INFO-F408: Computability & complexity

Rémy Detobel

18 Septembre 2017

1 Introduction

C'est donc un cours de math, permettant de classer les problèmes, leur complexité, ... Il s'agit de la suite du cours d'"informatique fondamentale".

1.1 Référence du cours

— Introduction to the Theory of computation 2nd Ed. Michael SIPSER.
— Hilbert (mathématicien), Gödel (il aurait prouvé que complètement transformer les mathématiques en "programme" n'est pas possible), Turing, Kleene, Church, Cook-Levin (NP et ses implications), Karp

1.2 Définition

— **What problems can be solved by a computer ?**
On doit définir plusieurs mots : "problems", "solved" et "computer".
— "Efficace" : quel est réellement sa définition. Que veut-on optimiser : le temps, l'énergie utilisé, la place, ressource naturelle (pour créer la machine par exemple).
— **How much resources are "required" to solve a problem ?**
— Parfois deux problèmes sont très différent mais en résolvant un des deux, on peut très facilement solutionner le problème en changeant peu l'algorithme.
— **Mathematical Treatment**
Il y a une certaine formalisme à avoir pour pouvoir caractériser, classer, ... les problèmes. Il y a différent niveau de langages. Il ne faut pas se satisfaire de comprendre en général mais bien en profondeur.
— **Compilateurs, langages de programmation**
— **Optimisation : problème linéaire (LP), algorithme génétique, problème de satisfaisabilité, plus court chemin**
— **P vs NP (traveling problem (voyageur de commerce))**

1.3 Connexion avec d'autres matières

Biologie, physique (ordinateur quantum (pas abordé dans ce cours)), ...

1.4 Quelques concepts de bases

Sets, logique booléen, séquences, relations, fonctions, graphes, "strings"/chaînes de caractères, preuve par l'absurde ("proof by contradiction").
Référence : chapitre 0 du livre pour se rappeler.

1.5 Problème de décision

Problème où la réponse est "oui" ou "non". Certains peuvent être résolus par des algorithmes et des ordinateurs. Il faut donner des informations qui permettent de décider si la réponse est "oui" ou "non". C'est là qu'est le problème principal des algorithmes. On va donc utiliser seulement des problèmes dont la solution est déjà définie

par l'entrée (e.g. y a-t-il un chemin entre deux points d'un graphe).

— well-defined
— self-contrained

Il y a une différence entre problème et une instance. Le problème est plus général qu'une instance. Une instance est un cas particulier d'un problème, dans lequel on a donc définit les entrées/inputs. Chaque "input" possible est donc une instance...
Problème : la liste des entrée pour lesquelles l'entrée est "oui".

1.5.1 Exemples

Problème : dire si un nombre est pair ou non.
Solution : tous les nombres pairs (donc tous les nombres impairs sont faux).

Problème : un graphe est-il connexe (donc est-il possible d'aller de n'importe quel point dans le graphe vers n'importe quel autre point). **Solution :** tous les graphes connexes (on peut les encoder sous forme de matrice (0 pas de lien, 1 si un lien)).

1.5.2 Input/Entrée

Les entrées/input doivent appartenir à un langages (qui peut être $\{0, 1\}$) : une liste de mots/chaînes de caractères. On considérera aussi que lorsque l'input n'est pas bon, la réponse est "non". Pour le second exemple donc, si on ne reçoit pas un graphe en entrée, on répondra "non". Il faudra donc que l'algorithme vérifie d'abord l'entrée avant d'essayer de résoudre.

1.6 Problème de décision $\neq$ problème d'optimisation

Bisection/dichotomie
Un problème de décision peut aider à savoir s'il est optimisé...
Exemple (problème de décision) : est-ce que le chemin coûte moins que la valeur "d" ? On prend "d" comme le milieu entre 0 et le maximum (nombre de point dans le graphe). **Transformé en problème de décision** : en général c'est possible en reformulant le problème.

2 Automate fini

Cela permet de déjà commencer à répondre à la question "qu'est-ce qu'un ordinateur". 5-uple : $(Q, \Sigma, \delta, q_0, F)$
Référence : chapitre 1.1 du livre

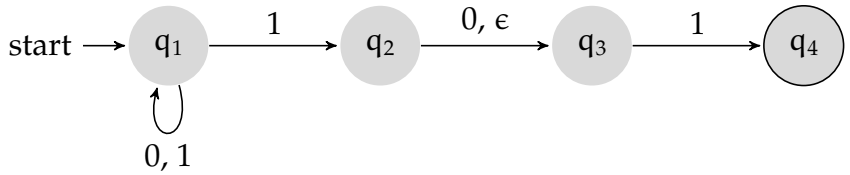
Q = nombre fini d'état (finite set of states).
Σ = alphabet utilisé par le problème (finite set called the alphabet)
$\delta : Q \times \Sigma \to Q$ (produit cartésien (cartesian product)) fonction de transition
$q_0 \in Q$ c'est l'élément de départ (the start state)
$F \subseteq Q$ liste des états finaux (set of accepting states)

2.1 Regular language

Définition : livre point 1.16
A language is called a regular language if some finite automaton recognizes it. (= Regular Exepression (REGEX))
Finite automaton M defines a language $L(M)$.
Exemple : $A = \{w \in \{0,1\} * | w$ contains at least on 1 and an even number of 0 follow the last 1$\}$

3 No determinism / Non déterministe

Référence : chapitre 1.2 du livre
Les automates finis sont déterministes.
ϵ = "empty" symbol



Dans un automate non déterministe le δ ne sera pas définit de la même manière. Il sera écrit :

$$\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$$

$P(Q)$ liste d'états dans Q (set of states in Q)
Les automates "non déterministes" reconnaissent exactement les mêmes langages que les automates "déterministes". Cela dit, les automates non déterministes sont plus simple à écrire et à comprendre au premier abord (la preuve arrivera).
Voir livre 1.39.
Chaque NFA (automate non déterministe) à un équivalent en DFA (automate déterministe).

INFO-F408: Computability & complexity

Rémy Detobel

25 Septembre 2017

1 Automates

1.1 Non déterministe

DFA

$$\delta : Q \times Z \to Q$$

$Q \to$ liste des états
$Z \to$ l'alphabet (liste de caractère)
$\delta \to$ langage

Pour les automates non déterministes, on pourra utiliser le symbole ϵ représentant un caractère vide. Mais également avoir deux transitions d'un état vers d'autres états où la transition serait la même... On a donc quelque chose de non déterministe.
Dans ce cas la, le second Q (valeurs de la fonction δ) sera donc une liste de liste d'état possible.

$$\delta : Q \times Z_\epsilon \to \mathcal{P}(Q)$$

Exemple :

$$\begin{aligned} Q &= \{q_1, q_2, q_3\} \\ P(Q) &= \{\emptyset, \{q_1\}, \{q_2\}, \{q_3\}, \{q_1, q_2\}, \{q_2, q_3\}, \{q_1, q_3\}, Q\}. \\ \left|P(Q)\right| &= 2^{|Q|} \end{aligned}$$

Voir livre : théorème 1.39 (page 55) :
Chaque NFA a un équivalent en DFA (qui reconnait le même langage).
Concernant le chapitre 1, nous n'avons pas tout vu. Il ne faut donc pas tout connaitre...

1.1.1 Transformer un NFA en DFA

$$\begin{gathered} \text{NFA} \\ N = (Q, \Sigma, \delta, q_0, F) \\ \text{DFA} \\ M = (Q', \Sigma, \delta', q_0', F') \end{gathered}$$

On va donc faire ça étape par étape :

$$\begin{aligned} Q' &= \mathcal{P}(Q) \\ \delta'(R, a) &= \bigcup_{r \in R} \delta(r, a) \\ q_0' &= \{q_0\} \\ F' &= \{R \in Q' | R \text{ contains an element of } F\} \\ &\leftrightarrow R \cap F \neq \emptyset \end{aligned}$$

Ce changement à un inconvénient : il est exponentiel... Mais dans le cas présent nous n'allons pas nous occuper de la taille des automates.

2 The Church-Turing Thesis

Livre : Chapitre 3

De manière intuitive, on peut dire qu'une machine de Turing a au moins une mémoire. Qui sera appelée un "TAPE" (un ruban). Sur ce ruban on met donc une tête de lecture (head) pour lire les éléments un à un. On se déplace donc vers la gauche ou vers la droite.

On peut ensuite faire des opérations comme remplacer l'information et se déplacer vers la droite. Une grande différence (d'un point de vue conceptuel) avec les pc d'aujourd'hui réside dans le fait qu'il n'y ait pas d'accès "random"/aléatoire à la mémoire. Il faut faire plusieurs opérations.

On considérera ici que le ruban est infini et un nombre d'états fini ainsi qu'une fonction de transition sous la forme :

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\},$$

avec $\delta(q, \gamma_1) = (q', \gamma_2, d)$ ($d \in \{L, R\}$) où q est l'état de la machine, γ_1, le caractère lu, q', l'état après la transition, γ_2, le caractère à écrire sur le ruban, et d la direction (gauche ou droite).

