111`	`________`
①	`00000111`	`________`
②	`_____111`	`00000___`
②	`_____xxx`	`00xxx___`
③	rejeter	

Analyse de M_3 :
① prend au plus n pas.
② prend au plus $n/2$ pas.
③ prend au plus $n/2$ pas.
④ prend au plus n pas.
Le temps d'exécution de M_3 est $O(n)$.
Par conséquent, $A \in \texttt{TIME}(n)$.

Conclusion :

Les MTs simple bande et multi-bandes ont donc :
— la même puissance en terme de calculabilité.
ils peuvent résoudre les mêmes problèmes.
— une puissance différente en terme de complexité.
ils ne les résolvent pas à la même vitesse.

Théorème 42 :
Soit $t(n)$ une fonction telle que $t(n) \geq n$.
Pour toute MT à k-bandes qui s'exécute en temps $t(n)$,
il existe une MT à une bande qui s'exécute en temps $O(k^2 t(n)^2)$.

Démonstration:
Soit M_k une MT à k bandes qui s'exécute en temps $t(n)$.
Construisons une MT à une bande qui s'exécute en temps $O(k^2 t(n)^2)$.
On a vu comment simuler M_k sur M :
— M stocke sur sa bande les k bandes de M_k en les séparant par #.
— position du pointeur sur une bande = symbole marqué (une marque par bande).

Simulation de M comme M_k :
— parcourir la bande pour lire les caractères sous chaque pointeur.
— parcourir la bande pour mettre à jour le caractère et la position de chaque pointeur.
— si l'on dépasse l'extrémité, on ajoute un espace en décalant le contenu de la bande.
Portion active des bandes :
— M_k s'exécute en temps $t(n)$, chacune de ses bandes accèdent **au plus** les $t(n)$ premières cellules.
— M utilise donc **au plus** les $k \times t(n) + k + 1 = O(kt(n))$ premières cellules.
$k + 1$ = les séparateurs de bandes.

M fait **à chaque pas** (dans le pire des cas) :

1. parcourir la bande pour lire les caractères sous chaque pointeur.
prend au plus un temps $O(k.t(n))$ où $t(n)$ est le temps d'exécution.
2. ajouter un espace à chaque bande
un ajout d'espace = un parcours de la bande en temps $O(kt(n))$.
au pire, k espace sont ajoutés.

3. mettre à jour la position de chaque pointeur et du caractère pointé.
   prend au plus un temps $O(kt(n))$.

Comme ceci est réalisé $O(kt(n))$ fois
$\Rightarrow$ l'exécution de M se fait en temps $O(k^2t(n)^2)$. □

3.2 Temps polynomial

Définition 21 : Temps polynomial
si un MT M s'exécute en temps $t(n) = O(n^c)$ avec $c > 1$,
alors on dit que M s'exécute en temps polynomial.

Corollaire 43: complexité d'une MT multi-bandes
Toute MT M_k à k-bandes à temps polynomial possède
une MT M à une bande équivalente à temps polynomial.

Démonstration:
Si M_k s'exécute en temps $t_k(n) = O(n^c)$, alors M s'exécute en temps $t(n) = O(k^2t_k(n)^2) = O(k^2n^{2c}) = O(n^{c'})$ où $c' > 2c > 1$.
Donc, M s'exécute aussi en temps polynomial. □

Conséquence : Multiplier les bandes ne permet pas de change pas la classe de complexité.

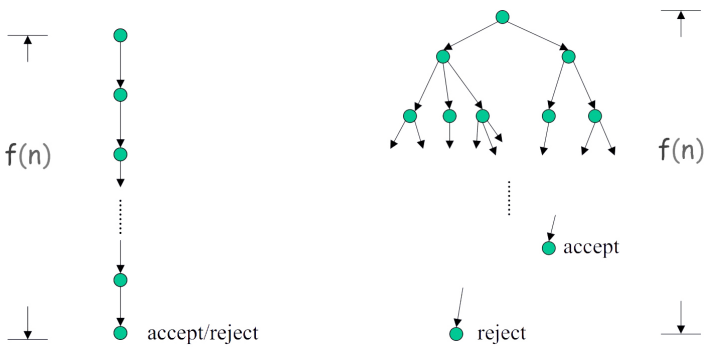
3.3 Complexité d'un machine de Turing non-déterministe

Qu'en est-il pour le cas non-déterministe ?

Définition 22 : Décideur dans le cas d'une MTND
une MTND M est un décideur si l'évaluation de toutes ses branches s'arrête pour toutes les entrées.
$\Rightarrow$ M décide $\mathcal{L}(M)$.

Définition 23 : Temps d'exécution d'une MTND
le **temps d'exécution** de M est une fonction $f : \mathbb{N} \rightarrow \mathbb{N}$ où $f(n)$ est le nombre maximum de transitions que M traverse dans n'importe laquelle de ses branches de calcul sur une entrée de longueur n.
$\Rightarrow$ $f(|w|)$ est le temps d'exécution de la branche la plus longue de $M(\langle w \rangle)$.

Comparaison de la complexité temporelle :
MT déterministe MT non déterministe



Théorème 44 : Complexité temporelle d'une MT non déterministe
Soit $t(n)$ une fonction telle que $t(n) \geq n$. Pour toute MT non déterministe M' qui s'exécute en temps $t(n)$, il existe une MT déterministe M à une bande qui s'exécute en temps $2^{O(t(n))}$.

Démonstration:
Pour M', chaque branche à un temps de calcul au plus $t(n)$. Chaque nœud de M' a au plus b fils (= nb maximum de choix de transitions). Donc, le nombre de feuilles de M' est au plus de $O(b^{t(n)})$. Le temps de traitement d'une branche est au plus de $O(t(n))$. L'ordre de parcours des nœuds importe peu : dans le cas le pire, on passe dans toutes les branches. Donc, le temps total de simulation de M' par M est en temps $O(t(n)b^{t(n)})$. Or $O(t(n)b^{t(n)}) = O(e^{t(n)\log b}) = O(2^{t(n)\log b/\log 2}) = 2^{O(t(n))}$. □

4 Classe P (temps polynomial)

4.1 Définition

Définition 24 : classe de complexité P
P est la classe des langages qui sont décidables en temps polynomial par une MT à simple bande.
Autrement dit : $\mathbf{P} = \bigcup_k \texttt{TIME}(n^k)$

Remarques:
- **P** est invariant pour tous les modèles de machine simulable sur une MT simple bande en un temps polynomial.
  i.e. si un modèle de machine M' résout un problème en temps polynomial et que la simulation de M' sur une MT M se fait en temps polynomial, alors M peut résoudre le même problème en temps polynomial.
- **P** correspond à la classe des problèmes qui sont solvables sur un ordinateur en un temps réaliste.

