TABLE DE MATIERES

LISTE DES TABLEAUX ET FIGURES

LISTE DES FIGURES

LISTES DES TABLEAUX

OBJECTIF PEDAGOGIQUE

L'objectif de ce chapitre c'est d'être plus précis que le paysage étudié en calculabilité en considérant dorénavant que des problèmes clairement décidables et en se posant la question de savoir, si l'on peut les résoudre efficacement ?

Il s'agira donc de distinguer ce qui est raisonnable de ce qui ne l'est pas en temps de calcul.

INTRODUCTION

En théorie de la complexité, un **problème NP-complet** est un problème de décision vérifiant les propriétés suivantes : Il n'a pas encore été trouvé une solution pour le résoudre en temps polynomial dans le pire des cas, Tous les problèmes de la classe NP se ramènent à celui-ci via une réduction polynomiale. Cela signifie que le problème est au moins aussi difficile que tous les autres problèmes de la classe NP.

Bien qu'on puisse vérifier rapidement toute solution proposée d'un problème NP-complet, on ne sait pas en trouver une efficacement. C'est le cas, par exemple, du problème SAT ou de celui du coloriage de graphe. Tous les algorithmes connus pour résoudre des problèmes NP-complets ont un temps d'exécution exponentiel à la taille des données d'entrée dans le pire des cas, et sont donc inexploitables en pratique même pour des instances de taille modérée. La seconde propriété de la définition implique que s'il existe un algorithme polynomial pour résoudre un quelconque des problèmes NP-complets, alors tous les problèmes de la classe NP peuvent être résolus en temps polynomial. Trouver un algorithme polynomial pour un problème NP-complet ou prouver qu'il n'en existe pas permettrait de savoir si P = NP ou P ≠ NP, une question ouverte qui fait partie des problèmes non résolus en mathématiques les plus importants à ce jour.

Dans ce Chapitre, nous allons nous intéresser aux classes P et NP, sont-elles différentes ? Nous allons le découvrir en présentant d'abord la notion de décidabilité et machine de Turing en I, la classe P en II, ensuite la classe NP en III la notion de NP-COMPLETUDE en VI et quelques exemples de problèmes NP-complets en V.

1. GENERALITES

1.1 Décidabilité

La théorie de décidabilité donne un cadre pour déterminer si un problème mathématique peut être résolu par un ordinateur.

Le cas échéant, la théorie de la complexité donne un cadre pour déterminer si ce problème peut être résolu par un algorithme efficace (qui n'utilise qu'une quantité raisonnable de mémoire et qui s'exécute en un temps raisonnable).

Un problème de décision est une question mathématique dont la réponse est <<**oui**>> ou <<**non**>>**.** (Vraie ou faux)

Exemple :

a) Le problème de primalité prime qui permet de savoir pour un nombre donné n s'il est premier ou pas.
b) Le problème du circuit **hamiltonien** qui permet de savoir si pour un graphe G on peut trouver un circuit passant une fois et une seule par tous les sommets.

Les objets mathématiques sur lesquels porte le problème sont des instances du problème. Pour le problème de primalité une instance c'est un nombre entier, pour celui du circuit hamiltonien c'est un graphe.

On cherche à savoir s'il existe une procédure algorithmique qui permette de calculer pour chaque instance si la réponse est oui ou non.

Un problème de décision est **décidable** s'il existe une machine de Turing déterministe qui accepte les instances ou la réponse est oui.

1.2 Machine de Turing

1.2.1 Machine de Turing déterministe

Une machine de Turing déterministe est la donnée :

- D'un ruban infini : chaque case contient un élément d'un alphabet fini R qui contient un caractère <<Blank>>
- D'une tête de lecture : celle est dans un état à choisir dans un ensemble fini E.il existe un état initial et des états finaux.
- D'une fonction de transition, qui à partir de l'état de la tête de lecture et du contenu de la case lue donne le nouvel état de la tête, le nouveau contenu de la case et déplace la tête de lecture d'une case vers la droite ou la gauche. Les données de départ sont inscrites sur le ruban et la tête de lecture est dans son état initial.

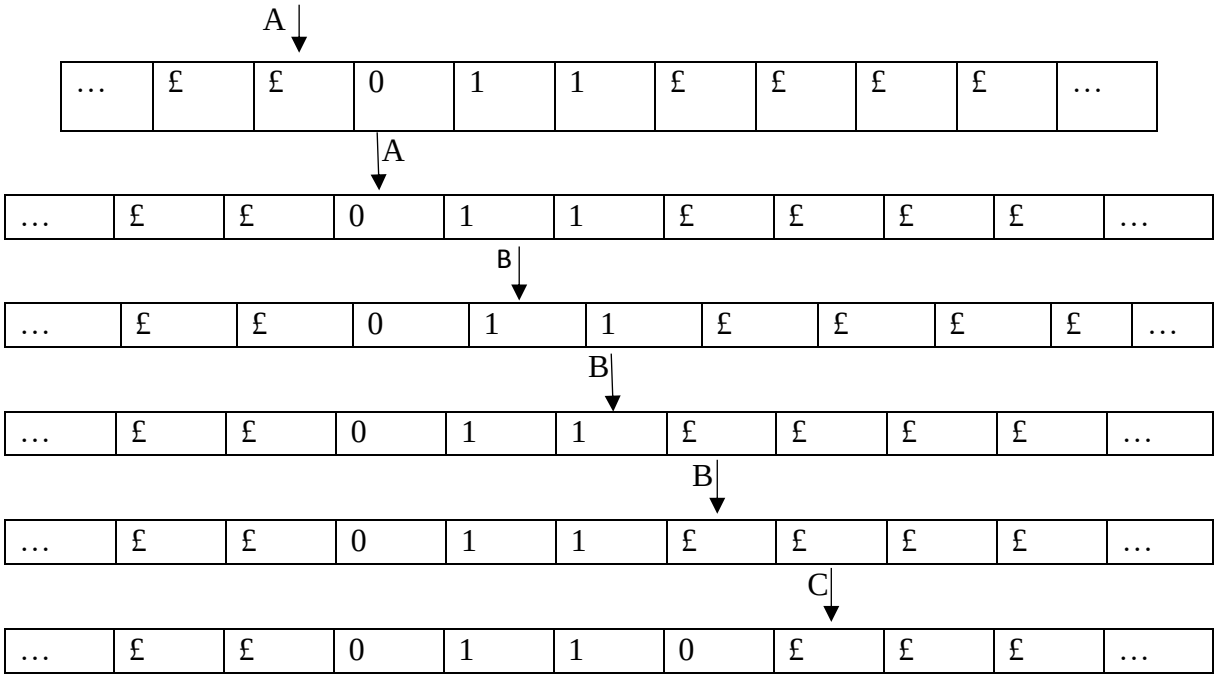
Exemple : Ruban 0 et 1 ; 3 états A (initiale), B et C (final)