$Q \to$ liste des états

$\Gamma \to$ alphabet sur le ruban (γ_1 est la lettre lue, et γ_2 la lettre que l'on écrit sur le ruban)

$\{L, R\}$ gauche ou droite

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accepté}}, q_{\text{rej.}})$$

$\Sigma \to$ alphabet présent sur le ruban

$$\Sigma \subseteq \Gamma$$

Exemple 3.9 Décision :

$$B = \{w\#w | w \in \{0, 1\}^*\}$$

Exemple

$00\#01 \in B \to$ Faux

$010\#010 \in B \to$ Vrai

Lorsque l'on est en q_1 et que l'on voit un 0, on peut écrire : $\delta(q_1, 0) = (q_2, x, R)$

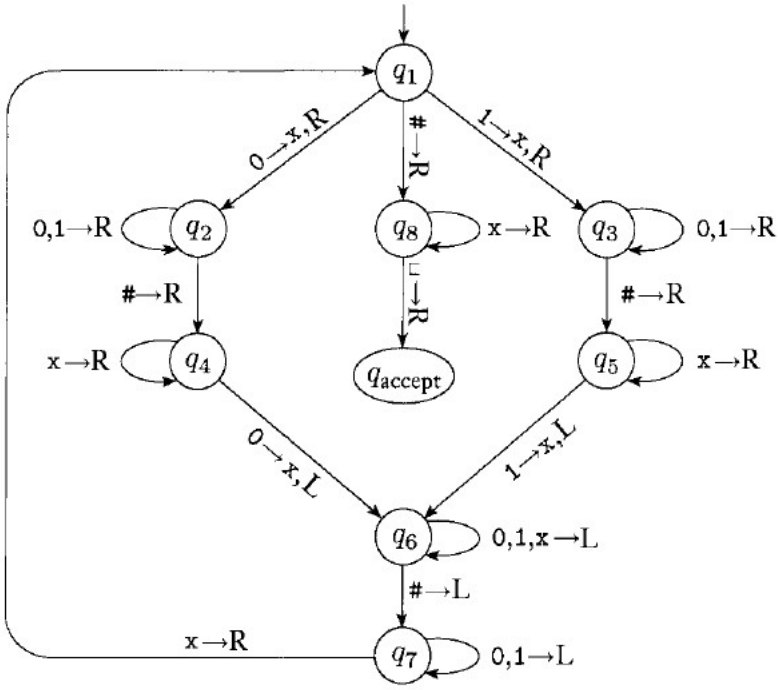
$$\delta(q_2, 0) = (q_2, 0, R)$$
$$\delta(q_2, 1) = (q_2, 1, R)$$

Note : "␣" désigne le symbole "blanc", vide...

EXAMPLE 3.9

The following is a formal description of $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$, the Turing machine that we informally described (page 139) for deciding the language $B = \{w\#w | \ w \in \{0,1\}^*\}$.

- $Q = \{q_1, \ldots, q_{14}, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0,1,\#\}$, and $\Gamma = \{0,1,\#,\mathtt{x},\sqcup\}$.
- We describe δ with a state diagram (see the following figure).
- The start, accept, and reject states are q_1, q_{accept}, and q_{reject}.



2.1 "Turing-decidabilité"

Un langage L est "Turing-decidable" ⇔ il existe une machine de Turing qui s'arrête sur toutes les entrées, comme si L était la liste des états acceptés.
Une machine de Turing qui s'arrête sur toutes les entrées (donc qui finit en un des deux états finaux $q_{\text{accepté}}$ ou $q_{\text{rej.}}$) est appelé "Decider".

2.2 "Turing-Recognizable"

Un langage L est "Turing-Recognizable" ⇔ il existe une machine de Turing telle que L est une liste d'états d'entrée acceptables. Parfois ici la machine ne s'arrête pas (et ces mots ne se donc pas dans le langage L).

2.3 Différence entre les deux

Decidable
Si on prend l'entrée x. Si il existe une machine de Turing tel que :
— si $x \in L$: cela s'arrête sur $q_{\text{accepté}}$
— si $x \notin L$: cela s'arrête sur $q_{\text{rejeté}}$
Reconnaitre
Si on prend l'entrée x. Si il existe une machine de Turing tel que :
— si $x \in L$: cela s'arrête sur $q_{\text{accepté}}$
— si $x \notin L$: cela s'arrête sur $q_{\text{rejeté}}$ OU il ne s'arrête jamais.
Par définition, les langages qui sont décidable, sont des langages que l'on peut "reconnaitre".

2.4 Variantes des machines de Turing

Livre 3.2 (page 148)

2.4.1 Multitape Turing machine

Si l'on a 3 rubans (de manière générale K rubans), on pourrait donc écrire de manière mathématique

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^K \times \{L, R, S\}^K$$

Où $\{L, R, S\}$ correspond (respectivement) à gauche (left), droite (right), rester (stay).
Exemple : $\delta(q, (a, b, c)) \to (q', (a, b, d), (L, R, S))$
Théorème 3.13 : Chaque multitape Turing machine a un équivalent (single-tape) en Turing Machine.

2.4.2 Simuler du Multitape sur une simple Turing Machine

L'idée est de "simuler" les "k tapes" sur une seule en se déplaçant. Pour ce faire, on augmente l'alphabet Γ avec "#" et le symbole $\dot{x}$ pour chaque $x \in \Gamma$ représentant les têtes.
Ici on ne fait pas attention à la complexité, l'efficacité. C'est évidemment plus lourd de tout mettre sur un ruban mais l'important c'est que ça soit possible.

2.4.3 Machine de Turing non déterministe

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

On va donc avoir des "branches" d'exécution. Quand l'entrée est acceptée, cela signifie qu'il existe une branche menant à $q_{\text{accepté}}$

2.5 Computation tree (arbre d'exécution)

Théorème 3.16 Chaque NTP a un équivalent (D)TM (qui reconnait donc le même langage).
Il y a plusieurs moyen de parcourir cet arbre : parcours en profondeur, parcours par

niveau, parcours en largeur. C'est ce dernier algorithme qui sera privilégié. Pour faire cela avec un TM, on va utiliser un 3-Tape (3 rubans) Turing machine déterministe. On aura donc 3 rubans avec : l'entrée, la simulation et l'adresse. En simulant cela, on va pouvoir voir si on arrive sur un statut accepté.

INFO-F408: Computability & complexity

Rémy Detobel

2 Octobre, 2017

1 Turing machine suite

1.1 Non déterministe

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \times \Gamma \times \{L, R\})$$

Voir livre : théorème 3.16 : Chaque NTM as un équivalent DTM.
On va donc faire un parcours de l'arbre en largeur (et non en profondeur).

1.2 Reconnaitre un langage de Turing

Voir théorème 3.21 :
Un langage est "Turing-recognizable" si et seulement si un "enumerator" l'énumère.

1.2.1 Démonstration

($\Leftarrow$) Supposons qu'il existe un tel énumérateur "E" :
Soit la machine de Turing M. Informellement : "lorsque l'entrée est w

1. Exécuter E, et chaque fois que E écrit(/output) un string, on le compare avec w
2. si le string contient w, on accepte."

($\Rightarrow$) Supposons qu'il existe une machine de Turing qui reconnaisse le langage L.
E = "ignorer l'entrée"

1. Répéter pour $i = \mathbb{N}^*$:
   Exécuter M pour i étapes, sur les entrées $S_1, S_2, \ldots, S_i$.
   Si une exécution est acceptée, on affiche le S_j correspondant.
   Au pire on fera i étapes pour afficher un mot, mais il pourra être affiché avant l'étape i

step/input	S_1	S_2	S_3	S_4	S_5
1	x				
2	x	x			
3	x	x	x		
4	x	x	x	x	
5	x	x	x	x	x

1.3 Langages réguliers (regular languages)

Langage régulier = reconnaissable par un automate fini (FA : finite automaton)
Langage décidable (= decidable = recursively) = décidable par une machine de Turing.
Langage reconnaissable (recognizable languages = recursively enumerable = RE) =

— reconnaissable par une machine de Turing,
— et admet un enumerateur ("enumerator")

Régulier < décidable < reconnaissable/recognizable

2 The Church-Turing thesis

C'est une thèse, pas une preuve.
$\Rightarrow$ La notion intuitive d'un algorithme est égal à un algorithme d'une machine de turing

2.1 Hilbert Problem

Est-ce qu'il existe un algorithme qui décide si un polynôme admet une racine composée uniquement de nombre entiers.
Exemple :

$$P(x) = x_1^2 + x_2x_3^4 - 6x_1x_2^3x_3x_4^2 + 7x_1$$

Et on cherche donc des nombres entier x_1, x_2, x_3, x_4

Il s'agit ici d'un problème "recognizable" (reconnaissable). Car si il y a une solution, on pourra la voir. Par contre, il n'est pas "décidable" parce que s'il n'y a pas de solution, il tournera à l'infini.

L'indécidabilité de ce problème à été prouvé en 1970 par Matijasevic.

3 Halting problème (problème de l'arrêt)

Point 4.2.
Diagonalization (cantor) $f : A \to B$ est :
"un à un" (injective) si tous les élément de A sont projeté de manière distincte sur des éléments de B.
"dans" (surjective) lorsque tous les éléments de B admettent une préimage dans A, i.e. :

$$\forall b \in B : \exists a \in A | f(a) = b.$$

"one-to-one" (un à un) ET "onto" (dans) = "one-to-one correspondance"
C'est équivalent à une bijection.