Notes:
- Description d'un algorithme.
  Le fonctionnement d'un algorithme sera, à partir de maintenant, décrit comme une suite d'étapes. Chaque étape pouvant être réalisée par un nombre plus ou moins grand de transitions sur une MT.
- Démontrer qu'un algorithme s'exécute en temps polynomial.
  Pour une entrée w de longueur n, pour chaque étape de l'algorithme :
  — donner une borne supérieure de la complexité temporelle de chaque étape en fonction de n.
  — s'assurer que chaque étape de l'algorithme s'implémente avec un modèle déterministe raisonnable en temps polynomial.
  L'algorithme s'exécute alors en temps polynomial.

4.2 Codage de l'entrée

La complexité temporelle dépend de n = la taille de l'entrée w.
Il est donc important d'utiliser un codage raisonnable de l'entrée sous peine d'augmenter artificiellement sa complexité temporelle.
On utilise la notation $\langle w \rangle$ pour indiquer un codage raisonnable de l'entrée :
— pour un graphe, un codage (V, E) est raisonnable.
— pour une ADF, un codage de $(Q, \Sigma, \delta, q_0, F)$ est raisonnable.
— pour un nombre, un codage en unaire n'est pas raisonnable (exemple : 5 = 11111).
— pour un nombre, un codage en base $k > 1$, est raisonnable (codage $\log_k$ plus efficace qu'un codage unaire).

4.3 Langage PATH et RELPRIME

Définition 25 : langage PATH

PATH = $\{\langle G, s, t\rangle \mid G$ est un graphe avec un chemin de s vers $t\}$

Donc $\langle G, s, t\rangle \in$ PATH ssi il existe un chemin entre les sommets s et t dans le graphe $G = (V, E)$, ce qui ne peut être le cas que si $(s, t) \in V^2$.

Théorème 45 : PATH $\in$ **P**

Définition 26 : langage RELPRIME

RELPRIME = $\{\langle x, y\rangle \mid x$ et y sont entiers et PGCD$(x, y) = 1\}$

Donc $\langle x, y\rangle \in$ RELPRIME ssi (x, y) sont des entiers premiers entre eux.

Théorème 46 : RELPRIME $\in$ **P**

Démonstration: PATH $\in$ **P**

Montrons qu'il existe un décideur M à temps polynomial pour PATH.

$M(\langle G, s, t\rangle)$ **=** ① dans G, marquer le sommet s.
② répéter jusqu'à ce que plus aucun nouveau sommet ne soit marqué dans la liste des sommets :
parcourir la liste des arêtes.
si un seul sommet est marqué, marquer l'autre.
③ si t est marqué alors accepter sinon rejeter.

Temps d'exécution de M : soit m le nombre de nœuds dans G.

— Les étapes ① et ③ parcourent l'entrée une fois en temps $O(m)$.

— L'étape ② s'exécute au plus m fois (temps maximum de propagation entre s et t). Chaque exécution a au plus $O(m^2)$ étapes (chaque nœud est lié avec au plus $m - 1$ autre nœud), et s'exécute en temps $O(m^3)$.

— $m = O(n)$ où n = longueur de la chaîne d'entrée.

Dont M s'exécute en temps $O(n + n^3 + n) = O(n^3)$.

Donc, PATH $\in$ TIME$(n^3) \subset$ **P**.

Démonstration: RELPRIME $\in$ **P**

Montrons qu'il existe un décideur M à temps polynomial pour RELPRIME.

On utilise l'algorithme d'Euclide.

$M(\langle x, y\rangle)$ **=** ① si $x < y$ alors échanger x et y.
② répéter jusqu'à ce que $y = 0$
$x = x \mod y$.
échanger x et y.
③ si $x = 1$ alors accepter sinon rejeter.

Temps d'exécution de M :

— Les étapes ① et ③ sont exécutées une seule fois.

— Chaque exécution de l'étape ②, réduit la valeur de x par aux moins 2. Donc le nombre d'exécutions est au plus de $O(z)$ où $z = \log_2 x + \log_2 y$.

— Toute opération arithmétique (comparaison, modulo) peut être exécutée en temps polynomial du codage des entrées, donc polynomial en z.

Au total, M est polynomial en z.

Comme $z = O(n)$, RELPRIME $\in$ TIME$(n) \subset$ **P**. □

5 Classe NP (temps non-polynomial)

5.1 Vérificateur et certificat

Avant de pouvoir parler de la classe **NP**, nous devons aborder la notion de vérificateur.

Définition 27 : vérificateur

Un **vérificateur** V pour un langage A est un algorithme tel que :

$A = \{w \mid \exists c; V(\langle w, c\rangle) \text{ accepte}\}$

Remarques:

- V utilise une information c supplémentaire pour vérifier que $w \in A$.
  c s'appelle un certificat (ou preuve) d'appartenance à A.
- le temps d'exécution du vérificateur se mesure en terme de longueur de w
  i.e. la longueur de c ne compte pas dans la longueur de l'entrée.

Définition 28 : vérification

— un **vérificateur à temps polynomial** est un vérificateur qui s'exécute en temps polynomial sur la longueur de w.

— un langage A est un **langage polynomialement vérifiable** si il possède un vérificateur à temps polynomial.

Exemple 13: de vérificateur

Considérons le problème suivant :
SUBSET-SUM$\langle S, t\rangle$
Soit $S = \{s_1, s_2, \ldots, s_n\}$ un ensemble d'entiers et t un entier.
Existe-t-il un sous-ensemble de S tel que la somme de ses entiers soit égal à t ?
Exemples de certificats pour $S = \{5, 3, 4, 2, 1, 6\}$ et $t = 13$
$\{6, 4, 3\}$, $\{5, 4, 3, 1\}$, ...
ce sont toutes des valeurs possibles pour c.

Reformulation :

un vérificateur est donc un algorithme qui :

— $\forall w \in A$, il existe (au moins) un certificat c tel que $V(w, c)$ accepte.

— $\forall w \notin A$, il n'existe aucun certificat c tel que $V(w, c)$ accepte.

On peut considérer c comme la solution au problème w,

V est une façon de vérifier que cette solution est valide.

5.2 Définition

Définition 29 : Classe NP

La classe **NP** est la classe des langages qui sont polynomialement vérifiables.

Remarques:

- ne provient pas de la complexité de vérification qu'une solution est acceptable.
- mais bien du problème lui-même.

Théorème 47 : caractérisation de la classe NP

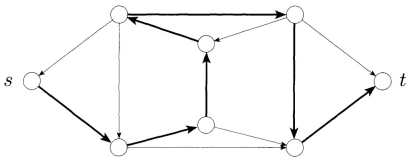
Un langage A est dans **NP** si et seulement si il est décidé par une MT non-déterministe en temps polynomial.

Autrement dit s'il existe un vérificateur à temps polynomial qui décide A.
Avant la démonstration, un exemple.