Table de transition :

Tableau 1: table de transition

	0	1	£
A	B, 0, →	B, 1, →	A, £, →
B	B, 0, →	B, 1, →	C, 0, →

Tableau 2:symbole lu en tête de lecture



NB : cette machine effectue une multiplication par 2 en supposant que c'était un nombre binaire.

Soit T une machine de Turing déterministe, le mot d'entrée x est représenté par le mot (fini) écrit sur le ruban au départ de l'exécution de la machine.

- Si la machine atteint un état final après un nombre fini d'étapes, on dit que le mot x est accepté et le mot inscrit sur le ruban après l'arrêt s'appelle **mot de sortie** et est noté T(x).
- Si la machine s'arrête sans atteindre un état final ou si la machine ne s'arrête jamais, alors x est rejeté.

1.2.2 Machine non déterministe

Le concept de machine de Turing non-déterministe est une variante de la notion de machines de Turing classique déterministe :

- La définition d'une machine de Turing non-déterministe est exactement comme celle de la notion de machine de Turing (déterministe) sauf sur un point :
  - Pour tout état et une lettre lue en face de la tête de lecture donnée, δ ne définit pas un seul triplet de Q x Γ x {←, →}, mais un ensemble de triplet
  - Intuitivement lors d'une exécution la machine a la possibilité de choisir n'importe quel triplet.

La différence est qu'une machine de Turing non-déterministe n'a pas une exécution unique sur une entrée w, mais éventuellement plusieurs

En fait, les exécutions de la machine sur un mot w donnent lieu à un arbre de possibilités (arbre de dérivation), et l'idée est qu'on accepte un mot si l'une des branches de cet arbre contient une configuration acceptante.

Exemple :

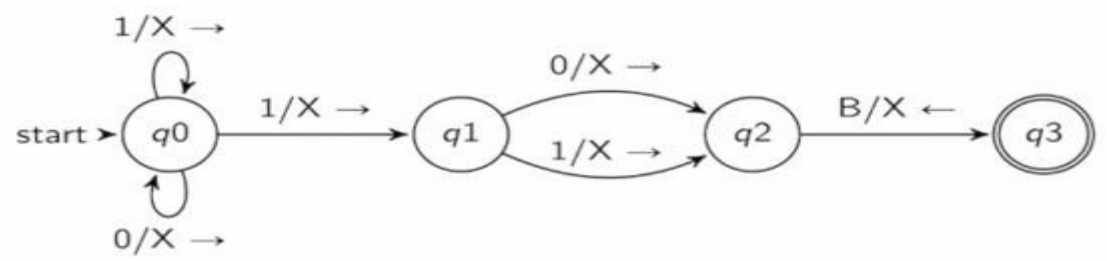


Figure 1: exemple de machine de Turing non déterministe

- **Exécutions sur le mot w = 010111 :**

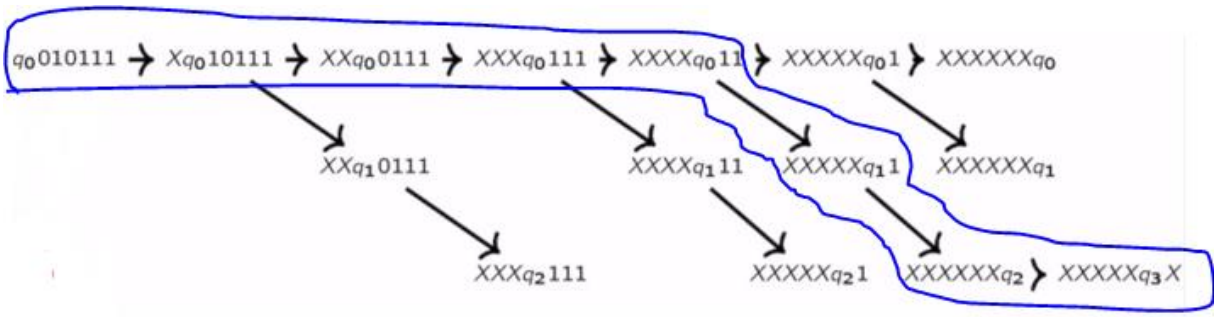


Figure 2: chemin acceptant

On dit que la machine accepte le mot w en Temps T= 7

Conclusion : On dit donc qu'un mot est accepté par une machine de Turing non-déterministe s'il y'a une exécution sur ce mot partant de l'état initial qui se termine en l'état acceptant.

2. LA CLASSE P

La théorie de la complexité cherche à classer les problèmes de décision (ceux qui ont une réponse par oui ou par non), selon la complexité des algorithmes capables de les résoudre. La première classe fondamentale est la **classe P**, formée des problèmes de décision qui peuvent être résolus par un algorithme polynomial.

> La complexité d'un algorithme est dite polynomiale si elle est en O (n^k) dans la pire des cas. K étant un entier et n la taille de l'instance.

Il existe plusieurs raisons de s'intéresser à cette classe d'algorithmes :

- D'abord les algorithmes efficaces sont polynomiaux, autrement dit les algorithmes non polynomiaux sont certainement inefficaces.
- En second lieu, cette notion est robuste, c'est-à-dire indépendante de la technologie. Outre le nombre d'instructions effectuées par un processeur, on peut définir le temps de calcul comme le nombre de transitions effectuées par une machine de Turing avant de s'arrêter.
- Enfin les algorithmes polynomiaux forment une classe stable : la composition de deux algorithmes polynomiaux reste polynomiale, et construire un algorithme polynomial à partir d'appels à des procédures de complexité polynomiale, reste polynomial.