Une ensemble A est "countable" (dénombrable) s'il existe une correspondance un à un ("one-to-one correspondence") entre A et $\mathbb{N}$ (ce qui est équivalent à dire qu'il a la même "taille" que $\mathbb{N}$). Un ensemble est "at most countable" (*au plus dénombrable*) s'il est fini OU dénombrable.
Exemple : est-ce que :

- les nombres paires sont dénombrable ?
  $\to$ Oui ($\mathbb{N}/2$)
- les nombres rationnels ($\mathbb{Q}$) sont dénombrable ?
  $\to$ Oui (pour cela il faut juste mettre un ordre. Pour se faire, on peut parcourir un tableau à double entrées représentant les numérateurs et dénominateurs. Il suffirait donc de simplement définir l'ordre de lecture qui logiquement se ferait plutôt en diagonal). De manière un brin formelle, $\mathbb{N} \to \mathbb{Q} : m \mapsto \frac{m}{1}$ est

une injection de $\mathbb{N}$ dans $\mathbb{Q}$, donc $|\mathbb{N}| \leqslant |\mathbb{Q}|$. De même $\mathbb{Q} \to \mathbb{Z} \times \mathbb{N} : \frac{a}{b} \mapsto (a, b)$ est une injection de $\mathbb{Q}$ dans $\mathbb{Z} \times \mathbb{N}$, donc $|\mathbb{Q}| \leqslant |\mathbb{Z} \times \mathbb{N}|$. Or $|\mathbb{Z}| = \mathbb{N}$, et $\left|\mathbb{N}^2\right| = |\mathbb{N}|$. Donc :

$$|\mathbb{N}| \leqslant |\mathbb{Q}| \leqslant |\mathbb{Z} \times \mathbb{N}| = |\mathbb{N} \times \mathbb{N}| = \left|\mathbb{N}^2\right| = |\mathbb{N}|.$$

On en déduit que toutes les quantités ici sont égales, et donc $|\mathbb{Q}| = \mathbb{N}$. (Pour démontrer cela rigoureusement, il faudrait expliciter les bijections $\mathbb{Z} \to \mathbb{N}$ et $\mathbb{N}^2 \to \mathbb{N}$ et les composer ; puisqu'une composée de bijections est une bijection, on a finalement une bijection de $\mathbb{Q}$ dans $\mathbb{N}$.)

— $\mathbb{Z}$ est dénombrable ?
→ Oui (nombre négatif étant des paires, nombre positif étant des impaires. De cette manière on compte tous les nombres) :

$$\varphi : \mathbb{Z} \to \mathbb{N} : n \mapsto \begin{cases} 2n + 1 & \text{si } n \in \mathbb{N} \\ -2n & \text{sinon.} \end{cases}$$

3.1 Cantor's Diagonal

Théorème : $\mathbb{R}$ est non-dénombrable ("not countable").
Prouvons cela par contradiction :
Supposons donc que $\mathbb{R}$ est dénombrable. On a donc une liste qui fait correspondre tous les nombres naturels ($\mathbb{N}$) à un nombre présent dans $\mathbb{R}$). On va donc prouver qu'il existe un $x \in [0, 1)$ qui n'est pas dans cette liste. Pour construire le x, on va prendre le

1	0,**3**1415926535
2	1,0**0**000000000
3	22,12**3**12312312
4	323,010**1**0101010
5	4,1502**6**535010
6	...

nom à la position i et l'incrémenter. Ici x vaut donc : $x = 0,41427...$ Donc, par construction, il ne peut pas être dans la liste car il diffère de tous les éléments de la liste.

Prenons $\mathcal{L}$ comme étant l'ensemble des langages sur l'alphabet Σ
Prouver que $\mathcal{L}$ est indénombrable ("uncountable").

INFO-F408: Computability & complexity

Rémy Detobel

9 Octobre, 2017

1 Cantor's Diagonal

Corollary 4.18 dans le livre :
Certain langages ne sont pas reconnaissable par une machine de Turing (Turing-recognizable).
Idée :
L'ensemble des machines des turing est dénombrable.

Exemple :
Tous les mots possibles dans l'alphabet : $\{0, 1\}$:

	ε	0	1	00	01	10	11	000	001
M_1	**0**	1	1	0	1	1	1	0	0
M_2	1	**1**	1	1	0	1	0	1	1
M_3	0	1	**0**	1	1	0	1	1	1
M_4									
M_5									

On construit donc ici un table qui définit un langage.

2 Halting Problem (problème de l'arrêt)

$$A_{TM} = \{\langle M, w\rangle | M \text{ est une machine de Turing qui accepte } w\}$$

M = Une machine de Turing
w = un mot en entrée (input)
Voir le livre, théorème 4.11 : le problème A_{TM} est indécidable.

2.1 Preuves

Par contradiction : supposons que A_{TM} est décidable. Cela signifie qu'il existe une machine $H(\langle M, w\rangle) = \{$Accepte si M accepte w et rejette dans les autres cas$\}$
On définit ensuite une machine D telle que pour l'entrée M(une machine de turing) :

1. Exécuter H sur l'entrée $\langle M, \langle M\rangle\rangle$
2. On renvoie l'inverse de H : accepte si M n'accepte pas $\langle M\rangle$ et rejette si M accepte $\langle M\rangle$.

Enfin, on exécute D sur lui-même : $D(\langle D\rangle)$.
Cela signifie que D s'accepte uniquement s'il ne s'accepte pas, ce qui est une contradiction, et donc une telle machine M ne peut exister.
Pour un langage A, on définit $\bar{A}$ comme étant son complément : $\bar{A} = \{w | w \notin A\}$
Supposons que A et $\bar{A}$ sont ("recognizable") reconnaissables. Donc A et $\bar{A}$ sont aussi décidables.
Preuve : Posons M et M′ reconnaissant respectivement A et $\bar{A}$. Construisons un "décideur" D pour A en exécutant M et M′ en "parallèle" (en alternant étape par étape

sur M et sur M′).

Posons maintenant A_{TM} comme étant indécidable. Est-il pour autant reconnaissable ("recognizable") ?
A_{TM} est reconnaissable (preuve : simuler M sur w).
On peut également écrire :

$$A \text{ est décidable} \Leftrightarrow A \text{ et } \bar{A} \text{ sont reconnaissables}$$

$\Rightarrow \overline{A_{TM}}$ n'est pas reconnaissable.

$$\overline{A_{TM}} = \{\langle M, w\rangle | w \text{ n'est pas accepté par } M\}$$

3 Reductibility (Réduction)

Pour une machine de Turing M, $L(M)$ décrit le langage reconnu par M.

$$L(M) = \{w | M \text{ accepte w}\} \subset \Sigma^*$$

$$REGULAR_{TM} = \{\langle M\rangle | M \text{ est une machine de Turing et } L(M) \text{ est régulier}\}$$

Rappel d'un langage régulier : qui peut être reconnu par un automate fini. Ce langage est indécidable.
Voir **livre**, théorème 5.3
Preuve :
Définissons une Machine de Turing M_2 comme une fonction de M (une machine de Turing) et w ($\in \Sigma^*$)
$M_2 =$ pour une certaine entrée x :

1. si x a la forme 0^n1^n, on accepte
2. sinon, on exécute M sur w et on accepte si M accepte w.

M_2 n'est pas spécialement un "décideur". Cela va déprendre de M. Notons également que M_2 est une "fonction" de M et w.

Quel est le langage de $L(M_2)$ **?**

1. Si M accepte w : alors $L(M_2) = \Sigma^*$ accepte tout.
   $\Rightarrow$ régulier
2. Si M n'accepte pas w, $L(M_2) = \{0^n1^n | n \geqslant 0\}$
   $\Rightarrow$ pas régulier

On a donc réussi à réduire le problème de l'arrêt à ce problème.
Par contradiction, supposons que $REGULAR_{TM}$ est décidable et qu'il existe R, un "décideur" pour $REGULAR_{TM}$.
Soit la machine de Turing S telle que S = en entrée $\langle M, w\rangle$:

1. Construire M_2 pour M et w
2. Exécuter R sur M_2 et on accepte si et seulement si R accepte.

En faisant cela, on montre que S décide $A_{TM} \rightarrow$ contradiction

4 Rice's Theorem

Chapitre 5 exercice 28

Posons P comme étant n'importe quelle propriété de langage NON-TRIVIAL (NON-TRIVIAL).

Théorème : Déterminer si le langage d'une machine de Turing a comme propriété P, est indécidable.

$$\{\langle M\rangle | M \text{ est une machine de Turing et } L(M) \text{ a une propriété } P\}$$

Une propriété triviale est une propriété sans importance, par exemple une propriété est triviale si tous les langages ou aucun langage ne l'a.

4.1 Démonstration

Par contradiction :

Posons R_P un "décideur" pour L_P.

— Posons $T_\varnothing$ comme une machine de Turing qui rejette toutes les possibilités ($L(T_\varnothing) = \varnothing$), supposons que $\varnothing \notin P$ (sans perte de généralité).

— Posons T comme une machine de turing tel que $L(T) \in P$.

Prenons, M, w, construit tel que M_w : pour l'entrée x :

1. Simuler M sur w. Si c'est accepté, aller en étape 2. Si c'est rejeté, on rejette.
2. Simuler T sur x, si c'est accepté on accepte, sinon on rejette.

M accepte w est équivalent à dire que $L(M_w) \in P$.

Maintenant nous pouvons donc créer un "décideur" pour A_{TM} tel que :

$S =$ pour une entrée M, w :

1. Construire M_w
2. Exécuter R_p sur M_w et donner la même réponse.

On a donc un "décideur" pour le problème de l'arrêt. Ce qui n'est pas possible. On a donc une contradiction.

