INGI1123
Calculabilité et Complexité

Notes de cours collaboratives

Edited by

Yves Deville

Unversité catholique de Louvain

11 février 2018

Ce document est un syllabus collaboratif pour le cours *INGI1123 Calculabilité et complexité* donné à l'Université catholique de Louvain par Yves Deville. Ces notes sont en cours d'élaboration et sont le fruit d'un travail collaboratif entre les étudiants de ce cours, avec une supervision de Yves Deville et différents chercheurs et assistants.

Ce document se base sur la *Synthèse de Calculabilité Q6 - LINGI1123*, produite notamment par Floran Hachez, Lionel Nobel Lena Peschke et François Robinet lorsqu'ils ont suivi ce cours. Cette synthèse a été rédigée sous licence Creative Commons CC-BY-SA et est disponible à l'adresse suivante `https://github.com/Gp2mv3/Syntheses`

Table des matières

Chapitre 1

Introduction

1.1 Qu'est-ce que la calculabilité

La calculabilité c'est l'étude des limites de l'informatique. Il faut bien faire attention à faire la différence entre les limites théoriques et les limites pratiques. Pour la calculabilité, on s'occupe des limites théoriques. Alors que pour la complexité on s'occupe des limites pratiques. La complexité détermine la frontière entre ce qui est faisable en pratique et infaisable en pratique. La question principale de la calculabilité est : "quels sont les problèmes qui peuvent être résolus par un programme ?". La caractéristique de calculabilité ne donne aucune autre information sur le programme que son existence.

Le but est donc de tracer des frontières entre les programmes calculables, non calculables et non calculables en pratique.

Cela nous permet de savoir quand ça ne sert à rien d'essayer de résoudre un problème. De plus, on est conscient de sa complexité intrinsèque d'un problème.

1.2 Notion de problème

Premièrement, on doit parler la notion de problème. Attention, il ne faut pas confondre un problème avec un programme. Les caractéristiques d'un problème sont :

— un problème est générique : il s'applique à un ensemble de données.

— pour chaque donnée particulière, il existe une réponse.

On représente un problème dans le cours par une fonction. Donc dans le cours, la description d'un problème est équivalente à la description d'une fonction.

1.3 Notion de programme

Un programme est une "procédure effective", c'est-à-dire exécutable par une machine. Il existe plein de formalismes permettant la description de "procédure effective".

1.4 Résultats principaux

1.4.1 Équivalence des langages de programmation

Il a t-il de langage de programmation qui sont plus puissantes que d'autres ? Il a des langages qui sont équivalents ?
Il y a une équivalence entre langages en termes de calculabilité quand un problème qui peut être résolu par un de ces langages peut être résolu par n'importe lequel autre. S'il existe un programme Java qui résoudre un problème il existera aussi un programme C/C++, PHP, ... qui peut résoudre le même problème, on peut dire qu'il existe une équivalence théorique.
D'un point de vue théorique ce langages s'appellent des langages complets, complet ça veut dire qu'il permet tous de résoudre le même problème donc le problème calculable à ces langages sont tous le même.
D'un point de vue pratique on peut voir des différences (le programme sera plus court, s'écrire plus rapidement, le programme sera plus rapide, plus propre, plus fiable et encore d'autres critères) c'est le langage le plus adapté qui aura un avantage.

1.4.2 Existence de problèmes non calculables

Problème non calculable : il existe des problèmes qui ne peuvent être résolus par un programme. Ex : détection de virus, équivalence de programme, déterminer si un polynôme à coefficients entiers à des racines entières, ...

Détection de virus

On veut déterminer si un programme P avec une entrée D est nuisible.

Être nuisible : Un programme est dit nuisible si son exécution a pour effet de contaminer d'autres programme, le programme va se recopier autre part.

Spécification du programme detecteur(P,D) :
Préconditions : un programme P et une donnée D
Postconditions : "Mauvais" si P(D) est nuisible, "Bon" sinon. Il faut aussi que detecteur ne soit pas nuisible.

On va créer un programme drole(P) et essayer de détecter s'il est nuisible.

```
drole(P)
if detecteur(P,P) = ``Mauvais''
        then stop
else infecter un autre programme en y inserant P
```

Testons `drole(drole)`.

```
drole(drole)
if detecteur(drole, drole) = ``Mauvais''
        then stop
else infecter un autre programme en y inserant drole
```

— Supposons que `drole(drole)` soit nuisible. Lorsqu'on exécute, `drole(drole)` `detecteur(drole,drole)` n'infecte rien car `detecteur` n'est pas nuisible. Comme `detecteur` retourne "Mauvais", le programme s'arrête. Rien a donc été infecté, ce qui est contradictoire avec le fait que `drole(drole)` est nuisible.