2.1 Définition :

La classe P est la classe des problèmes pouvant être résolus en temps polynomial. C'est encore dit que la classe P est la classe des problèmes faciles.

La classe P est l'ensemble des problèmes de décision (vu comme un langage) décidé par une machine de Turing (ou un algorithme) déterministe en temps polynomial en la taille de l'entrée.

2.2 Exemple de problème dans P :

Exemple1 : Le problème BONCOLORIAGE

Donnée : Une formule $F(x_1, x_2, \ldots . x_n)$ du calcul propositionnel en forme normale conjonctive, des valeurs $x_1, x_2, \ldots . x_n \in \{0,1\}$ pour chacune des variables de la formule

Réponse : Décider si la formule F s'évalue à vrai pour ces valeurs de variables

Considérons pour illustrer l'exemple l'algorithme ci-dessous écrit en en pseudo français

- **Algorithme :**

Résultat = vrai ;

Pour chaque arête (u, v)

Si couleur (u) = couleur (v) alors Résultat = faux ;

Retourner Résultat ;

- **Calculons sa complexité :**

Si l'on veut mesurer la complexité de cet algorithme, on doit fixer une mesure élémentaire (le nombre d'instruction élémentaires) et on va supposer qu'accéder à une arête se fait en temps constant

Si m désigne le nombre d'arête, alors cet algorithme est de l'ordre de $\mathbf{O(m)}$ puisqu'on parcourt chaque arête (u, v) une fois

Et le nombre d'arêtes dans un graphe est $\mathbf{O(n^2)}$ car $\boldsymbol{m \leq \frac{n(n-1)}{2} \leq n^2}$

D'où complexité = $\mathbf{O(n^2)}$ ce qui prouve que c'est un algorithme polynomial en n^2

Conclusion : Ce problème est donc raisonnable et peut être résolue de manière efficace

En effet, On se donne donc par exemple la formule $\neg p \wedge r \vee \neg q$

Pour p= 0, r =1, et q = 0, ça se résous en temps polynomial il suffit de tester la formule pour ces valeurs de p, r et q

Donc BONCOLORIAGE ∈ P

Exemple2 : Le Problème 3-COLORABILITE

On se donne un graphe et on veut savoir s'il y'a une façon de le colorier qui utilise au plus 3 couleurs.

Une façon de faire c'est de tester toutes les façons d'affecter des couleurs à chaque sommet et il y en a autant de fois que de façon d'affecter 3-couleurs à n sommets.

Pour chacune je retourne vrai si c'est un bon coloriage sinon je retourne faux

- **Algorithme :**

résultat = faux ;

Pour les 3^n façons possibles d'affecter une couleur à chaque sommet

Tester si cela est un bon coloriage de G ;

Si oui alors résultat = vrai ;

Retourner résultat ;

- **Calcul de la complexité** :(en nombre d'instructions élémentaires)

On suppose qu'accéder à une arête se fait en temps constant

L'instruction : Tester si cela est un bon coloriage de G se fait en $O(n^2)$ et on la fait 3^n fois

Donc complexité = $O(3^n \times n^2) = O(3^{(1+\varepsilon)n})$ ou n est le nombre de sommets.

Sauf qu'en pratique cet algorithme est inutilisable car 3^n explose très vite d'où la nécessité d'avoir des complexités aussi basses que possible afin de réduire la complexité asymptotique des problèmes ce qui permettre d'avoir des algorithmes utilisables.

Conclusion : Cet algorithme est bien de complexité polynomiale mais il est difficile de dire si 3colorabilité ∈ P

3. LA CLASSE NP :

3.1 Définition :

C'est l'abréviation pour "nondeterministic polynomial time". Cette classe renferme tous les problèmes de décision dont on peut associer à chacun d'eux un ensemble de solutions potentielles (de cardinal au pire exponentiel) tel qu'on puisse vérifier en un temps polynomial si une solution potentielle satisfait la question posée. Le terme non déterministe désigne un pouvoir qu'on incorpore à un algorithme pour qu'il puisse deviner la bonne solution.

Les différentes exécutions d'un algorithme non déterministe, pour un argument u, sont les branches d'un arbre, autrement dit les chemins depuis la racine (départ de l'exécution) vers les feuilles (terminaison de l'exécution) ; chaque nœud interne de l'arbre correspond à un choix. Pour définir NP, on se restreint au cas où toutes les branches sont finies, c'est-à-dire où toutes les exécutions terminent ; restent deux questions :

- Que calcule un algorithme non déterministe ?
- Comment mesurer le temps de calcul d'un tel algorithme ?

La première question n'a guère de réponse satisfaisante en général, c'est pourquoi on entend peu parler d'algorithme non déterministe dans les cours de programmation. Il faut se restreindre aux problèmes de décision pour pouvoir définir une convention intéressante.

A partir de maintenant, on ne considère donc que les algorithmes non déterministes possédant la propriété suivante : chaque branche de l'arbre d'exécution, pour un argument u donné, calcule 0 ou 1 (on dit aussi que chaque exécution se termine par refuse ou accepte). On convient alors, comme pour les automates, que la réponse de l'algorithme est "oui" s'il existe une branche qui calcule 1 (autrement dit s'il existe une exécution qui accepte l'argument u) ; par conséquent la réponse est "non" si toutes les branches calculent 0 (si toutes les exécutions refusent l'argument u).

En ce qui concerne le temps de calcul, on convient, conformément au principe "envisager le pire", que c'est celui de la plus longue branche (la longueur d'une branche est mesurée en nombre d'instructions), qui calcule 0 ou 1.

Donc, NP = P non déterministe

Avec ces conventions, on a la définition fondamentale suivante :

La classe NP est formée des problèmes de décision qui peuvent être résolus par un algorithme Polynomial non déterministe.

- **Variante** : avec une autre terminologie, équivalente et fréquente dans ce contexte, cette définition devient :

La classe NP est formée des langages reconnaissables par une machine de Turing polynomiale non déterministe.

Tout algorithme non déterministe peut être simulé par un algorithme déterministe, qui parcourt récursivement l'arbre de tous les choix possibles ; les fonctions calculables dans un modèle le sont donc dans l'autre, le non-déterminisme n'apporte rien de ce point de vue.