INFO-F408: Computability & complexity

Rémy Detobel

16 Octobre, 2017

1 Théorème de Rice

Pour P n'importe quelle propriété "non triviale" d'un langage d'une machine de Turing :

$$\exists M_1 \text{ tel que } L(M_1) \in P$$
$$\wedge \exists M_2 \text{ tel que } L(M_2) \notin P$$

— Problème de décision ($\equiv$ Langages)
— DFA (machine de turing déterministe)
— Non déterministes
— Power set construction : DFA (équivalent à NFA)
— Langage régulier ("Regular languages"), est l'ensemble des langages reconnus par un automate déterministe

— Machines de Turing / Church-T thesis
— Multitape Turing Machine / Machine de Turing non déterministe
— Décidabilité VS reconnaissable ("recognizable") $\Leftrightarrow$ $\exists$ un énumérateur ("enumerator")
Un langage A est :

$$\text{reconnaissable ("recognizable")} \Leftrightarrow \exists M : L(M) = A$$
$$\text{décidable} \Leftrightarrow \exists M : \text{ M est un "decider" } L(M) = A$$

— Cantor's Diagonalization argument :
$\Rightarrow \exists$ langage L $\in$ Reconnaissable
— Problème de l'arrêt (Halting problem)
$A_{TM} = \{\langle M, w\rangle | M \text{ est une machine de Turing qui accepte } w\}$
Reconnaissable et pas décidable $\overline{A_{TM}}$ n'est pas reconnaissable.
— Théorème de Rice.

2 Un problème indécidable simple

Voir livre chapitre 5.2

$$\left\{ \left[\frac{ab}{abab}\right], \left[\frac{b}{a}\right], \left[\frac{aba}{b}\right], \left[\frac{aa}{a}\right] \right\}$$

(Emil) Post correspondence problem
Théorème 5.15 PCP est indécidable (Post correspondence problem)

$$M, w \in \Sigma^* \rightarrow (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$$

Construit une instance P du problème PCP.
P ("has a match") à une correspondance $\Leftrightarrow$ M accepte w.

1. Les premiers dominos sont :

$$\#$$
$$\#q_0 w_1 w_2 ... w_n \#$$

2. Pour chaque $a, b \in \Gamma$ et $q, r \in Q$ où $q \neq q_{reject}$
   Si $\delta(q, a) = (r, b, R)$ on ajoute $\left[\frac{qa}{br}\right]$
3. $a, b, c \in \Gamma; q, r \in Q$
   Si $\delta(q, a) = (r, b, L)$ on ajoute $\left[\frac{cqa}{rcb}\right]$
4. Pour chaque $a \in \Gamma$, on ajoute $\left[\frac{a}{a}\right]$
5. Ajouter : $\left[\frac{\#}{\#}\right]$ et $\left[\frac{\#}{\_\#}\right]$
6. $\left[\frac{a\,q_{accept}}{q_{accept}}\right] \forall a \in \Gamma$ et $\left[\frac{q_{accept}\,a}{q_{accept}}\right]$
7. $\left[\frac{q_{accept}\#\#}{\#}\right]$

Hypothèse : une correspondance doit commencer par le premier domino (Modified PCP)

Exemple :

$>\#$

$>\#q_0 0100\#$

$\delta(q_0, 0) = (q_7, 2, R)$

$\left[\frac{q_0 0}{2 q_7}\right]$

$>\#q_0 0$

$>\#q_0 0100\#2q_7$

$>\#q_0 0100$

$>\#q_0 0100\#2q_7 100$

$>\#q_0 0100\#$

$>\#q_0 0100\#2q_7 100\#$

$\delta(q_7, 1) = (q_1, 3, L)$

$c = 2$

$\left[\frac{2q_7 1}{q_1 23}\right]$

$>\#q_0 0100\#2q_7 100\#$

$>\#q_0 0100\#2q_7 100\#q_1 2300\#$

Comment supprimer la condition de M PCP ? Posons :

$$\circledast w = *u_1 * u_2 * u_3 ... * u_n$$

$$u\circledast = u_1 * u_2 * ...u_n *$$

$$\circledast u\circledast = *u_1 * u_2 * u_3 ... * u_n *$$

On remplace donc :

$$\left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\right\}$$

Par :

$$\left\{\left[\frac{\circledast t_1}{\circledast b_1 \circledast}\right], \left[\frac{\circledast t_2}{b_2 \circledast}\right], \ldots, \left[\frac{\circledast t_k}{b_k \circledast}\right]\right\}$$

Avec cela ils doivent donc être en premier. Mais on doit en plus rajouter :

$$\left[\frac{\circledast \Diamond}{\Diamond}\right].$$

3 Reducibility (Mapping/Many-One)

Définition : $f : \Sigma^* \to \Sigma^*$ est exécutable (computable) si et seulement s'il existe une machine de Turing M telle que pour n'importe quel entrée w, M s'arrête avec seulement $f(w)$ sur son stack/ruban (tape)

Définition : Un langage A est réductible dans un langage B, si il existe une fonction f exécutable (”computable“) telle que

$$\forall w \in \Sigma^* : f(w) \in B \Leftrightarrow w \in A.$$

On note cela : $A \leqslant_M B$

Théorème : Si $A \leqslant_M B$ et B est décidable, alors A est décidable.

Théorème : Si $A \leqslant_M B$ et B est reconnaissable (”recognizable”), alors A est reconnaissable.

INFO-F408: Computability & complexity

Rémy Detobel

23 Octobre, 2017

1 Time Complexity

Cela n'a pas de sens de regarder le temps réel d'exécution car cela va dépendre de la machine qui exécute le code. On va donc plutôt se concentrer sur les étapes exécutées.
On va donc réduire un programme à une fonction prenant en paramètre la taille de l'input, la complexité du paramètre utilisé pour exécuter cette fonction. La complexité du comportement de la fonction va *a priori* varier selon cette entrée. On va donc se concentrer sur la complexité dans le pire des cas.
En d'autres mots :
Nombre d'étape de la machine de Turing $\Rightarrow f : \mathbb{N} \to \mathbb{N}$: nombre maximum d'étapes de la machine de Turing sur une entrée de taille n.
Pour cela, on va utiliser la notation "grand O".

Par exemple : $f(n) = O(g(n))$.
Qui traduit la relation suivante : $\exists n_0 \in \mathbb{N}, c \in \mathbb{R} | \forall n > n_0 : f(n) \leqslant c \cdot g(n)$.

$$f(n) = n$$
$$f(n) = O(n \log n)$$

Dans la même idée, on peut utiliser Ω qui lui utilise donc un $\geqslant$ ($f = O(g) \iff g = \Omega(f)$, et $f = \Theta(g) \iff f = O(g)$ et $f = \Omega(g)$).

Enfin, on peut définir o tel que :

$$f(n) = o(g(n))$$
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

On peut donc écrire :

$$\sqrt{n} = o(n)$$
$$n = o(n \log \log n)$$
$$n^2 = o(n^3)$$

Théorème 7.8 : posons $t(n)$ une fonction telle que $\forall n \in \mathbb{N} : t(n) \geqslant n$. Chaque $t(n)$-time multitape machine de Turing (plusieurs rubans) a un équivalent en temps $\mathbf{O(t^2(n))}$ – **times** avec un seul ruban (*single-tape*) sur la machine de Turing.
Souvenons nous du théorème 3.13 : si l'on simule une machine de Turing à plusieurs rubans sur une machine de Turing à un ruban. $\delta : Q \times \Gamma^K \to Q \times \Gamma^K \times \{L, R\}^K$,
on peut observer plusieurs choses :

1. Pour chaque étape de la machine de Turing à plusieurs ruban, O(1) scan sur le contenu du ruban.
2. le contenu total a une taille $\leqslant K \times t(n)$

Souvenons nous du théorème 3.16 : chaque machine de Turing non déterministe a une équivalence en une machine de Turing déterministe.

Rappel d'une machine de Turing non déterministe : $S : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$
Pour évaluer une machine de Turing non déterministe, on peut construire un arbre qui va associer chacun des choix fait. Le **temps d'exécution** = le nombre d'étapes maximum pour toutes les entrées de taille n et pour tous les chemins calculables.
Notons que ce n'est possible **que** pour les machines **décidables**.
Théorème 7.11 : Chaque $t(n)$-time machine de Turing non déterministe a un équivalent en temps $2^{O(t(n))}$-time machine de Turing déterministe.
On définit b, le nombre maximum de choix possible pour chaque élément dans l'arbre.
On va donc pouvoir dire qu'il y aura :

1. au maximum $b^{t(n}$ nœuds
2. pour chaque nœud, le nombre d'étapes $\leqslant O(t(n))$

Au total : $O(t(n)b^{t(n)}) = 2^{O(t(n))}$ sur une machine de Turing à 3 rubans.
Notons que $(2^{O(t(n))})^2 = 2^{O(t(n))}$.

2 La class P

"Tous les modèles calculables raisonnables sont équivalent en temps polynomial.".
Ici, polynomial est $O(n^K)$ pour une certaine constante K.
$t : \mathbb{N} \to \mathbb{R}^+$ classe de complexité du temps $TIME(t(n))$ est une collection de langage décidables par une TM $O(t(n))$-time.
SIPSER : 7.2

P est une collection de langages décidables dans un temps polynomial sur une machine de Turing déterministe. On note également $P = \bigcup_K TIME(n^k)$

2.1 Exemple

Entrée : le graphe G
Question : est-ce que G est connexe ?
$L = \{\langle G\rangle | G \text{ est un graphe connexe}\}$
$L \in P$

RELPRIME (premier entre eux) $= \{\langle x, y\rangle | x \text{ et } y \text{ sont des nombres premier entre eux}\}$
$(gcd(x, y) = 1)$
Dans l'exécution de l'algorithme d'Euclide : $O(\log x + \log y))$
RELPRIME $\in P$

Il reste cependant des problèmes qui sont encore ouverts. Par exemple, lorsque l'on prend un ensemble de nombres et qu'il faut vérifier s'il existe un sous ensemble dont la somme vaut zéro. C'est un problème en temps exponentiel.