5.3 Exemples

Exemple 14: Graphe Hamiltonien

Un chemin Hamiltonien dans un graphe orienté est un chemin orienté qui passe exactement une seule fois par tous les sommets :
HAMPATH = $\{\langle G, s, t\rangle \mid G$ graphe orienté avec un chemin Hamiltonien de s à $t\}$



— HAMPATH est polynomialement vérifiable : si je connais un chemin s dans G, je peux facilement vérifier s'il est Hamiltonien.
— complexité de résolution par force brute : $n! = O(n^n)$ (test de tous les chemins possibles).
— pas d'algorithme connu résolvant HAMPATH en temps polynomial.

Définition 30 : Langage "graphe non Hamiltonien"
HAMPATH = graphes sans chemin Hamiltonien.
non polynomialement vérifiable : impossible de valider sans tout vérifier.

Théorème 48 : HAMPATH $\in$ **NP**.

Démonstration:
Définissons un vérificateur V à temps polynomial. Il faut montrer que pour tout $\langle G\rangle \in$ HAMPATH, il existe un certificat c tel que :

HAMPATH = $\{\langle G\rangle \mid V$ accepte $\langle G, c\rangle\}$.

Définissons un vérificateur V comme suit :

$V(\langle G, c\rangle)$ = si c est un chemin Hamiltonien dans G alors accepter
sinon rejeter

V s'exécute en temps polynomial (vérifier que chaque transition du chemin est dans la liste des arêtes, puis vérifier que chaque sommet n'est traversé qu'une seule fois).

Soit H = $\{\langle G\rangle \mid V(\langle G, c\rangle)$ accepte$\}$.
— $\forall\langle G\rangle \in$ H, il existe c qui accepte $\langle G, c\rangle$. Ceci implique que $\langle G\rangle$ est un graphe Hamiltonien. Donc, H $\subseteq$ HAMPATH.
— Pour tout $\langle G\rangle \in$ HAMPATH, soit c le cycle Hamiltonien dans ce graphe, alors V accepte $\langle G, c\rangle$. Donc, HAMPATH $\subseteq$ H.
Donc, HAMPATH $\in$ **NP**. □

Définition 31 : Langage "composite"
Un entier est composite s'il est le produit de deux entiers plus grands que 1 :
On définit le langage :

COMPOSITES = $\{x \mid x = pq,$ pour deux entiers $p, q > 1\}$

Théorème 49 : COMPOSITES $\in$ **NP**.

Démonstration:
Soit le vérificateur V défini par $V(\langle x, c\rangle)$ = si (c n'est pas 1 ou x) et (c divise x) alors accepter sinon rejeter.

V s'exécute en temps polynomial $O(< x >)$.

Soit C = $\{\langle x\rangle \mid V(\langle x, c\rangle)$ accepte$\}$.
— $\forall\langle x\rangle \in$ C, il existe c tel que V accepte $\langle x, c\rangle$. Donc, x est un nombre composite, et C $\subseteq$ COMPOSITES
— $\forall\langle x\rangle \in$ COMPOSITES, soit c un diviseur de x avec $1 < c < x$. Alors V accepte $\langle x, c\rangle$, COMPOSITES $\subseteq$ C.
Donc, COMPOSITES $\in$ **NP**. □

Démonstration:

($A \in$ **NP** $\Leftrightarrow$ A décidé par une MTND en temps polynomial).

$\Rightarrow$ Si A est dans **NP**, alors il existe un vérificateur V à temps polynomial qui permet de le vérifier.
Soit n^k le temps d'exécution de V.
Soit N, la MT non-déterministe suivante :

$N(\langle w \rangle) =$ ① choisir une chaîne c de longueur au plus n^k
② exécuter $V(\langle w, c \rangle)$
③ si V accepte alors accepter sinon rejeter.

Alors, sur la MTND N :
① générer l'ensemble des chaînes de longueur n^k se fait en temps polynomial $O(n^k)$ (arbre avec $|\Sigma|$ fils par nœud et de profondeur n^k). On obtient ainsi tous les certificats possibles.
② vérifier chaque certificat avec $V(\langle w, c \rangle)$ prend lui-aussi un temps polynomial (revient à poursuivre chaque feuille de l'arbre précédent et l'accepter ou la rejeter).
③ acceptation des branches. $O(1)$ sur une MTND.
V accepte si l'une quelconque des branches accepte. Donc, si un certificat c existe pour w, alors N le trouve nécessairement.
N permet donc de décider pour tout w de A en temps polynomial. $\Rightarrow \square$

$\Leftarrow$ Soit A un langage accepté par une MTND N à temps polynomial. On construit un vérificateur à temps polynomial de la façon suivante :

$V(\langle w, c \rangle) =$ ① simuler $N(\langle w \rangle)$, en utilisant c comme la description du choix non-déterministe de N à chaque étape.
② si la branche d'exécution de N accepte, alors accepter sinon rejeter

Si un MTND est à temps polynomial, alors chacune de ses branches s'exécute en temps polynomial. Or, le certificat c pris est celui qui permet de choisir l'une des branches acceptante à chaque nœud de la MTND. Donc, l'exécution de V ne revient à évaluer qu'une seule branche de la MTND. Par conséquent, V s'exécute en temps polynomial. $\Leftarrow \square$

5.4 Caractérisation de la classe NP

Ce théorème permet donc de donner une caractérisation équivalente de la classe **NP**

Définition 32 : `NTIME`($t(n)$)

`NTIME`($t(n)$) = ensemble des langages qui peuvent être décidés par une MT non-déterministe à temps $O(t(n))$.

Corollaire 50 : **NP** $= \bigcup_k$ `NTIME`(n^k)

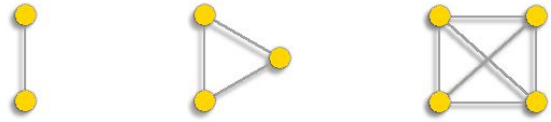
Démonstration: conséquence directe du théorème précédent. $\square$

Nous avons donc 2 façons de démontrer qu'un problème appartient à **NP** :
— en utilisant un vérificateur.
— en utilisant une MT non-déterministe à temps polynomial.

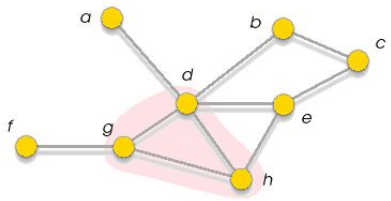
Maintenant, application pour montrer l'appartenance de nouveaux langages à **NP**.

Définition 33 : Langage Clique

Un graphe complet K_p est un graphe qui contient p noœuds, et tel que chaque noœud soit connecté au $p-1$ autres noœuds par une arête.



Une p-clique est un sous-graphe complet K_p d'un graphe G non orienté.



Le graphe ci-contre contient deux 3-cliques :
$\{g, h, d\}$ et $\{d, e, h\}$.

On définit le langage :

$\texttt{CLIQUE} = \{\langle G, k\rangle \mid G \text{ est un graphe avec une } k\text{-clique}\}$

5.5 CLIQUE

Théorème 51 : $\texttt{CLIQUE} \in \mathbf{NP}$.