Par contre la complexité d'un algorithme non déterministe est mesurée par le temps d'exécution de la plus longue branche c'est-à-dire : soit T une machine de Turing à calcul infini pour un mot initiale x ,le temps de calcul de T est défini par : **Dt(x)= MAX { m |il existe une exécution de T sur x qui s'arrête après m étapes }**, alors que la complexité de sa simulation déterministe est égale au temps d'exécution de l'arbre entier; lorsque la première est polynomiale, la seconde est en général exponentielle.

En fait, il existe beaucoup de problèmes dans NP pour lesquels aucun algorithme polynomial déterministe n'est connu, ce qui semble indiquer, avec une grande vraisemblance, que la classe NP est plus large que la classe P, d'où la nécessité de connaitre une base la plus large possible.

La réponse à la question : P = NP ? est donc très vraisemblablement négative, mais, aucune démonstration de ce résultat négatif n'a été trouvée en 30 ans de recherche très active, d'où la célébrité de ce problème, qui intrigue tous les théoriciens de l'informatique.

3.2 Certificats et vérifieurs (Vérificateurs) :

3.2.1 Définition :

Un **vérifieur** pour un problème A est une machine de Turing M déterministe telle que w est une instance positive de A si et seulement s'il existe c tel que M accepte (w, c). Un tel c est appelé **certificat**.

Il existe une variante de la définition de NP qui élimine le non-déterminisme, en codant la succession de choix non déterministes par un mot x ; en effet une branche d'un arbre peut être codée par une suite de (petits) entiers : par exemple 4,1,2… code la branche qui débute par le quatrième fils de la racine (appelons-le s), puis passe par le premier fils de s (appelons-le t), puis par le second fils de t, etc. Le mot x devient le second argument de l'algorithme ; cet algorithme déterministe à deux variables (le non-déterminisme se trouve en quelque sorte externalisé dans le second argument), est appelé vérifieur pour le problème de décision correspondant :

> Une procédure (déterministe) booléenne V à deux variables est un vérifieur pour le problème de décision A si et seulement si : A (u) ⇔ ∃x |V (u, x)

Un argument x qui satisfait le prédicat V (u, x) est appelé un certificat de u (en clair, le certificat est une interprétation de u) ; d'autre part, un vérifieur est dit polynomial si sa complexité est O (n^k), où n désigne la taille du premier argument u. Cette nouvelle définition de NP est celle qu'on utilise presque toujours en pratique. Elle illustre aussi la nature des problèmes appartenant à cette classe, qu'on peut résumer par : deviner et vérifier (en temps polynomial) ; on n'a pas à rechercher le certificat (ce qui demande en général un temps de calcul exponentiel), on est autorisé à le deviner, mais ensuite on a un algorithme polynomial de vérification du certificat.

Comme on l'a déjà mentionné, les problèmes dans NP sont en général plus difficiles à résoudre (du point de vue du temps de calcul) que les problèmes dans P. Appartenir à NP est une contrainte (celle de la vérification en temps polynomial), autrement dit la classe NP est loin d'être la classe des problèmes les plus difficiles à résoudre, comme la célébrité de la question P = NP pourrait le laisser croire

Exemple de problème NP : **Le problème SAT**

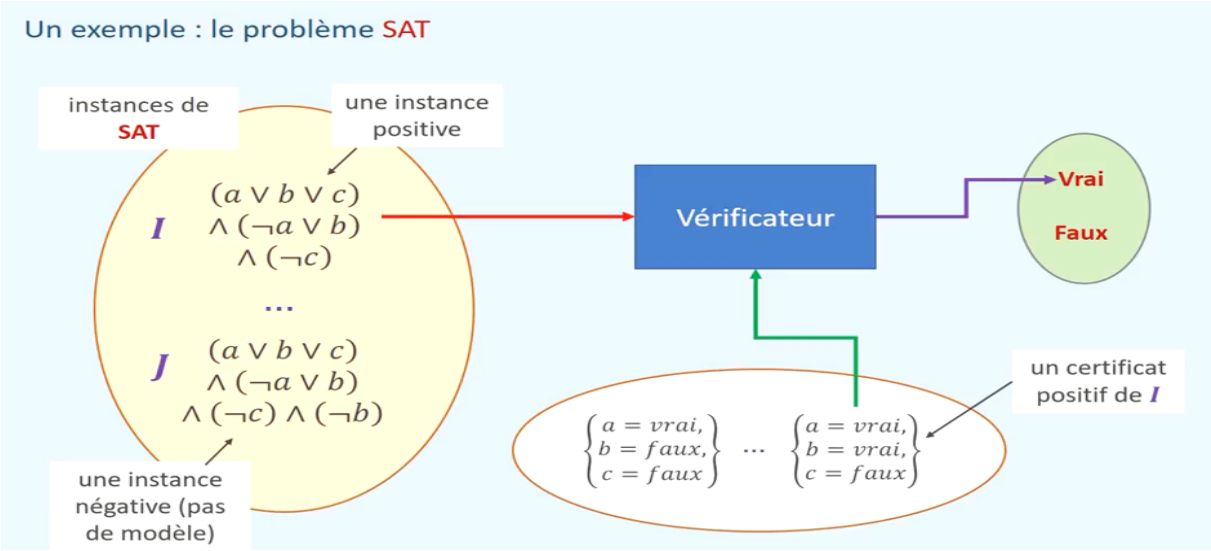


Figure 3: illustration du problème SAT

Les certificats sont des interprétations (attribution d'une valeur vraie à chaque variable d'une formule), le vérificateur détermine si une interprétation est un modèle d'une formule.

Et cette vérification se fait en temps polynomiale donc SAT appartient à NP.

3.3 Réduction Polynomiale :

La complexité exacte de certains problèmes tels que SAT et la coloration de graphes est inconnue à ce jour, elle a échappée à toutes les tentatives d'analyse mathématiques. Mais faute de pouvoir déterminer dans l'absolu la complexité de certains problèmes, on peut s'intéresser à leurs complexités relatives et se poser la question suivante : Un problème dont on ignore la complexité exacte est-il plus difficile qu'un autre problème dont on ignore la complexité exacte ? La notion de réduction polynomiale permet de donner une ébauche de réponse à cette question.