3 La class NP

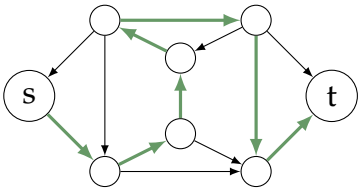
"Non déterministe Polynomial".
Une vérification pour un langage A est un algorithme V tel que :

$$A = \{w | V \text{ accepte } \langle w, c \rangle \text{ pour un texte } c\}.$$

Un temps polynomial vérifie si un vérificateur qui s'exécute en un temps polynomial ($|w|$). A est vérifiable en un temps polynomial $\leftrightarrow \exists$ un vérificateur en un temps polynomial.

3.1 Exemple

Chemin Hamiltonien : pour un graphe dirigé G et 2 sommets/point s, t
Est-ce qu'il existe un chemin depuis s vers t qui passe par tous les sommets une seule fois ?
Exemple :



3.2 Définition

Def 7.19 : NP est la classe de langages qui admettent une vérification en temps polynomial.
Théorème 7.20 Un langage B est dans NP si si et seulement si B est décidable dans une certaine machine de Turing dans un temps polynomial non déterministe.
Preuve
$\Rightarrow$ donner une vérification V en temps polynomial construit en suivant une machine de Turing non déterministe.
N = "Pour l'entrée w de taille n :

1. Prenons c, de manière non déterministe, de taille $\leqslant n^k$
2. Exécuter V sur l'entrée $\langle w, c \rangle$
3. si V est accepté, on accepte, sinon on rejette."

$\Leftarrow$ Supposons que l'on a une machine de Turing N non déterministe qui décide du langage en un temps polynomial. Montrer qu'il existe un vérificateur en un temps polynomial.
V = "Pour l'entrée $\langle w, c \rangle$:

1. Simuler N sur w, et traiter chaque symbole de c comme une description du choix non déterministe pour faire a chaque étape
2. Accepter si la branche est acceptée

"

3.3 Deux définition équivalent pour NP

1. Il existe un décideur non déterministe en un temps polynomial (”poly-time non deterministic decider“)
2. Il existe un vérificateur en un temps polynomial (”polynomial-time verifier“).

3.4 ...

$\text{SUBSET\_SUM} = \{\langle S, t\rangle | S = \{x_1, x_2, ...x_n\}$ et il existe $\{y_1, y_2, ...y_k\} \subseteq \{x_1, ...x_n\}, \Sigma_{i=1}^{k} y_i = t\}$.

Exemple : $\big(\{4, 11, 16, 21, 27\}, 25\big)$ | SUBSET_SUM $\in$ NP ?
Oui, il est bien dans NP.

3.5 $\mathbf{P \stackrel{?}{=} NP}$

Il est prouvé que $P \subseteq NP$, mais on ne sait pas si l'inclusion est stricte ou si $P = NP$ (il est conjecturé que $P \neq NP$).

INFO-F408: Computability & complexity

Rémy Detobel

23 Octobre, 2017

1 P vs NP

Définition

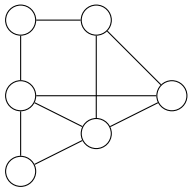
NP est l'ensemble des langages vérifiables en temps polynomial.

Théorème

Un langage appartient à NP $\Leftrightarrow$ il peut être décidé dans une machine de Turing non déterministe en temps polynomial.

$$\text{SUBSET\_SUM} = \left\{ \langle S, t \rangle | S = \{x_1, ..., x_k\} \text{ et il existe } Y \subseteq \{1, ...k\}, \sum_{j \in Y} x_j = t \right\}.$$

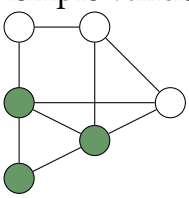
1.1 Problème du CLIQUE



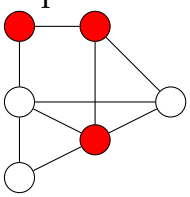
Une clique est un sous-graphe d'un graphe non-dirigé formant un graphe complet (toute paire de nœuds est relié par exactement une arête). Le problème CLIQUE peut dont être définit comme étant :

$$\text{CLIQUE} = \{\langle G, K \rangle | G \text{ est un graphe non dirigé avec une k-clique}\}.$$

Exemple valide :



Exemple d'ensemble ne répondant pas au problème :



Définition

Une fonction $f : \Sigma^* \to \Sigma^*$ est calculable en temps polynomial s'il existe une machine de Turing qui s'arrête avec seulement $f(w)$ sur son ruban pour une entrée w et s'exécute en un temps polynomial.

Définition

Un langage A est réductible en temps polynomial dans un langage B, noté $A \leqslant_p B$ s'il existe une fonction f s'exécutant en un temps polynomial telle que :

$$\forall w \in \Sigma^* : w \in A \Leftrightarrow f(w) \in B.$$

Important

Si $A \leqslant_p B$ et $B \in P$, alors $A \in P$.

1.1.1 Exemple : Problème de 3-SAT

$$\begin{aligned}\phi =&(x_1 \vee x_1 \vee x_2) \wedge \\ &(\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge \\ &(\overline{x_1} \vee x_2 \vee x_2)\end{aligned}$$

Tel que

$$\begin{aligned}x_1 &\leftarrow F \\ x_2 &\leftarrow T\end{aligned}$$

3-CNF formule (il s'agit d'une forme normal conjonctive)

$$3\ SAT = \{\phi | \phi \text{ est une formule booléenne satisfaisant 3-CNF }\}$$

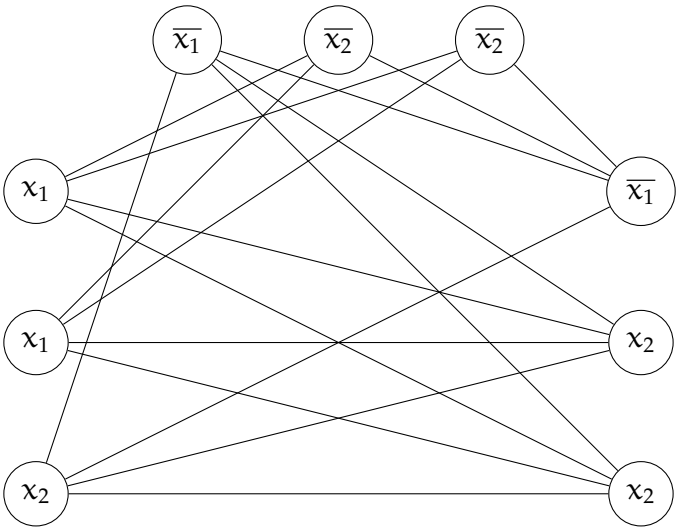
3 SAT $\in$ NP

Théorème

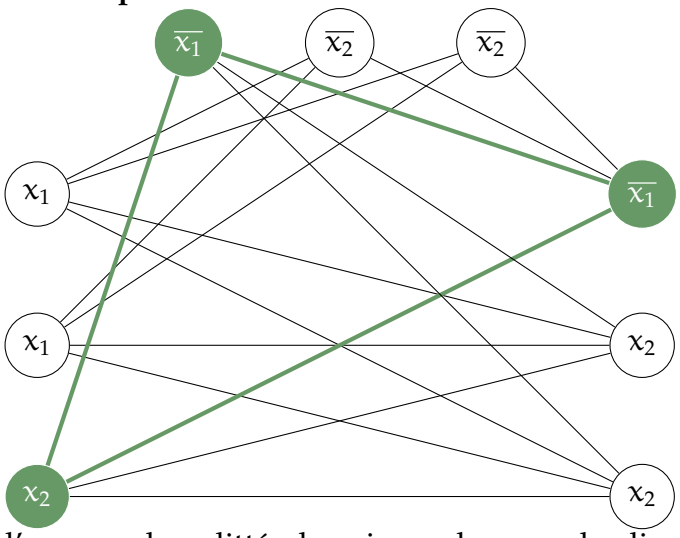
3 SAT $\leqslant_p$ CLIQUE

1.1.2 Passage de 3-SAT à CLIQUE

On va donc devoir trouver un algorithme pouvant transformer un problème "3 SAT" en un problème de type "K-CLIQUE" en un temps polynomial. Cela permettra donc de résoudre "3 SAT" avec seulement un algorithme de résolution de type "K-CLIQUE".
Un "litéral" est simplement une variable ou son inverse (ex : x_i ou $\overline{x_i}$).



K = 3, il s'agit donc du nombre de clause.
On peut donc écrire que si ϕ est satisfait, alors G a une k-clique.
Une solution pourrait être :



$$\phi = (x_1 \vee x_1 \vee \mathbf{x_2}) \wedge (\overline{\mathbf{x_1}} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{\mathbf{x_1}} \vee x_2 \vee x_2)$$

Si l'on prend un littéral vrai sur chacune des lignes, on aura une solution au k-clique.

2 NP-Compleet

Définition

Un langage B est "NP-complet" si et seulement si :

1. $B \in NP$
2. Pour chaque $A \in NP, A \leqslant_p B$

2.1 Cook-Levin Théorème

Définissons SAT comme étant un problème retournant un booléen pour toutes les formules sous forme normal conjonctive.
SAT est NP-Complet.

1. Si B est NP-complet et $B \in P$, alors $P = NP$
2. Si B est NP-complet et $B \leqslant_p C$ (pour $C \in NP$), alors C est NP-Complet.