— Si par contre il n'est pas nuisible alors `detecteur(drole,drole)` ne va pas retourner Mauvais et on va arriver dans le **`else`**. On a donc infecter un autre programe que qui contredit le fait que `drole(drole)` n'est pas nuisible.

On a donc une contradiction dans tous les cas ce qui implique que le programme `drole` ne peut exister, ce qui implique que le programme detecteur non plus.

1.4.3 Existence de problèmes intrinsèquement complexes

Problème intrinsèquement complexe. (Voir complexité) Les problèmes qui ont une complexité supérieure ou égale à l'exponentielle. Peu importe les évolutions technologiques, un problème exponentiel ne peut et ne pourra être résolu que pour de petite taille, par exemple le problème du voyageur de commerce.

Chapitre 2

Concepts

2.1 Ensembles, langages, relations et fonctions

2.1.1 Ensembles

Un ensemble est une collection d'objets, sans répétition, appelés les éléments de l'ensemble.

Notation :

— Ensemble fini : $\{0, 1, 2\}$
— Ensemble infini : $\{0, 1, 2, \ldots\}$
— Produit cartésien : $A \times B$
— Le nombre d'éléments : $|A|$
— Ensemble des sous-ensembles : 2^A ou $\mathcal{P}(A)$, e.g. $2^{\{2,4\}} = \{\{\}, \{2\}, \{4\}, \{2, 4\}\}$. On peut remarquer que $|2^A| = 2^{|A|}$
— Complément : $\overline{A}$

2.1.2 Langages

Notation :

— une *chaîne de caractère* ou un *mot* : séquence FINIE de symboles. abceced, 010101101
— chaîne de caractères vide : ϵ
— un *alphabet* Σ est un ensemble de symboles. $\Sigma = \{1, 2\}$
— un *langage* est un ensemble de mots constitués de symboles d'un alphabet donné.
— ensemble de tous les mots possibles sur l'alphabet Σ : Σ^*

2.1.3 Relations

Soient A, B des ensembles.

— Une *relation* R sur A, B est un sous-ensemble de $A \times B$. C'est-à-dire un ensemble de paires $\langle a, b\rangle$ avec $a \in A$, $b \in B$.
— On peut définir une relation par sa table
— On peut écrire $\langle a, b\rangle \in R$ ou aRb ou $R(a, b)$.

2.1.4 Fonctions

Soient A, B des ensembles.

— Une *fonction* $f : A \to B$ est une relation telle que pour tout $a \in A$, il existe au plus un $b \in B$ tel que $\langle a, b\rangle \in f$

— écrire $f(a) = b$ est équivalent à $\langle a, b\rangle \in f$

— Si il n'existe pas de $b \in B$ tel que $f(a) = b$ alors $f(a)$ est indéfini, $f(a) = \perp$

Soit $f : A \to B$, on définit le *domain* et l'*image* respectivement comme suit

$$\begin{aligned} \mathrm{dom}(f) &= \{\, a \in A \mid f(a) \neq \perp \,\}, \\ \mathrm{image}(f) &= \{\, b \in B \mid \exists a \in A : b = f(a) \,\}. \end{aligned}$$

Si $\mathrm{dom}(f) \subseteq A$, f est appelée *partielle* et si $\mathrm{dom}(f) = A$, f est appelée *totale*. Notez qu'avec cette définition, une fonction totale est partielle. Pour dire que $\mathrm{dom}(f) \subset A$, c'est plus explicite de dire que f n'est pas totale que dire que f est partielle.

Une fonction est *surjective* si $\mathrm{image}(f) = B$ et *injective* si $\forall a, a' \in A$, $a \neq a' \Rightarrow f(a) \neq f(a')$.

Une fonction est *bijective* si elle est totale, injective et surjective.

On utilise aussi le concept d'*extension* : $f : A \to B$ est une extension de $g : A \to B$ si $\forall x \in A$: $g(x) \neq \perp \Rightarrow f(x) = g(x)$. Autrement dit, f a la même valeur que g partout où g est définie.

Définition d'une fonction On définit une fonction par sa table qui peut-être infinie.
On peut définir la table de plusieurs façons :

— Par un texte fini déterminant sans contradiction ni ambigüité le contenu de la table.

— Par un algorithme ex : $f(x) = 2x^3 + 5$

— Écrire toutes les paires de la relation.

Attention, il n'est pas nécessaire de décrire ou de connaître un moyen de la calculer pour pouvoir la définir. Ex : $f(x) = 1$ s'il y a de la vie autre part que sur terre, 0 sinon.