3.3.1 Définition :

Un problème A se réduit à un problème B au sens de la Turing réduction si et seulement s'il existe un solveur du problème A faisant appel au solveur du problème de B utilisé comme un sous-programme et que ce solveur de A s'exécute en temps polynomial si on ne comptabilise pas le temps d'exécution du solveur de B. Autrement dit, on imagine une machine capable de résoudre B instantanément (cette machine est appelée **oracle)** et on se demande si dans ces conditions, on pourrait résoudre A en temps polynomial, si la réponse est « oui » alors, on dit qu'A se réduit à B

Dans un monde ou « facile » signifie polynomial et difficile non polynomial, A n'est pas plus difficile que B :

On note $A \leq B$ lorsque A se réduit à B

On appelle Karp réduction, en référence au mathématicien Richard Karp, un cas particulier de Turing réduction, dans lequel le solveur de B est appelé une fois après traduction en temps polynomial de l'instance du problème A à traiter en instance du problème B.

On dit qu'un problème est NP-difficile si et seulement si tout problème de NP peut se réduire à ce problème au sens de la Turing réduction et NP-facile s'il peut se réduire à au moins un problème NP.

Un problème est NP-équivalent s'il est à la fois NP-difficile et NP-facile

Ces notions sont intéressantes pour situer la difficulté d'un problème n'appartenant pas à la classe NP par rapport à celles des problèmes les plus difficiles de la classe NP

Exemples : Le problème SAT peut se réduire au problème 3-SAT, car il est possible de traduire toute instance du problème SAT en instance du problème 3-SAT.

- Prouver que $SAT \leq 3-SAT$: Le problème SAT peut se réduire au problème 3-SAT

On va procéder par réduction en prouvant que le fait que SAT est NP-complet, implique clairement que 3-SAT est NP-complet.

- Soit F une formule SAT.
- Soit C une clause de F, par exemple $C = x \vee y \vee \neg z \vee u \vee \neg v \vee w \vee t$

- On introduit de nouvelles variables a, b, c, d associées à cette clause et on remplace C par la formule

$$R(C) = (x \vee y \vee a) \wedge (\neg a \vee \neg z \vee b) \wedge (\neg b \vee u \vee c) \wedge (\neg c \vee \neg v \vee d) \wedge (\neg d \vee w \vee t)$$

- Il est facile de vérifier qu'une assignation de x, y, z, u, v, w, t peut être complétée par une assignation d a, b, c, d de façon à rendre cette formule vraie si et seulement si C est vraie.

- A partir de $F = C_1 \wedge C_2 \ldots . \wedge C_l$, on construit $F' = R(C_1) \wedge R(C_2) \wedge \ldots . \wedge R(C_l)$
- On a donc que F ∈ SAT ssi $F' \in 3 - \text{SAT}$.
- Et la fonction qui à F associe F′se calcule en temps polynomial

4. CLASSE CO-NP

Le problème *complémentaire* (ou *dual*) du problème *A* est le problème *B* = ¬ *A* ; autrement dit *B* est déduit de *A* en échangeant les réponses "vrai" et "faux" (ou "oui" et non", ou 0 et 1). Ce vocabulaire est cohérent avec celui des ensembles : le langage associé à *B* est le complémentaire du langage associé à *A*.

A priori il semble évident qu'un problème et son complémentaire appartiennent à la même classe de complexité ; c'est exact tant qu'on considère des algorithmes *déterministes.*

4.1 Définition :

La classe des problèmes dont le complémentaire est dans **NP** est une nouvelle classe, notée **Co-NP**, et définie par :

> *A* ∈ **Co-NP** si et seulement s'il existe un algorithme polynomial *V* tel que *A* (u) ⇔ ∀*x V* (*u*, *x*)

En effet, en prenant la négation de la condition ci-dessus, on a : ¬ *A* (u) ⇔ ∃*x* ¬*V* (*u*, *x*)

Et donc le problème complémentaire *B* admet ¬*V* pour vérifieur. Un vérifieur est déterministe, donc l'échanger avec sa négation ne change rien ; par contre, la définition de **NP** utilise le quantificateur existentiel ∃, qui est remplacé par le quantificateur universel ∀ dans la définition de **Co-NP**.

On ne voit aucune raison pour qu'on puisse se débarrasser systématiquement du quantificateur ∃ dans la définition de **NP**, d'où la conjecture **P** ≠ **NP** ; de même on ne voit aucune raison pour qu'on puisse remplacer systématiquement le quantificateur ∀ par le quantificateur ∃ dans la définition de **Co-NP**, d'où la conjecture :

NP ≠ **Co-NP**

Cette seconde conjecture est moins médiatique, mais pratiquement aussi importante que la conjecture **P** ≠ **NP**. L'implication suivante est évidente :

P = **NP** ⇒ **NP** = **Co-NP**

Puisque, pour un algorithme déterministe, il n'y a pas de différence entre un problème et son complémentaire ; autrement dit la classe **P** est égal à *co-P* (qui ne mérite donc pas de caractères gras…). En prenant la contraposée de l'implication précédente, on a donc :

NP ≠ **Co-NP** ⇒ **P** ≠ **NP**

Il n'est pas impossible qu'on ait à la fois **P** ≠ **NP** et **NP** = **Co-NP**, mais personne ne le croit.

5. NP-COMPLETUDE

5.1 Motivation :

On souhaite comprendre quel problème est plus difficile dans NP ou plus généralement dans une classe C de problèmes de décision

Il a été démontré en 1971 par Steven Cook que tous les problèmes de NP se réduisent à SAT, de ce fait, on dit que SAT est NP-complet. Et comme SAT se réduit au problème de coloriage de Graphe, ce dernier est aussi NP-Complet par transitivité.

On dit souvent que les problèmes NP-complets sont les plus difficile de la classe NP, il faut l'entendre dans le sens où, s'il existait un algorithme polynomial, capable de résoudre un seul de ces problèmes, peu importe lequel, alors, on pourrait résoudre tous les problèmes de NP en temps polynomial et donc P serait égal à NP. Or nul ne sait si un tel algorithme existe et personne n'a réussi à démontrer qu'il n'existe pas.