On peut donc écrire par exemple :

$$\begin{aligned} &\text{Si } A \leqslant_p B \\ \wedge &B \leqslant_p C \\ \Rightarrow &A \leqslant_p C \end{aligned}$$

Définition

Un langage B est NP-Hard (NP-Dur ou NP-Difficile) si et seulement si :
pour tout $A \in NP, A \leqslant_p B$

Considérons $A \in NP$. Il existe donc une machine de turing "N" non déterministe qui peut décider du langage A en un temps polynomial. Supposons en temps $\leqslant n^k$ pour un certain $k = O(1)$

$$\#q_0 w_1 w_2 ... w_n \_ \_ ... \_ \_ \#$$
$$\#a q_1 w_2 ... w_n \_ \_ ... \_ \_ \#$$
$$...$$

Où on peut compter n^k lignes sur une longueur de n^k

$$C = Q \cup \Gamma \cup \{\#\}$$

Variables : $X_{i,j,s} | i, j = 1...n^k, S \in C$

$$\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

$$\phi_{start} = X_{1,1,\#} \wedge X_{1,2,q_0} \wedge X_{1,3,w_1} \wedge ... \wedge X_{1,n+2,w_n} \wedge X_{1,n+3,\_} \wedge ... \wedge X_{1,n^k-1,\_} \wedge X_{1,n^k,\#}$$

$$\phi_{cell} = \bigwedge_{1 \leqslant i,j \leqslant n^k} \left[\left(\bigvee_{S \in C} X_{i,j,s} \right) \wedge \bigwedge_{s,t \in C | s \neq t} \left(\overline{X_{i,j,s}} \vee \overline{X_{i,j,t}} \right) \right]$$

$$\phi_{accept} = \bigvee_{1 \leqslant i,j \leqslant n^k} X_{i,j,q_{accept}}$$

$$\textbf{ffi}_{\textbf{move}} = \bigwedge_{i,j} \left[\text{où } (i,j) \text{ windows est légal} \right]$$

↑ "Use Legal Windows", il s'agit d'une fenêtre de 2 (ligne) par 3 (colonne) placé partout dans le tableau

Où une fenêtre légal est définit par :

$$\bigvee_{a_1, a_2, ... a_6 \text{ est une fenêtre légal}} \left(X_{i,j-1,a_1} \wedge X_{i,j,a_2} \wedge X_{i,j+1,a_3} \wedge ... \right)$$

	j-1	j	j+1
i	a_1	a_2	a_3
i+1	a_4	a_5	a_6

#	q_0	w_0
#	a	q_1

est légal.
$(q_1, a, R) \in \delta(q_0, w_1)$

a	b	c
a	d	e

$d \neq b$ et $e \neq c$ et $\in \Gamma$ n'est pas légal

INFO-F408: Computability & complexity

Rémy Detobel

13 novembre, 2017

1 Cook-Leving

Rappel : SAT est NP-Complet.
Ce qui signifie que $A \in NP$ et $\exists$ une machine de Turing non déterministe N
$\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$

1.1 Exemple

$$\delta(q_1, a) = \{(q_1, b, R)\}$$

Qui signifie que si on lit a sur le ruban, on écrit b en q_1 et on se déplace vers la droite. On a donc ici une machine non déterministe, donc un ensemble de srotie, mais dans les faits cet ensemble n'a qu'un seul élément. Cette transition est donc déterministe.

$$\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$$

Si on lit b sur le tape, on peut soit écrite c sur q_2 et se déplacer vers la gauche ou aller à droite en écrivant a sur q_2.

Tableaux :

a	q_1	b
q_2	a	c

Ce tableau est légal/valide

b	b	b
c	b	b

Ce tableau est légal/valide

#	b	a
#	b	a

Ce tableau est légal/valide

a	b	a
a	a	a

Ce tableau n'est **pas** valide

a	q_1	b
q_2	a	a

Ce tableau n'est **pas** valide

a	q_1	a
a	b	a

Ce tableau n'est **pas** valide

$$* = \bigvee_{a_1, a_2, \ldots, a_6 \text{est une fenêtre légal}} (X_{i,j-1,a_1} \wedge X_{i,j,a_2} \wedge X_{i,j+1,a_3} \wedge X_{i+1,j,a_4} \wedge \ldots \wedge X_{i+1,j+1,a_6})$$

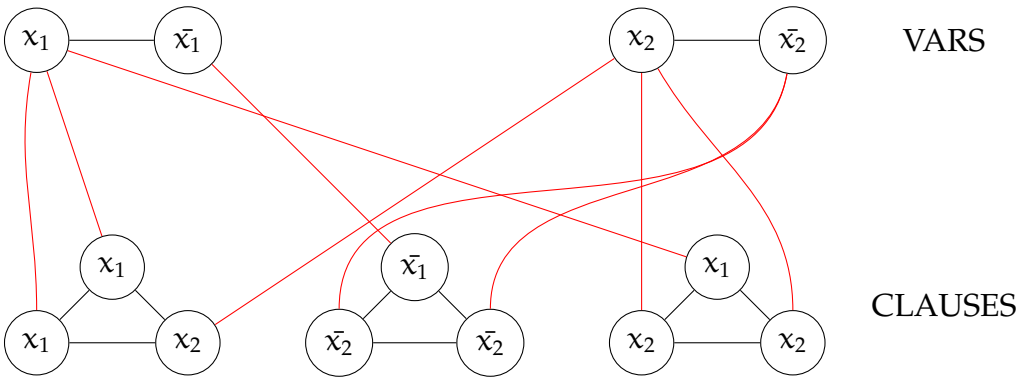
	j-1	j	j+1
i	a_1	a_2	a_3
i+1	a_4	a_5	a_6

Si ϕ as une taille polynomiale, alors ϕ est satisfaisable, de manière équivalente, $\Leftrightarrow$ *w* est accepté par N.

2 3-SAT

Forme normale conjonctive se note "CNF" en anglais. La formule de la FNC-4 s'écrit :

$$\bigwedge_j (l_{i_1,j} \vee l_{i_2,j} \vee l_{i_3,j})$$



On appelle donc ce qu'il y a dans le grand "ET" une clause, et chaque élément de cette close s'appelle un littéral et est soit une variable, soit la négation d'une variable.

2.1 3 SAT est NP-Complet

— ϕ peut être écrit en FNC (seulement ϕ_{move} doit être transformé)
— Maintenant les clauses ne doivent pas avoir une taille de 3. Pourquoi faire ?
Supposons une clause : $(a_1 \vee a_2 \vee ... \vee a_l)$

$$(a_1 \vee a_2 \vee Z_1) \wedge$$
$$(\overline{Z_1} \vee a_3 \vee Z_2) \wedge$$
$$(\overline{Z_2} \vee a_4 \vee Z_3) \wedge$$
$$... \wedge$$
$$(\overline{Z_{l-3}} \vee a_{l-1} \vee a_l)$$

Exemple $a_4 = T, a_{i \neq 4} = F$

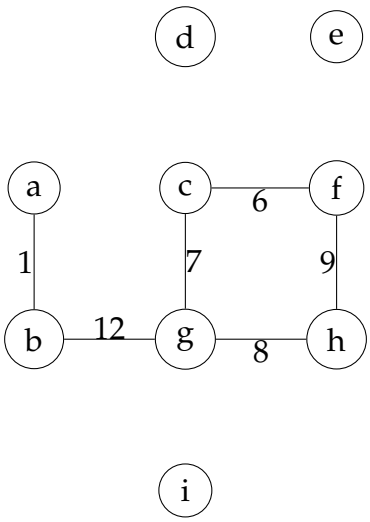
$$(a_1 \vee a_2 \vee Z_1 = T) \wedge$$
$$(\overline{Z_1} \vee a_3 \vee Z_2 = T) \wedge$$
$$(\overline{Z_2} \vee a_4 = T \vee Z_3 = F) \wedge$$
$$(\overline{Z_3} \vee a_5 \vee Z_4 = F) \wedge$$
$$(\overline{Z_4} ...)$$

3 CLIQUE est NP-Complet

3.1 Vertex Cover

$3\,SAT \leqslant_p Vertex\ Cover$
K-Vertex cover :

$$\phi = (X_1 \vee x_1 \vee x_2) \wedge$$
$$(\bar{x_1} \vee \bar{x_2} \vee \bar{x_2}) \wedge$$
$$(x_1 \vee x_2 \vee x_2)$$



Une solution est de mettre $x_1 = T$ et $x_2 = F$

4 Hamiltonian Path

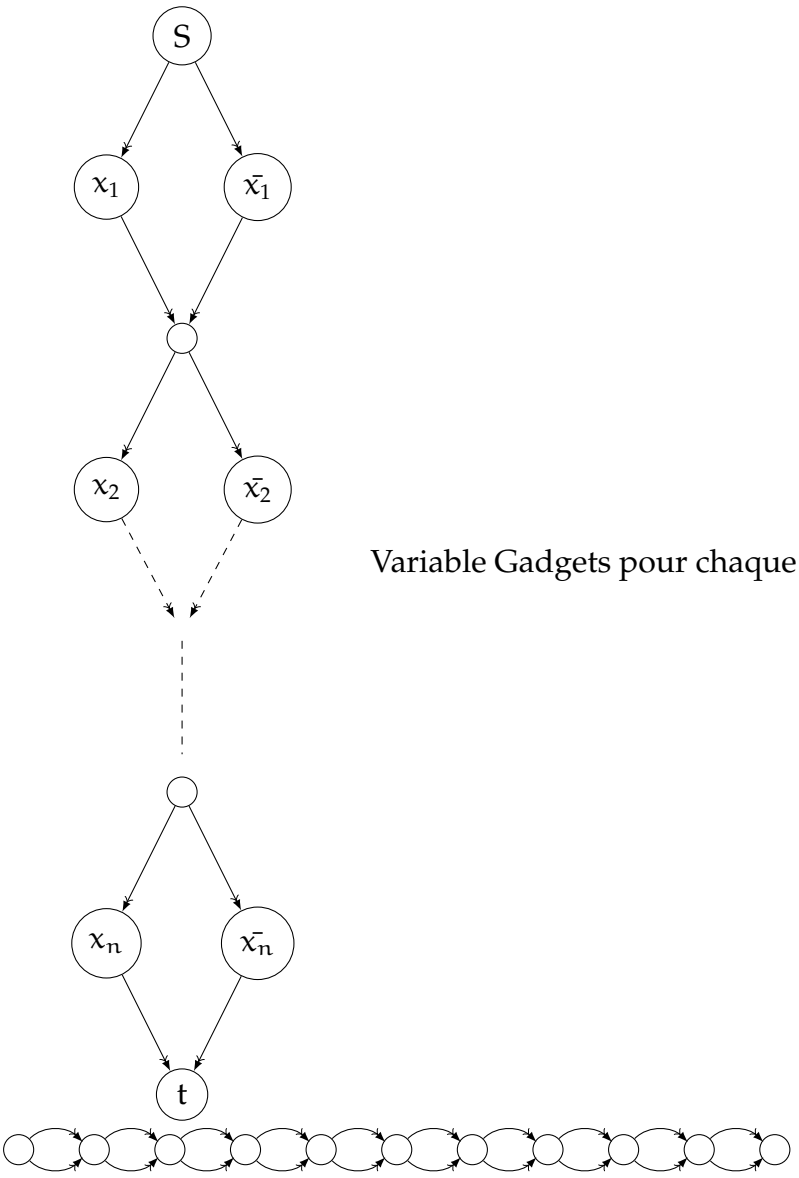
Hamiltonian VS Eulerian
Euler : "Bridge of Königsberg"