Démonstration:

Méthode 1 : en utilisant un vérificateur.

Le certificat c du vérificateur V sont les nœuds constituant une k-clique.

$V(\langle G, k, c\rangle)$ = ① Tester si c est un sous-ensemble de k nœuds de G.
② Tester si chaque couple de nœuds de c est connecté.
③ Si les deux tests sont vrais, alors accepter sinon rejeter.

Trivialement, V accepte tout $\langle G, k\rangle \in \texttt{CLIQUE}$ en temps polynomial.

Donc, CLIQUE est polynomialement vérifiable et $\texttt{CLIQUE} \in \mathbf{NP}$. □

Méthode 2 : en utilisant une MT non-déterministe à temps polynomial qui accepte CLIQUE.

La MTND N suivante permet de décider CLIQUE :

$N(\langle G, k\rangle)$ = ① choix (non-déterministe) de c contenant k-nœuds de G.
② tester si chaque couple de nœuds de c est connecté.
③ si le test est vrai, alors accepter, sinon refuser.

Toutes les étapes de N s'exécutent en temps polynomial.

Donc, $\texttt{CLIQUE} \in \mathbf{NP}$. □

5.6 SUBSET-SUM

Théorème 52 : $\texttt{SUBSET-SUM} \in \mathbf{NP}$.

Démonstration:

Méthode 1 : en utilisant un vérificateur

Le certificat c du vérificateur V sont des entiers de S dont la somme est t.

$V(\langle S, t, c\rangle)$ = ① Tester si c est un sous-ensemble d'entiers S.
② Tester si la somme des nombres de c fait t.
③ Si les deux tests sont vrais, alors accepter, sinon rejeter.

Trivialement, V accepte tout $\langle S, t\rangle \in \texttt{SUBSET-SUM}$ en temps polynomial.

Donc, SUBSET-SUM est polynomialement vérifiable et $\texttt{SUBSET-SUM} \in \mathbf{NP}$. □

Méthode 2 : en utilisant une MT non-déterministe à temps polynomial qui accepte SUBSET-SUM.

La MTND N suivante permet de décider SUBSET-SUM :

$N(\langle S, t\rangle)$ = ① choix (non-déterministe) d'un sous-ensemble c de S.
② tester si la somme des nombres de c fait t.
③ si le test est vrai, alors accepter, sinon refuser.

Toutes les étapes de N s'exécutent en temps polynomial.

Donc, $\texttt{SUBSET-SUM} \in \mathbf{NP}$. □

6 NP-complétude

Comparaison entre P et NP

P classe des algorithmes qui peuvent être décidés "rapidement".

NP classe des algorithmes qui peuvent être vérifiés "rapidement".

où "rapidement" = en temps polynomial.

Réflexions :

La puissance de la vérifiabilité à temps polynomial
(i.e. d'une MT non-déterministe à temps polynomial)
semble plus importante que la décidabilité à temps polynomial
(i.e. d'une MT déterministe à temps polynomial)
Autrement dit, que **P** $\subset$ **NP**
Mais ...

Malheureusement,

- personne n'a jamais réussi à démontrer si **P** = **NP**
  un des grands problèmes d'informatique/mathématiques
- beaucoup pensent que **P** $\neq$ **NP**
  comme cela n'a jamais été démontré, cela est peut-être faux.

Ce que l'on peut dire avec certitude : **P** $\subseteq$ **NP** $\subseteq$ `EXPTIME` où `EXPTIME` = $\bigcup_k$ `TIME`(2^{n^k})
En 1971, Cook & Levin démontrent séparément que :

- Il existe des problèmes caractéristiques de la classe **NP** (dit problèmes **NP**-complets).
- Pour un problème **NP**-complets,
  si **n'importe lequel** d'entre eux est décidable par un MT déterministe en temps polynomial, alors **tous** les problèmes de **NP** peuvent être décidés par une MT déterministe en temps polynomial.
- **Reformulation** :
  il existe certains langages L tels que si $L \in$ **P**, alors **P** = **NP**.

et ils trouvent un exemple de problème **NP**-complet.

6.1 Réduction en temps polynomial

Rappel

Si un problème A peut être réduit en un autre problème B, alors :

- si B est décidable, alors A est décidable.
- si B est énumérable, alors A est énumérable.

On remarquera qu'une réduction est une transformation en un **nombre fini de pas**.

Définition 34 : fonction calculable en temps polynomial
Une fonction $f : \Sigma^* \to \Sigma^*$ est une **fonction calculable en temps polynomial** s'il existe une MT M à temps polynomial qui s'arrête avec $f(w)$ sur sa bande lorsque son entrée est w.

Définition 35 : langage réductible en temps polynomial
Un langage A est **réductible en temps polynomial** en un langage B, s'il existe une fonction f calculable en temps polynomial telle que : $w \in A \Leftrightarrow f(w) \in B$

Notation : $A \leq_p B$

On appelle f une réduction en temps polynomial de A à B

Théorème 53 : Réduction en temps polynomial à P
Soit $A \leq_P B$ et $B \in \mathbf{P}$.
Alors $A \in \mathbf{P}$.

Démonstration:
Soit un algorithme M qui décide B en temps polynomial.
Soit f une réduction en temps polynomial de A vers B.
Soit N l'algorithme suivant :

$N(\langle w \rangle)$ **=** calculer $f(w)$.
exécuter $M(< f(w) >)$
renvoyer la sortie de M

N décide B car M accepte $f(w)$ pour tout $w \in A$, puisque f est une réduction de A dans B et M un décideur pour A.
N s'exécute en temps polynomial puisque la réduction f et le décideur M s'exécutent consécutivement, chacun en temps polynomial. □

Maintenant, un exemple de réduction en temps polynomial.

6.2 Rappel de logique des propositions

Tables de vérité : opérateurs logiques OU $\vee$, ET $\wedge$ et NON $\neg$

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

x	$\overline{x}$
0	1
1	0

Définition :

littéral = variable booléenne (1/vrai ou 0/faux).
clause = plusieurs littéraux connectés par $\vee$
exemple : $v_1 \vee \overline{v_2} \vee \overline{v_3} \vee v_4$
forme normale conjonctive = plusieurs clauses connectées par $\wedge$
exemple : $(v_1 \vee \overline{v_2}) \wedge (\overline{v_3} \vee v_4)$
note 1 : aussi nommée FNC.
note 2 : FNCk = FNC où chacune des clauses a k littéraux.

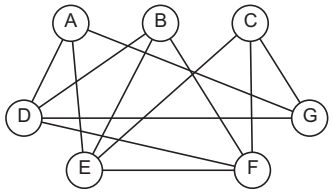
Forme normale conjonctive satisfiable :

Une FNC F est satisfiable s'il existe une combinaison de littéraux telle que F soit logiquement vrai.
exemple : $F = (v_1 \vee \overline{v_2}) \wedge (\overline{v_3} \vee v_4)$ est satisfiable ($F = 1$)
avec $v_1 = 1$, $v_2 = 1$, $v_3 = 0$ et $v_4 = 0$.