A ces jours, des centaines de problèmes NP-complets ont été répertoriés dans de nombreux domaines tels que la logistique, la logique, la théorie des graphes…

Certaines restrictions des problèmes NP-Complets sont dans P

5.2 Définitions :

- Un problème A est dit **NP-difficile** si tout B de NP est tel que $B \leq A$.

Plus généralement un problème est C-difficile si tout problème B de C est tel que $B \leq A$.

Intuitivement, il est donc plus difficile que tous les problèmes dans la classe.

- Un problème A est dit **NP-complet** si en plus il on a : A ∈ NP

Plus généralement, un problème A est dit **C-complet** si en plus il on a : A ∈ C.

Autrement dit, A est NP-complet s'il est un élément maximum dans NP pour ≤

Exemple : Le théorème de **Cook-Levin** nous fait savoir qu'il y'a des problèmes NP-complets, il en donne un exemple précis : **le problème SAT est NP-complet.**

5.2.1 Conséquence :

P = NP si et seulement si SAT ∈ P, c'est-à-dire qu'il y'a un algorithme raisonnable pour SAT

Preuve : Puisque SAT est dans NP et si P = NP alors SAT ∈ P

Et réciproquement puisque SAT est complet pour tout problème B ∈ NP, $B \leq SAT$ et donc B ∈ P si SAT ∈ P.

5.2.2 Généralisation :

Soit A un problème NP-complet :

P = NP si et seulement si A ∈ P :

- Il suffira de trouver un problème A pour lequel il y aurait un algorithme raisonnable auquel cas on aura prouvé que P = NP, ou alors d'en trouver un qui n'est pas polynomial et dans ce cas on aura ainsi prouvé que P n'est pas égal à NP
- D'où l'intérêt de produire des problèmes NP-complets

5.3 A quoi sert de prouver la NP-complétude d'un problème ?

- Arriver à prouver que P = NP
  Remarque : Si un problème A et un problème B sont NP-complets alors
  $A \leq B$ **Et** $B \leq A$ c'est-à-dire que puisque B est complet alors A est plus facile que B **Et** puisque A est complet B est plus facile que A
- Tous les problèmes NP-complets sont donc de même difficulté

En réalité arriver à prouver que P=NP n'est pas facile car tous les problèmes NP-complets sont donc de même difficulté et ce n'est pas vraiment comme ça qu'on utile la NP-complétude ; C'est surtout pour la raison suivante :

- Supposons que l'on n'arrive pas à trouver un algorithme polynomial pour un problème.
- Prouver alors sa NP-complétude permet de se convaincre que cela n'est pas possible sauf si P= NP

5.4 Méthode pour prouver qu'un problème est NP-complet :

➢ Pour prouver la NP-complétude d'un problème A, il suffit :

1. De prouver qu'il admet un vérificateur polynomial
2. Et de prouver que $B \leq A$ pour tout problème B que l'on sait déjà NP-complet

Preuve :

- En effet, le point 1. Permet de garantir que A ∈ NP ,
- Et le point 2. Que pour tout problème C ∈ NP on a C ≤ A :

En effet, par la NP-complétude de B on a C ≤ B, et puisque $B \leq A$, on obtient $C \leq A$.

Cette méthode nécessite de connaitre une base le plus large possible des problèmes connus pour être NP-complets.

Exemple : 3-COLORABILITE est NP-complet.

Etape 1 : prouver que 3-COLORABILITE est dans NP

En effet la donnée d'une couleur pour chacun des sommets du graphe constitue un certificat vérifiable en temps polynomial

Etape 2 : montrer que 3-SAT ≤ 3-COLORABILITE

- On se donne donc une conjonction F de clauses à 3 littéraux, et il nous faut à partir de là, construire le graphe.
  - Il faut parvenir à traduire deux contraintes : une variable peut prendre la valeur 0 ou 1 d'une part et les règles d'évaluation d'une clause d'autre part.
- On construit un graphe ayant 3 + 3n + 5m sommets, ou n est le nombre de variables et m de clauses.

1. Les trois premiers, notés VRAI, FAUX, NSP sont reliés deux à deux en triangle
2. On associe pour chaque variable x_i un sommet x_i et un sommet $\neg x_i$: On relie deux à deux en triangle les sommets x_i , $\neg x_i$ et NSP.
3. On ajoute 5 sommets pour chaque clause $x \vee y \vee z$ selon le dessin suivant

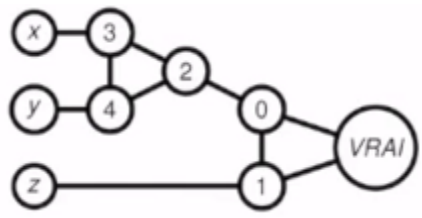


Figure 4: graphe utilisé pour l'algorithme

En effet, nous utilisons le **gadget** suivant :

Gadget :

- Si l'on impose que les 3 sommets à gauches sont soit bleus soit rouges,
  - Alors on peut colorer le sommet le plus à droite en bleu si et seulement si au moins un sommet à gauche est en bleu

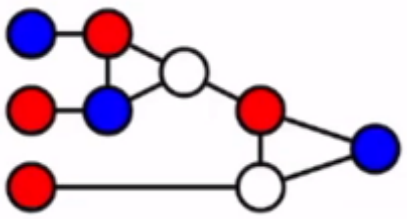


Figure 5: gadget

NB : bleu et rouge peuvent être remplacé par VRAI et FAUX

Ce gadget nous permet donc de coder une clause.

En effet,

- Si F est satisfiable, on peut colorier le graphe G(F) avec 3-couleurs
  Mettre les variables Vrai avec la couleur du sommet Vrai (par exemple bleu), les variables faussent avec la couleur fausse (par exemple rouge) et compléter.
- Si le graphe G(F) est coloriable avec 3 couleurs :
  - Le triangle 1. Assure que Vrai et Faux ne sont pas de la même couleur.
  - Le triangle 2. Assure que $x_i \, et \quad \neg x_i$ ne sont pas de la même couleur pour chaque variable x_i et soit de la couleur de Vrai ou de la couleur de Faux.
  - Le gadget 3. Assure que pour chaque clause C au moins x ou y ou z est de couleur Vrai.
  - En prenant pour Vrai toutes les variables de la couleur de Vrai, on obtient une affectation des variables qui satisfait la formule F ;

- En conclusion : F ∈ 3-SAT ssi $G(F)$ ∈ 3-COLORABILITE.
- Et la fonction qui à F associe G(F) se calcule bien en un temps polynomial

Attention (pour mémoire)

La NP-complétude d'un problème A s'obtient en prouvant qu'il est plus difficile qu'un autre problème NP-complet et pas le contraire, C'est une erreur fréquente dans les raisonnements.