$\texttt{HamPath} = \{\langle G, S, t\rangle | G \text{ est un graphe dirigé avec un chemin Hamiltonien de S a t}\}$.

Théorème

Hamiltonian Path est NP-Complet

Pour une formule 3-FNC ϕ, on peut construire G, s, t tels que ϕ est SAT $\Leftrightarrow$ G a un chemin Hamiltonien de S à t.



Variable Gadgets pour chaque variable x_i

"Hard" direction : $\exists$ un chemin Hamiltonien $\Rightarrow$ $\exists$ une assignation (valuation) au problème de SAT.

INFO-F408: Computability & complexity

Rémy Detobel

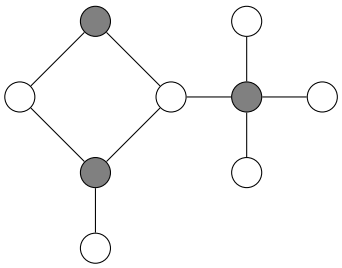
27 novembre, 2017

1 NP-Complete problems

— 3-SAT
— CLIQUE
— INDEPENDENT SET (complément de CLIQUE) $\equiv$ CLIQUE in $\overline{G}$
— VERTEX COVER
— HAMILTONIAN PATH
— SUBSET SUM

1.1 K-Independant Set in G

$\equiv$ CLIQUE in $\overline{G}$
$\equiv (n - K)$ VERTEX COVER in G



Vertex cover

Lemme :
Pour $G = (V, E)$
Si $S \leqslant V$ est un vertex cover, alors V_S est un INDEPENDANT SET

1.2 SUBSET SUM

$$= \Big\{ < S, t > | S = \{s_1, s_2, ...s_k\} \\ \exists \{y_1, ...y_l\} \subseteq S \\ \text{tel que} \sum_{s \in S} s = t \Big\}$$

Théorème

SUBSET-SUM est NP-Complet

Preuve :
Réduction depuis 3 SAT où $x_1...x_v$ sont des variables et où $c_1..c_n$ sont des clauses (conditions).

		1	2	3	...	v		1	2	3	...	m
x_1	y_1	1	0	0	0	0			1	1		
	z_1	1	0	0	0	0						
x_2	y_2	0	1	0	0	0			1	0		
	z_2	0	1	0	0	0				1		
x_3	y_3	0	0	1	0	0			1			
	z_3	0	0	1	0	0						
*	g_1	0	0	0	0	0		0	0	0	*	0
	h_i	0	0	0	0	0		0	0	0	*	0
t		1	1	1	1	1		3	3	3	3	3

$$C_2 = (x_1 \vee x_2 \vee x_3)$$
$$C_3 = (x_1 \vee \overline{x_2} \vee x_4)$$

C_3 contient $\overline{X_2}$
C_2 contient $\overline{X_1}$

*Pour chaque clause/colonne c_i, on inclut deux fois le nombre : 0...010...0 (où le 1 est placé en index i).

1. Supposons qu'il existe une instance de SAT :
   $\rightarrow$ construisons s' comme ceci : $\forall i = 1...V$ si $x_i = T$ prenons y_i dans s', z_i dans les autres cas.
   Jusqu'à ce que cela satisfasse chaque clause où chaque 1, 2 ou 3 littéraux sont égaux à "True".
   $\forall i = 1...m$:
   — Si c_i a 1 littéral à vrai qui inclut g_i et h_i dans S'
   — Si c_i a 2 littéraux à vrai qui inclut g_i seulement.
2. Supposons qu'il existe $S' \subseteq S$ tel que $\sum S' = t$. L'assignation est construite telle que :
   $X_i = T \Leftrightarrow y_i \in S'(= F$ dans les autres cas$)\forall i = 1...V$
   Et ceci **n'est pas une assignation valide** pour SAT.

2 Programmation dynamique

Algorithme pour SUBSET-SUM
x[i] est un nombre dans S et t est la somme a calculer.

Table T
$T[i][j] = True \Leftrightarrow$ il existe un sous-ensemble du premier nombre i où la somme est j
$i = 1...|S|$
$j = 0... \sum_{x \in S} x$

Algorithme
Initialisons T. Pour $i = 1...|S|, j = 0...t$:
$T[i][j] \leftarrow T[i-1][j] \vee T[i-1][j - x[i]]$
Return$T[|S|][t]$.
n a la taille de l'entrée (par exemple : le nombre de bits).
La complexité de l'algorithme DP pour le problème SUBSET SUM : $|S| \times t$

$$n \simeq \left(\sum_{x_i \in S} \log_2 x_i \right) + \log_2 t$$

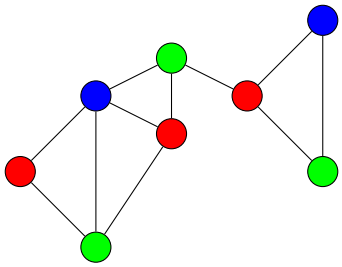
2.1 Exemple

$$S = \{5\ 000\ 000\ 000, 939\ 000\ 000, 333\ 212\}$$
$$t = 5\ 939\ 333\ 212$$

Complexité : $n \leqslant 4 \times 10 \times 4 = 160$
$|S| \times t \geqslant 4 \times 5 \times 10^9$

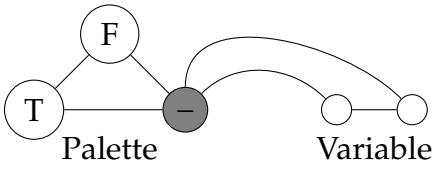
3 3-COLORING

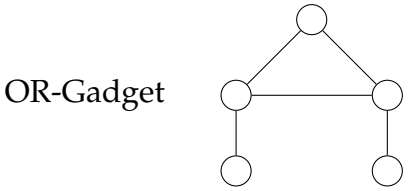
Prenons un simple graphe non dirigé G, peut-on colorier ses points en 3 couleurs tel que deux point adjacents n'ont pas la même couleur ?



3.1 Exercice 7.27

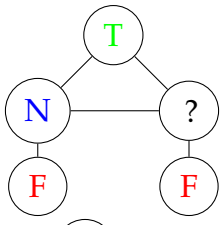
Prouver que 3-COLORING est NP-Complet



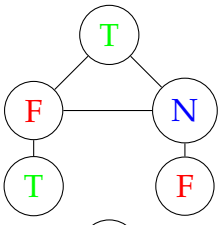


Il y a plusieurs manière de représenter le “OR-GADGET” :

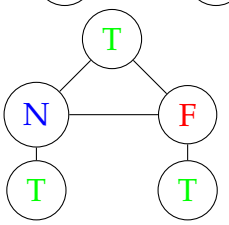
Pas OK



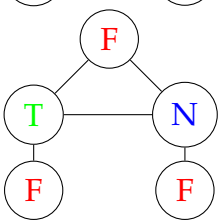
OK



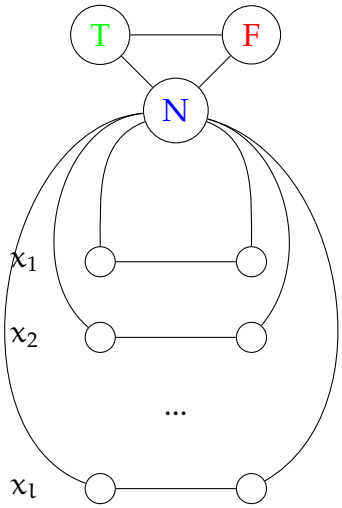
OK



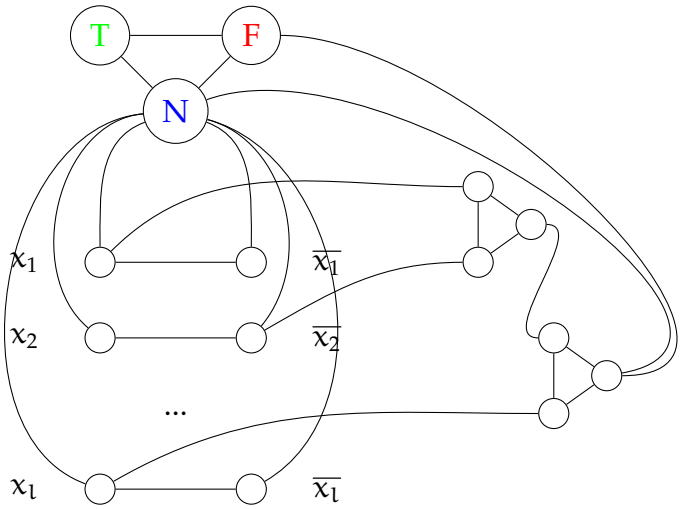
OK



Réduction depuis 3 SAT : 3 CNF formule ϕ, variables : $x_1, ...x_l$:
Preuve :

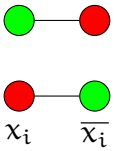


Par exemple : $(x_1 \vee \overline{x_2} \vee x_l)$



$\exists 3 - \texttt{coloring} \Leftrightarrow y$ est satisfaisable.