Graphe associé à une FNC :

— chaque littéral représente un nœud.
— chaque clause représente un ensemble de nœuds non connectés entre eux.
— chaque littéral x d'une clause est connecté à tous les littéraux de toutes les autres clauses, sauf si ce littéral est $\overline{x}$.

Exemple : $(p_1 \vee p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3}) \wedge (\overline{p_1} \vee p_2)$



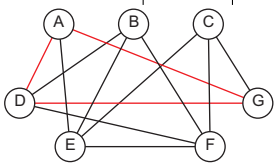
La FNC représente des "blocs" de relations similaires entre nœuds d'un graphe.

Lien entre FNC et clique

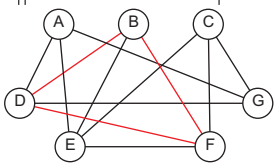
Un FNC à k clauses peut être transformé en un graphe. Si on trouve une k-cliques dans ce graphe, alors la FNC est satisfiable.

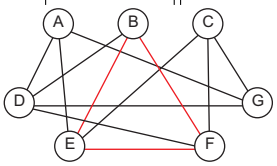
Exemple : $(p_1 \vee p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3}) \wedge (\overline{p_1} \vee p_2)$

p_1	p_2	p_3	$p_1 \vee p_2 \vee p_3$	$\overline{p_2} \vee \overline{p_3}$	$\overline{p_1} \vee p_2$	p
faux	faux	faux	faux	vrai	vrai	faux
faux	faux	vrai	vrai	vrai	vrai	vrai
faux	vrai	faux	vrai	vrai	vrai	vrai
faux	vrai	vrai	vrai	faux	vrai	faux
vrai	faux	faux	vrai	vrai	faux	faux
vrai	faux	vrai	vrai	vrai	faux	faux
vrai	vrai	faux	vrai	vrai	vrai	vrai
vrai	vrai	vrai	vrai	faux	vrai	faux

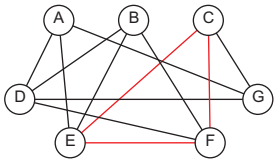


$p_1 = p_2 = 0, p_3 = 1$





$p_1 = 0, p_2 = 1, p_3 = 0$



$p_1 = p_2 = 1, p_3 = 0$

6.3 NP-complétude

Rappel

- $\texttt{SAT}_3 = \{\langle F \rangle \mid F$ est une $\texttt{FNC}_3$ satisfiable$\}$
- $\texttt{CLIQUE} = \{\langle G, q \rangle \mid G$ est un graphe avec une q-clique$\}$

Théorème 54 : $\texttt{SAT}_3 \leq_P \texttt{CLIQUE}$.

Démonstration:

Définissons la fonction f qui construit un graphe G associé à une FNC comme vu dans l'exemple précédent.

① Pour tout $F \in \texttt{SAT}_3$, alors $f(F) \in \texttt{CLIQUE}$, puisque pour une FNC à n clauses, $f(F) \in \texttt{CLIQUE}_n$ et que $\texttt{CLIQUE}_n \subset \texttt{CLIQUE}$.

② Inversement, la construction associant un littéral par nœud, f est clairement inversible. Donc, $f(F) \in \texttt{CLIQUE} \Rightarrow F \in \texttt{SAT}_3$.

③ La construction du graphe se fait en temps polynomial (générer les sommets = $O(3n)$, générer les arêtes = $O(3n^2)$).

Donc f est bien une réduction de $\texttt{SAT}_3$ vers $\texttt{CLIQUE}$ calculable en temps polynomial. □

Conséquence :

- on a vu que : $\texttt{SAT}_3 \leq_P \texttt{CLIQUE}$
- et que si $A \leq_P B$ et $B \in \mathbf{P}$ alors $A \in \mathbf{P}$.

Donc, si $\texttt{CLIQUE} \in \mathbf{P}$ alors $\texttt{SAT}_3 \in \mathbf{P}$ aussi.

Mais peut-on dire quelque chose si $\texttt{CLIQUE} \in \mathbf{NP}$?

Définition 36 : langage NP-complet

Un langage est B est **NP**-complet si :

- $B \in \mathbf{NP}$
- $\forall A \in \mathbf{NP}$, $A \leq_P B$ (= B est **NP**-difficile)

Remarque 3 : Les langages **NP**-complets sont les langages les plus difficiles de **NP**.

Théorème 55 : Si B est **NP**-complet et $B \in \mathbf{P}$ alors $\mathbf{P} = \mathbf{NP}$.

Démonstration:
Si B est **NP**-complet, alors pour tout $A \in \mathbf{NP}$, il existe une réduction en temps polynomial vers B. Or si $B \in \mathbf{P}$, B est décidable en temps polynomial.
Décider A = réduire à B + décider B ; faisable en temps polynomial. □

Théorème 56 : Si B est **NP**-complet et $B \leq_P C$ et $C \in \mathbf{NP}$ alors $\mathbf{C}$ est **NP**-complet.

Démonstration:
si B est **NP**-complet, alors $\forall A \in \mathbf{NP}$, $A \leq_P B$. Or $B \leq_P C$ implique $A \leq_P C$ (les deux réductions consécutives sont effectuées en temps polynomial). Donc, si $C \in \mathbf{NP}$, alors C est **NP**-complet puisque $\forall A \in \mathbf{NP}$, $A \leq_P C$. □

Problème principal : nous n'avons pas de problème **NP**-complet.

Définition 37 : $\texttt{SAT} = \{\langle F \rangle \mid F \text{ est une expression logique satisfiable}\}$

Théorème 57 : $\texttt{SAT} \in \mathbf{NP}$.

Démonstration:
— **avec un vérificateur à temps polynomial** : prendre comme certificat c une chaîne contenant la valeur de vérité de chaque littéral unique de l'expression logique. L'évaluation de l'expression logique à partir des valeurs des littéraux se fait en temps polynomial $O(n)$. □
— **avec une MT non déterministe** : choix non déterministe de la valeur de vérité de chaque littéral unique, puis évaluation de l'expression logique. Chaque branche s'exécute trivialement en temps polynomial. □

6.4 Théorème de Cook-Levin

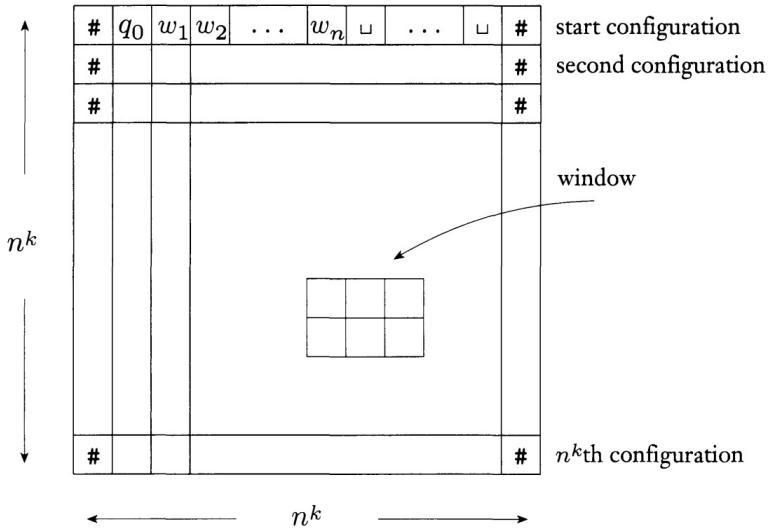
Théorème 58 (Cook-Levin): SAT est **NP**-complet.