5.5 Comment gérer la NP-complétude ?

- En cherchant à contourner la / les difficultés
- En relâchant les contraintes : par exemple en autorisant des solutions approchées.

Exemple : Le problème du voyageur de commerce

Formulation : un représentant doit visiter n villes. Le représentant souhaite faire une tournée en visitant chaque ville au moins et exactement une fois, et en terminant par sa ville de départ.

❖ **Sous forme de graphe :**

- ✓ On se donne un graphe connexe G = (V, E), avec un poids w(e) pour chaque arête e = (u, v)∈ E.
- ✓ On veut déterminer un cycle hamiltonien (un cycle simple qui passe par tous les sommets) de G de poids minimal.

- On ne connait aucun algorithme pour le résoudre en temps polynomial (par tout ce qui précède, ce problème est équivalent à la question P = NP ?).
- Même si l'on se limite au cas Euclidien (les poids vérifient l'inégalité triangulaire), le problème du voyageur de commerce reste NP-complet

Une autre façon d'essayer d contourner le problème c'est de remplacer l'idée qu'on doit produire une solution optimale par une approximation.

❖ Le cas euclidien (avec l'inégalité triangulaire) est 2-approximable

- On se place dans le cas suivant :
  - Le graphe est complet : pour toute paire d sommets u, v il y'a une arête (u, v).
  - Les poids vérifient l'inégalité triangulaire :

    Pour 3 sommets u, v, w arbitraires de V, w (u, v) ≤ w (u, w) + w (w, v).
- On ne connait pas d'algorithme en temps polynomial qui produit une solution optimale (sauf si P = NP).
- Mais on connait un algorithme en temps polynomial qui produit une solution dont le poids est inférieur au double de l'optimal
  Poids (Solution de l'algorithme) ≤ 2 x Poids (cycle optimal).

Algorithme :

- Construire un arbre couvrant de poids minimal T et G.
- Soit L la liste des sommets de G dans l'ordre ou ils sont visités dans un parcourt préfixe de T.
- Renvoyer le cycle hamiltonien qui visite les sommets dans l'ordre de L

Exemple : graphe initial (les poids sont des distances euclidiennes)

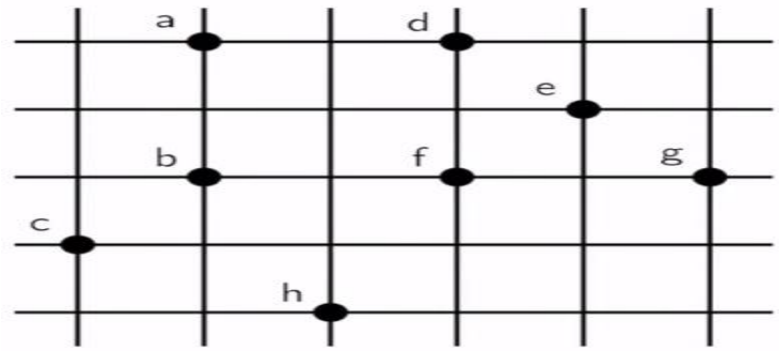


Figure 6:graphe initial

- L'algorithme détermine un arbre couvrant T de poids minimal

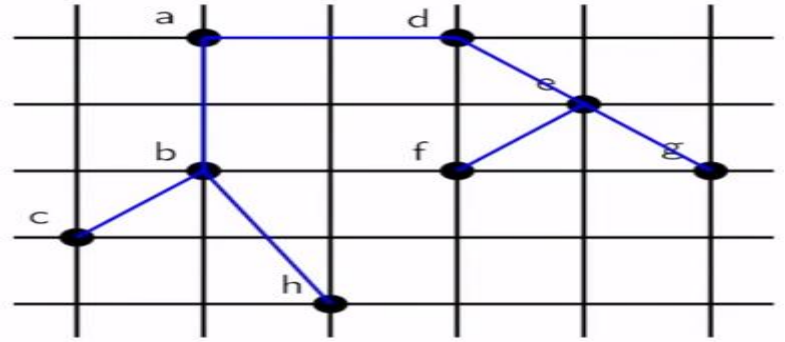


Figure 7: arbre couvrant

- Il consiste à faire un parcourt préfixe de T : abchdefg

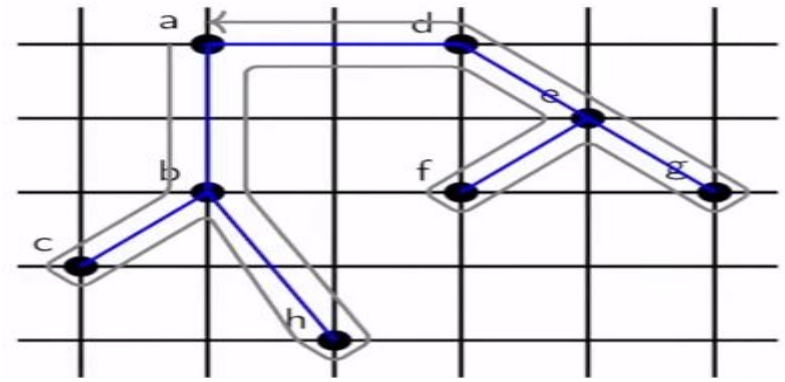


Figure 8:chemin trouvé

Résultat : On considère les sommets dans l'ordre qui correspond au parcourt préfixe

5.6 Quelques exemples de problèmes NP-complets

5.6.1 Sur les graphes :

- **Le problème CLIQUE**

Donnée : Un graphe G = (V, E) non orienté et un entier k.
Réponse : on veut savoir s'il possède une clique de taille k. C'est-à-dire décider s'il existe $V' \subset V, \text{avec } |V'| = k, \text{tel que } u, v \in V' \Rightarrow (u, v) \in E$