1. Supposons que ϕ est satisfaisable :
   $\exists$ un SAT associé a $f : \{x_1, ...x_n\} \to \{T, F\}$. Colorer la palette en rouge vert bleu comme indiqué.

   $\forall$ varaible x_i : colorier la variable gadget et :
   Si $f(x_i) = T$ dans les autres cas
   $\forall$ "OR-GADGET" : utiliser la couleur A ou B pour le "second" ou gadgets et colorer A, B ou C pour le "premier".
2. Supposons $\exists$ un 3-COLORING colorer en rouge (F) vert (T) bleu (N).
   Mettre x_i a vrai si :  et a faux si 

INFO-F408: Computability & complexity

Rémy Detobel

4 décembre, 2017

1 Chapitre 8 : SPACE Complexity

Pour une machine de Turing, fonction : $f : \mathbb{N} \to \mathbb{N}$.
Taille maximum d'une case scannée par une machine de Turing sur chaque input de taille n.

Machine de Turing non déterministe : $* +$ maximise le calcule de chaque branche

$$\begin{aligned} SPACE(f(n)) = \{&L|L \text{ un langage décidable par une machine} \\ &\text{de turing déterministe en } O(f(n)) - space\} \end{aligned}$$

$$\begin{aligned} NSPACE(f(n)) = \{&L|L \text{ un langage décidable par une machine} \\ &\text{de turing } \textbf{non} \text{ déterministe en } O(f(n)) - space\} \end{aligned}$$

Exemple : $SAT \in SPACE(n)$

1.1 Théorème de Savitch

Théorème 8.5

Théorème

Pour toute fonction $f : \mathbb{N} \to \mathbb{N}$ telle que $\forall n \in \mathbb{N} : f(n) \geqslant n$:
$NSPACE(f(n)) \subseteq SPACE(f^2(n))$, où $f^2(n) = (f(n))^2$, à ne pas confondre avec $f(f(n))$.

Preuve du théorème de Savitch :
Machine de turing non déterministe N décidable pour A en $f(n)space$
Algorithme Canyield sur l'input : c_1, c_2 (configurations), t (nombre d'étapes).

Question Peut-on aller de la configuration c_1 à la configuration c_2 avec un maximum de t étapes.
Supposons sans perte de généralité, qu'il n'existe qu'une seule configuration acceptante (supprimer le contenu du tape et déplacer la tête sur la première case).

$$CANYIELD(c_{start}, c_{accept}, 2^{df(n)}) \text{ simule N}$$

1. Si $t = 1$, est-ce que N peut atteindre c_2 depuis c_1 en une étape. (cfr "legal windows")
2. Dans les autres cas, pour chaque configuration c_m de N en utilisant $space\ f(n)$:
   (a) exécuter $CANYIELD(c_1, c_m, t/2)$
   (b) exécuter $CANYIELD(c_m, c_2, t/2)$
   (c) Accepter si les deux acceptent
3. Si pas accepté, alors on rejette

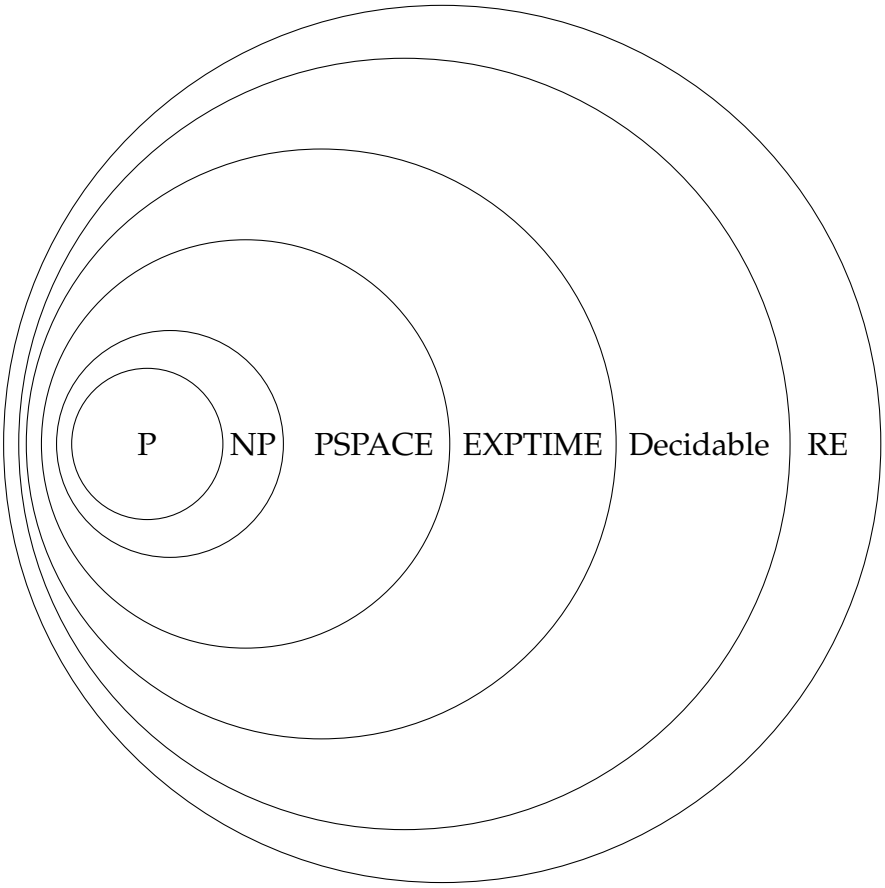
Remarque : t ne peut pas être impaire car il est égal à une puissance de 2 (à savoir $2^{df(n))}$)

Le nombre maximum d'appel récursif est égal à $\log_2 t$, donc $\log_2 2^{df(n)} = O(f(n))$
L'espace requis pour chaque appel récursif est plus au moins égal à $c_m \leqslant f(n))$
$\Rightarrow O(f^2(n))$

1.2 PSPACE

$$PSPACE = \cup_k SPACE(n^k)$$

On ne sait pas si NP est strictement inclut dans PSPACE ou pas. Par contre on est sur que $P \neq EXPTime$



Cfr : Complexité de Zoo

$$NPSPACE = \cup_k NSPACE(n^k)$$

Par le théorème de Savitch, on peut écrire : $PSPACE = NPSPACE$.

Complémentarité de PSPACE :
Un langage B est PSPACE-complet si et seulement si :

1. $B \in PSPACE$
2. $\forall A \in PSPACE, A \leqslant_p B$ ("PSPACE-Hard")

NB : Une instance $\phi(x)$ (qui est une formule booléen) du problème SAT $\Leftrightarrow$ est-ce que cette fonction est valide : $\exists x : \phi(x)$?

TRUE QUANTIFIED BOOLEAN FORMULE (TQBF)

Exemple : $\exists x | \forall y : \exists z | \forall t : \phi(x, y, z, t)$

$TQBF = \{\langle\phi\rangle | \phi$ est vrai et complètement quantifié comme étant une formule booléen$\}$

$TQBF \in NP$, la réponse n'est pas définie clairement, mais on peut facilement montrer par un exemple qu'il est dans NP.

Théorème

TQBF est PSPACE Complet

1. $TQBF \in PSPACE$
2. TQBF est PSPACE-Dur ($\forall A \in PSPACE, A \leqslant_p TQBF$).
   Pour M décidable sur A dans n^k space, nous ajoutons un input w à une formule booléenne totalement quantifié (Fully quantified Boolean formula) qui est vrai si et seulement si w est accepté par M.
   Rappelons nous l'encodage de la configuration de "string" en variable booléens dans le théorème de Cook-Levin : $X_{i,s}$ = "Symbole S est trouvé à la position i" [a].

a. Chaque configuration peut être encodé avec $O(n^k)$ variables booléen

$\phi_{c_1,c_2,t} \Leftrightarrow$ M peut aller de la configuration c_1 à la configuration c_2 avec au maximum t étapes.

Pour $t = 1$: cfr "legal windows"

Échauffement : $\phi_{c_1,c_2,t} = \exists m_1 \left[\phi_{c_1,m_1,t/2} \wedge \phi_{m_1,c_2,t/2}\right]$ Faire ceci revient à séparer en deux le problème et donc cela devient exponentiel.

Meilleure idée : $\phi_{c_1,c_2,t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\}$
$[\phi_{c_3,c_4,t/2}]$

NB : $\forall x \in \{y, z\}$ peut être remplacé par $\forall x \left[(x = y \vee x = z) \rightarrow ...\right]$

Observation :

— A chaque niveau de la récursion, on rajoute une partie de taille $O(f(n)) = O(n^k)$

— Le nombre de niveau est égal à $\log_2 t$

$$\phi_{c_{start},c_{accept},2^{(df(n)=n^k)}} \text{ a une taille de } O(f^2(k)) = O(n^{2k})$$