Démonstration:
Il faut montrer que :
— $\texttt{SAT} \in \mathbf{NP}$ (fait : voir théorème précédent)
— $\forall A \in \mathbf{NP}$, $A \leq_P \texttt{SAT}$
Il faut montrer que tout langage de **NP** peut être réduit en temps polynomial en SAT.

Soit N une MT non déterministe qui décide le langage A.
On veut construire une réduction f telle que :

$\mathbf{N}$ accepte $w \Leftrightarrow F$ est satisfiable.

$N \in \mathbf{NP}$ signifie qu'une branche d'exécution de N s'exécute en temps n^k pour une entrée de taille n, donc :
— une branche d'exécution de N passe par n^k configuration différente
Rappel : configuration = bande + état courant à sa position.
— N visite au plus n^k cases : une configuration fait au plus n^k symboles.
On définit un tableau (voir page suivante) :
— de taille $n^k \times n^k$ qui représente une branche d'exécution de N sur w.
— qui utilise un alphabet étendu $C = \Gamma \cup Q \cup \{\#\}$ (alphabet de la bande, états et marqueur d'extrémité).
Note : devrait être $n^k + 3$ avec # au début et à la fin + état courant.



La branche d'exécution de N est stockée comme suit dans le tableau :
— La première ligne représente la configuration de depart.
— Les lignes suivantes représentent la suite des configurations dans l'ordre d'exécution de la branche. Chaque ligne peut être déduite de la précédente par les règles de transition de N.
— Si n'importe quelle ligne du tableau est une configuration acceptante, alors le tableau est acceptant.

Donc, un tableau accepte w si N accepte w. Ainsi,
— décider si N accepte w équivaut à décider s'il existe un tableau acceptant pour l'exécution de $N(\langle w \rangle)$.
— il faut trouver une formule logique F permettant de trouver si un tableau acceptant existe.

On veut construire une expression logique F qui permet de décider si un tableau est acceptant lorsque F est satisfiable. F doit garantir que toutes les conditions suivantes sont vraies :

1. Chaque cellule est occupée par exactement 1 symbole (ϕ_{cell}).
2. Le tableau est dans une configuration acceptante (ϕ_{accept}).
3. La première ligne est la configuration d'entrée avec w (ϕ_{start}).
4. La suite des configurations suivantes est cohérente avec une exécution de N (ϕ_{move}).

Autrement dit, F s'écrit sous la forme :

$$F = \phi_{\text{cell}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}}$$

On définit le littéral $x_{i,j,s}$ de la façon suivante.

$x_{i,j,s} = 1$ **ssi** la cellule (i, j) du tableau stocke le symbole $s \in C$ et 0 sinon.

Expression de ϕ_{cell} :

$f_{i,j,1}$ = la cellule (i, j) contient **au moins** un symbole = $\bigvee_{s \in C} x_{i,j,s}$

$f_{i,j,2}$ = la cellule (i, j) contient **au plus** un symbole = $\bigwedge_{s,t \in C, s \neq t} (\overline{x}_{i,j,s} \vee \overline{x}_{i,j,t})$

Ces conditions doivent être vérifiées pour toutes les cellules du tableau : $\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} f_{i,j,1} \wedge f_{i,j,2} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t \in C, s \neq t} (\overline{x}_{i,j,s} \vee \overline{x}_{i,j,t}) \right) \right]$.

Expression de ϕ_{accept} :

Si il y a l'état q_{accept} n'importe où dans la table.

$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}}$

Expression de ϕ_{start} :

Si la première ligne commence par #, suivi par q_0, puis $w = w_1 w_2 \dots w_n$, et complété par des blancs $\sqcup$, et fini par #.

$\phi_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,3,w_2} \wedge \dots \wedge x_{1,n+2,w_n}$
$\wedge x_{1,n+2,\sqcup} \wedge \dots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}$

Expression de ϕ_{move} :

Fenêtre : définie par sa position (i, j) et sa taille 2×3.

Fenêtre légale : transformation possible par N sur la fenêtre.

a	q_1	b	légale si N a une transition $\delta(q_1, b) = (q_2, c, L)$
q_2	a	c	
a	q_1	b	légale si N a une transition $\delta(q_1, b) = (q_2, a, R)$
a	a	q_2	
a	a	q_1	légale si N a une transition $\delta(q_1, c) = (q_2, b, R)$
a	a	b	
#	a	b	légale (pas de transition dans cette fenêtre)
#	a	b	
a	b	a	légale si N a une transition $\delta(q_1, b) = (q_2, c, L)$
a	b	q_2	
a	a	a	légale si N a une transition $\delta(q_1, a) = (q_2, b, L)$
b	a	a	
a	b	b	non légale (la transition devrait être dans la fenêtre)
a	a	b	
a	q_1	b	non légale (2 transitions sur la 2ème ligne)
q_2	a	q_2	
a	q_1	a	non légale (transition à gauche, caractère à droite modifié)
q_2	c	b	

Expression de ϕ_{move} :

Validité du tableau : si toutes les fenêtres sont légales.

Soit le prédicat Legal(i, j) qui est vrai si la fenêtre (i, j) est légale.

$$\text{Legal}(i, j) = \bigvee_{\{a_1,\dots,a_6\} \text{ est légale}} x_{i,j,a_1} \wedge x_{i,j,a_2} \wedge \dots \wedge x_{i,j,a_6}$$

Autrement dit, Legal(i, j) est l'union de toutes les configurations légales sur une fenêtre 3×2 en accord avec le fonctionnement de N sur deux configurations consécutives.

En conséquence, $\phi_{\text{move}} = \bigwedge_{1 \le i,j \le n^k} \text{Legal}(i, j)$

Note : raison pour laquelle l'application du prédicat ϕ_{move} produit une suite de configuration valide correspondant à l'exécution d'une MT :

- la première ligne du tableau est une ligne valide (entrée w de la MT assurée par ϕ_{start}).
- la fenêtre 3×2 permet de produire (par récurrence) une configuration valide (la suivante) à partir de la configuration précédente (valide pour i=1).