En passant au complémentaire sur les arêtes

- **Le problème STABLE**

Donnée : Un graphe G = (V, E) non orienté et un entier k.
Réponse : on veut savoir s'il y'a un sous ensemble de sommets tel que pout toute paire de sommet, il n'y a pas d'arêtes entre ses deux sommets. C'est-à-dire décider s'il existe $V' \subset V, \text{avec } |V'| = k, \text{tel que } u, v \in V' \Rightarrow (u, v) \notin E$

En passant au complémentaire sur les arêtes

- **Le problème RECOUVREMENT DES SOMMETS**

Donnée : Un graphe G = (V, E) non orienté et un entier k.
Réponse : décider s'il existe $V' \subset V, \text{avec } |V'| = k,$ tel que toute arete de G ait au moins une extremité dans V'

- **Le problème DU CIRCUIT HAMILTONIEN**

Donnée : Un graphe G = (V, E) non orienté.
Réponse : Décider s'il existe un circuit hamiltonien, c'est-à-dire un chemin de G passant une et une seule fois par chacun des sommets et revenant à son point de départ.

- **Le problème du VOYAGEUR DE COMMERCE**

Donnée : Un couple (n, M), ou M est une matrice n x n d'entiers et un entier k.
Réponse : Décider s'il existe une permutation Π de $[1, 2, \ldots . n]$ telle que

$$\sum_{1 \leq i \leq n} M_{\Pi(i)\Pi(i+1)} \leq k$$

Ce problème porte ce nom, car on peut voir cela l'établissement de la tournée d'un voyageur de commerce devant visiter n villes, dont les distances sont données par la matrice M de façon à faire moins de k-kilomètres.

5.6.2 Sur les entiers :

- **Le problème de la SOMME SE SOUS ENSEMBLE**

Donnée : Une suite finie d'entiers x_1 , x_2 , ……x_n et un entier t.
Réponse : Décider s'il existe E⊂ $\{1,\ 2, \ldots \ldots n\}$ tel que $\sum_{i \in E} x_i = t$.

- **Le problème de PARTITION**

Donnée : Une suite finie d'entiers x_1 , x_2 , ……x_n .
Réponse : Décider s'il existe $E \subset \{1,\ 2, \ldots \ldots n\}$ tel que $\sum_{i \in E} x_i = \sum_{i \notin E} x_i$.

- **Le problème du SAC A DOS**

Donnée : Un ensemble d'objets de poids a_1 , a_2 , ……a_n, et un ensemble de valeurs v_1 , v , ……v_n, un poids limite A et un entier V
Réponse : Décider s'il existe $E' \subset \{1,\ 2, \ldots \ldots n\}$ tel que $\sum_{i \in E'} a_i \leq A$ et $\sum_{i \in E'} v_i \leq V$

CONCLUSION

Au-delà des problèmes de décisions, il existe des problèmes dont la réponse peut être un objet. Par exemple, le coloriage de graphe peut être décliné en problème de recherche : existe-t-il un graphe qu'on peut colorier avec k-couleur ? si oui trouver ce graphe. Il peut aussi être décliné en problème d'optimisation : étant donné un graphe G quel est le plus petit entier k pour lequel il existe un k coloriage correct de ce graphe ?

La Turing réduction permet de comparer les complexités de ces problèmes.

BIBLIOGRAPHIE

- Introduction à l'algorithme

WEBOGRAPHIE

- https://www.labri.fr/perso/betrema/MC/MC8.html
- https://fr.wikipedia.org/wiki/P_(complexit%C3%A9)
- http://www.enseignement.polytechnique.fr/informatique/INF550/Cours1112/INF550-2011-4.pdf
- http://www2.ift.ulaval.ca/~bherer/Chapitre%209%20(np-completude).pdf

FICHE DE TRAVAUX DIRIGES

Exercice 1

Montrer que le problème 3-SAT se réduit au problème COLORING.

Exercice 2 : colorabilité

1. Montrer que le problème de k-colorabilité´ se réduit polynomialement au problème de (k + 1) -colorabilité

2. Dans quelle classe de complexité est le problème 2-colorabilite´ ?

3. Montrer que le problème 3-colorabilite´ est NP-complet.

Exercice 3 : Cycles.

1. Montrer que le problème **EC** (l'ensemble des graphes ayant un cycle eulérien) est dans P.

2. Montrer que les problèmes **UHC** (graphes ayant un cycle hamiltonien) et **UHP** (graphes ayant un chemin hamiltonien) sont polynomialement ´équivalents.

3. Montrer que les problèmes **DHC** (graphes orientes ayant un cycle hamiltonien) et **DHP** (graphes orientes ayant un chemin hamiltonien) sont polynomialement ´équivalents.

4. Montrer que les problèmes **DHC** (graphes orientes ayant un cycle hamiltonien) et **UHC** (graphes ayant un cycle hamiltonien) sont polynomialement ´équivalents.

5. Sachant que **DHP** est NP-complet. En déduire que **UHC, DHP, DHC** sont également NP-complets.

Exercice 4 : Un autre point de départ NP-complet

1. Expliquer le fonctionnement d'une machine de Turing non d´déterministe fonctionnant en temps polynomial qui reconnaisse L1 = {(<M>, t) : t $\leq$ |Q| et M admet un calcul acceptant sur le mot vide en temps $\leq$ t}.

2. Montrer que L1 est NP-complet. On pourra commencer, ´étant donnée une machine de Turing non d´déterministe M fonctionnant en temps polynomial p(n) et un mot x, construire une machine de Turing Mx avec un nombre d'états > 2 |x| + p(|x|) qui admet un calcul acceptant sur le mot vide en temps 2 |x| + p(|x|) ssi M admet un calcul acceptant sur x en temps $\leq$ p(|x|).

3. On considère un modèle de machine de Turing non ´déterministe avec un seul ruban en lecture ´écriture. Montrer que si M est non déterministe au sens du cours, on peut trouver une machine M0 avec un seul ruban qui reconnait le même langage que M et qui fonctionne en temps$\leq$ K (1 + t 2) ou t est le temps de calcul de M et K est une constante.

Exercice 5 :

Si P= NP, montrer que le problème de calculer une affectation de valeurs satisfaisant une formule propositionnelle peut ˆêtre résolu en temps polynomial.