Conséquence de cette construction :

F satisfiable
$\Rightarrow$ F représente l'affectation d'un tableau acceptant.
$\Rightarrow$ le tableau acceptant représente une exécution de N sur w.
$\Rightarrow$ N accepte l'entrée w.
Inversement, si N accepte l'entrée w,
$\Rightarrow$ la suite des configurations se représente comme un tableau acceptant.
$\Rightarrow$ la formule F équivalente est satisfiable.
$\Rightarrow$ F acceptant.

Donc, pour tout MT N non déterministe à temps polynomial :
N accepte $w \Leftrightarrow F$ est satisfiable
Cette construction est donc une réduction de tout langage de **NP** vers SAT.

Montrons maintenant que cette réduction peut être faite en temps polynomial.
Notons tous d'abord que :
— l'alphabet étendu C utilisé pour coder le tableau (on notera $c = \#C$).
— la MT N
ne dépendent pas de n.
Alors, le temps de calcul est de l'ordre de :

ϕ_{cell} : $f_{i,j,1}$ et $f_{i,j,2}$ ne dépendent que de c. Donc ϕ_{cell} est en $O(n^{2k})$.
ϕ_{accept} : Trivialement de l'ordre de $O(n^{2k})$.
ϕ_{start} : Trivialement de l'ordre de $O(n^k)$.
ϕ_{move} : Legal(i, j) ne dépend que de c. Donc ϕ_{move} est en $O(n^{2k})$.

Par conséquent la réduction s'effectue en temps polynomial.
Donc, $\forall A \in \mathbf{NP}$, $A \leq_P$ SAT. □

6.5 FNC-SAT

Définition : FNC-SAT = $\{\langle F\rangle \mid F$ est une FNC satisfiable$\}$
Théorème : FNC-SAT est **NP**-complet.
Démonstration :

— FNC-SAT $\in$ **NP** Même ligne de preuve que pour SAT.
— $\forall A \in \mathbf{NP}$, $A \leq_P$ FNC-SAT. la preuve du théorème du Cook-Levin peut être directement réutilisée car la réduction utilise des FNCs.

Donc, FNC-SAT est **NP**-complet. □

Définition 38 : $\text{SAT}_3 = \{\langle F\rangle \mid F$ est une FNC_3 satisfiable$\}$

Théorème 59 : SAT_3 est **NP**-complet.

Démonstration:
— $\text{SAT}_3 \in \mathbf{NP}$. Même ligne de preuve que pour SAT.
— $\forall A \in \mathbf{NP}$, $A \leq_P \text{SAT}_3$. Pour se faire, on réduit FNC-SAT à SAT_3. Soit $F = \bigwedge_i C_i$ où C_i sont les clauses de F. On a alors 3 cas :

— C_i **a moins de 3 littéraux** : il suffit de dupliquer l'un des littéraux.
Exemple : $C_i = L_1 \Rightarrow C'_i = L_1 \vee L_1 \vee L_1$
— C_i **a 3 littéraux** : on le laisse tel quel.

— C_i **a plus de 3 littéraux** : notons $C_i = L_1 \vee L_2 \vee \ldots \vee L_m$.
Introduire des nouveaux littéraux z_i de la façon suivante :

$$C'_i = (L_1 \vee L_2 \vee z_1) \wedge (\overline{z}_1 \vee L_3 \vee z_2) \wedge (\overline{z}_2 \vee L_4 \vee z_3) \wedge \ldots \wedge (\overline{z}_{m-3} \vee L_{m-1} \vee L_m)$$

— C_i a plus de 3 littéraux (suite)
$C'_i = (L_1 \vee L_2 \vee z_1) \wedge (\overline{z}_1 \vee L_3 \vee z_2) \wedge (\overline{z}_2 \vee L_4 \vee z_3) \wedge \ldots \wedge (\overline{z}_{m-3} \vee L_{m-1} \vee L_m)$
Si C_i est satisfiable, alors il existe $(z_1, z_2, \ldots, z_{m-3})$ tels que C'_i soit aussi satisfiable.
Exemple :

$C'_i = (L_1 \vee L_2 \vee z_1) \wedge (\overline{z}_1 \vee L_3 \vee z_2) \wedge (\overline{z}_2 \vee L_4 \vee L_5)$
Si C_i n'est pas satisfiable, alors il n'existe aucune combinaison de (z_1, z_2) qui rende C'_i satisfiable, sinon il existe une combinaison de (z_1, z_2) qui rend C'_i satisfiable.

Soit $F \in$ FNC-SAT satisfiable. Soit F' construit à partir de F par la méthode indiquée. Alors $F' \in \mathtt{SAT}_3$ est satisfiable. Inverse évident car $\mathtt{SAT}_3 \subset$ FNC-SAT. Comme FNC-SAT $\leq_P \mathtt{SAT}_3$ et FNC-SAT **NP**-complet, alors $\mathtt{SAT}_3$ aussi. □

6.6 CLIQUE

Rappel : CLIQUE $= \{\langle G, k\rangle \mid G$ est une $k -$ CLIQUE $\}$

Théorème 60 : CLIQUE est **NP**-complet.

Démonstration:
— CLIQUE $\in$ **NP** : déjà démontré.
— $\forall A \in$ **NP**, $A \leq_P$ CLIQUE. On a déjà montré que $\mathtt{SAT}_3 \leq_P$ CLIQUE. Comme $\mathtt{SAT}_3$ **NP**-complet, alors CLIQUE aussi. □

On voit que l'on dispose d'une méthode facile pour montrer qu'un problème K est **NP**-complet.
— trouver un autre problème K' qui soit **NP**-complet et "proche" de K
— montrer que $K \in$ **NP**.
— montrer qu'il existe une réduction en temps polynomial de K' vers K.
— en déduire que K est **NP**-complet.

6.7 SUBSET-SUM

Rappel : SUBSET-SUM $= \{\langle S, t\rangle \mid \exists S' \subseteq S = \{s_1, s_2, \ldots, s_n\} / \Sigma_{s_i \in S'} s_i = t\}$

Théorème 61 : SUBSET-SUM est **NP**-complet.

Démonstration:
— SUBSET-SUM $\in$ **NP** : déjà démontré.
— Par réduction en temps polynomial de $\mathtt{SAT}_3$ vers SUBSET-SUM.
Soit $F = \bigwedge_{i=1\ldots n} C_i$ une expression logique sous forme de conjonction de clauses C_i, chaque clause étant composée de la disjonction de 3 littéraux parmi $x_1, x_2, \ldots, x_p$ ou de leurs négations.
On cherche une transformation telle que :
— chaque littéral x_j possède une valeur unique (vrai ou faux).
— chaque clause C_i soit vraie.
Construisons un ensemble d'entiers à $p + n$ chiffres où l'on affecte une propriété à chaque chiffre.

Le rôle de chaque chiffre est le suivant :
— les p premiers chiffres représentent les littéraux x_j. Le $j^{\text{ème}}$ chiffre représente le littéral x_j.
— les n chiffres suivants représentent les clauses C_i. Le $(p + i)^{\text{ème}}$ chiffre représente la clause C_i.

On va donc créer un premier ensemble de $2p$ entiers à $n+p$ chiffres, où tous leurs chiffres sont à 0, sauf pour :

a_j (associé à x_j) son $j^{\text{ème}}$ chiffre à 1 si x_j est **vrai**.
et partout où x_j apparaît dans C_i, son $(p+i)^{\text{ème}}$ chiffre est à 1.

$\overline{a}_j$ (associé à $\overline{x}_j$) son $j^{\text{ème}}$ chiffre à 1 si x_j est **faux**.
et partout où $\overline{x}_j$ apparaît dans C_i, son $(p+i)^{\text{ème}}$ chiffre est à 1.

Quelle valeur de t faut-il alors choisir ?

Il faut nécessairement que chaque a_j ou (exclusif) $\overline{a}_j$ soit dans S'.

$\Rightarrow$ les n premiers chiffres de t sont 1.

$\Rightarrow$ les p chiffres suivants de t sont supérieurs à 1.

Exemple : $F = C_1 \wedge C_2 = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3 \vee x_4)$

	x_1	x_2	x_3	x_4	C_1	C_2
a_1	1	0	0	0	1	0
$\overline{a}_1$	1	0	0	0	0	0
a_2	0	1	0	0	0	1
$\overline{a}_2$	0	1	0	0	1	0
a_3	0	0	1	0	1	0
$\overline{a}_3$	0	0	1	0	0	1
a_4	0	0	0	1	0	1
$\overline{a}_4$	0	0	0	1	0	0

Exemples de choix :

$a_1 + \overline{a}_2 + a_3 + a_4 =$	1	1	1	1	3	1	valide / F vrai
$a_1 + a_2 + \overline{a}_3 + \overline{a}_4 =$	1	1	1	1	1	2	valide / F vrai
$\overline{a}_1 + a_2 + \overline{a}_3 + a_4 =$	1	1	1	1	0	3	valide / F faux
$a_1 + \overline{a_1} + a_3 =$	2	0	1	0	2	0	invalide / F indéfini

Pour la partie attribut : le n premiers chiffres sont 1.

Pour la partie clause : Chaque chiffre est entre 1 et 3.
Pas de t unique si on fait comme cela.

Solution : Choisir $t = 1 \ldots 13 \ldots 3$ en ajoutant 2 variables identiques supplémentaires par clause b_j et b'_j (donc $2p$ variables) définies par le $(n+j)^{\text{ème}}$ chiffre est à 1 et les autres à 0.

Il y a alors 3 cas différents pour la $(n+j)^{\text{ème}}$ colonne :
$= 3$ pas de problème (les 3 littéraux sont vrais).
$= 2$ (resp. 1), alors ajouter b_j **ou** b'_j (resp. b_j **et** b'_j) pour arriver à 3.
$= 0$ alors même en ajoutant b_j et b'_j, impossible de faire 3.
Donc, avec $S = \{a_1,\ldots,a_n,\overline{a}_1,\ldots,\overline{a}_n,b_1,\ldots,b_p,b'_1,\ldots,b'_p\}$ et $t = \underbrace{1\ldots1}_{n \text{ fois}}\underbrace{3\ldots3}_{p \text{ fois}}$.

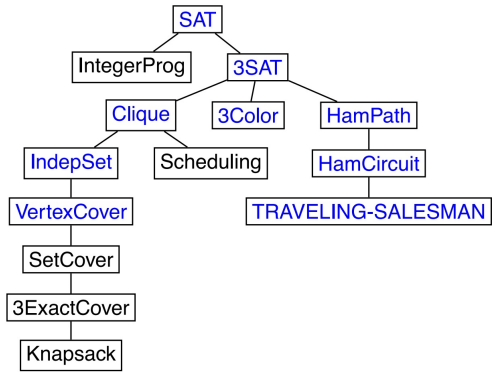
On a donc trivialement ($\langle F\rangle \in \texttt{SAT}_3 \Leftrightarrow f(\langle F\rangle) \in \texttt{SUBSET-SUM}$).

La réduction f s'effectue clairement en temps polynomial. En conséquence, $\texttt{SAT}_3 \leq_P \texttt{SUBSET-SUM}$ et $\texttt{SAT}_3$ **NP**-complet implique SUBSET-SUM **NP**-complet. □

7 Hiérarchies des classifications

7.1 Hiérarchies des problèmes NP-complet

De nombreux problèmes **NP**-complets ont été trouvés à ce jour.
Ils constituent une hiérarchie formée par les réductions en temps polynomial pour passer de l'un à l'autre.



7.2 Hiérarchies des classes de complexité algorithmique

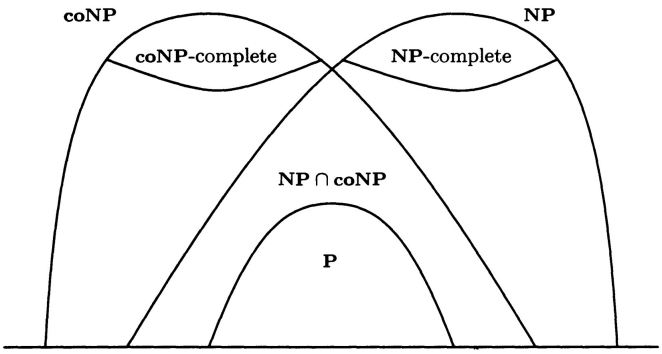
Définition 39 : *co*NP et *co*NP-complétude

- **ensemble *co*NP** : un langage C appartient à ***co*NP** si $\overline{C} \in$ **NP**.
- **langage *co*NP**-complet : un langage C est ***co*NP**-complet si :
  - $\overline{C} \in$ **NP**.
  - $\forall D \in$ ***co*NP**, $D \leq_p C$.

Ces nouveaux ensembles conduisent à la définition du monde de la complexité algorithmique tel qu'on le pense actuellement, c'est à dire sous les hypothèses que :

- **P** $\neq$ **NP**
- **NP** $\neq$ ***co*NP**

On obtient alors le diagramme suivant.



8 Conclusion

On a vu :

- les différents modèles d'ordinateurs : du plus simple au plus complexe.
- les limitations des différents modèles.

Ce que vous devez en retenir : **un ordinateur ne peut pas tout faire**.

- certains problèmes ne peuvent pas être résolus dans le cas général
  ce qui nécessite se placer sur une sous-partie décidable (calculable) du problème.

— certains problèmes ont une complexité algorithmique trop forte pour être résolu exactement. ce qui nécessite d'accepter d'avoir des solutions sous-optimales, ou que le problème ne soit pas soluble.

Il nous allons maintenant aborder la complexité spatiale (étude des limitations d'algorithmes en raison de la complexité de stockage).