2.2 Ensemble énumérable

Avant de dire ce qu'est un ensemble énumérable, on doit savoir que deux ensembles ont le même cardinal s'il existe une bijection entre eux.

Définition 1 (Ensemble énumérable)**.** *Un ensemble est énumérable ou dénombrable s'il est fini ou s'il a le même cardinal que* $\mathbb{N}$.
Quelques propriétés :

Propriétés 2. *Tout sous-ensemble d'un ensemble énumérable est énumérable.*

Propriétés 3. *L'union et l'intersection de deux ensembles énumérables sont énumérables.*

Propriétés 4. *L'union d'une infinité énumérable d'ensembles énumérables est énumérable.*

Démonstration. La démonstration est similaire à la démonstration de l'énumérabilité de $\mathbb{Q}$. On met chaque ensemble en ligne, il y a un nombre énumérable de lignes qu'on peut numéroter en parcourant le tableau en zigzag en partant de $(0, 0)$. □

L'union d'une infinité non-énumérable d'ensembles énumérable peut ne pas être énumérable. Par exemple, l'union des singletons $\{x\}$ pour tout réel x forme l'ensemble des réels $\mathbb{R}$ qui n'est pas énumérable :

$$\bigcup_{x\in\mathbb{R}} \{x\} = \mathbb{R}.$$

Remarque 5. *Une bonne intuition à avoir : Tout ensemble dont les éléments peuvent être représentés de manière finie est énumérable.*

Par exemple, les rationnels, même s'ils ont une représentation décimale infinie peuvent être représentés de manière finie en fraction d'entiers.

Quelques ensembles non énumérables :

Exemple 6. *L'ensemble* $\mathbb{R}$

Démonstration. Voir ci-dessous. □

Exemple 7. *L'ensemble des sous-ensembles de* $\mathbb{N}$, $\mathcal{P}(\mathbb{N})$

Démonstration. Rappelons nous tout d'abord que comme $\mathbb{N}$ est infini, ses sous-ensemble peuvent l'être aussi. Il est adéquat de visualiser un ensemble à l'aide d'un mot binaire où le bit i vaut 1 si le $i$ième élément est pris dans le sous-ensemble. Par exemple, le sous-ensemble $\{1, 4\}$ de $\{1, 3, 4\}$ peut être représenté par 101. Seulement, comme il y a un nombre infini de nombre entiers, on a mot binaire infini. Par exemple, les nombres pairs, c'est le mot $10101010101\ldots$:

```
0123456789...
1010101010... nombres pairs

0123456789...
0011010100... nombres premiers

0123456789...
0001001001... multiples de 3 non-nuls
```

On voit maintenant la bijection entre les sous-ensembles de $\mathbb{N}$ et $[0, 1]$. En effet, on peut associer par exemple l'ensembe des nombres pairs au réel $0.101010101\ldots$. Plus précisément on associe le sous-ensemble de $\mathbb{N}$ représenté par le mot m, le réel entre 0 et 1 dont l'écriture binaire est $0.m$. □

Exemple 8. *L'ensemble des chaînes infinies de caractère sur un alphabet fini*

Démonstration. On utilise le même raisonnement que pour les sous-ensembles de $\mathbb{N}$ sauf que pour un alphabet de k symboles, on utilise la représentation des réels en base k. □

Exemple 9. *L'ensemble des fonctions de* $\mathbb{N}$ *dans* $\mathbb{N}$ *(Cas important)*

Démonstration. Si c'était dénombrable, soit $f_0, f_1, f_2, \ldots$ leur dénombrement. On construit le tableau

$$\begin{array}{ccccc} f_0(0) & f_0(1) & f_0(2) & f_0(3) & \cdots \\ f_1(0) & f_1(1) & f_1(2) & f_1(3) & \cdots \\ f_2(0) & f_2(1) & f_2(2) & f_2(3) & \cdots \\ \vdots & \ddots & \ddots & \ddots & \ddots \end{array}$$

et on conclut par diagonalisation de façon semblable à $\mathbb{R}$ en construisant $d\colon \mathbb{N} \to \mathbb{N}$ tel que $d(k) = f_k(k)$. □

2.3 Cantor

On va montrer qu'il existe des ensembles non énumérables par diagonalisation. Ex $\mathbb{R}$.

Exemple 10. *Exemple de démonstration par diagonalisation :*

1. *Construire une table : Liste de tous les grands mathématiciens*
   ***D**E MORGAN*
   *A**B**EL*
   *BO**O**LE*
   *BRO**U**WER*
   *SIER**P**INSKI*
   *WEIERS**S**TRASS*
2. *Sélectionner la diagonale : $diag$ =DBOUPS*
3. *Modifier l'élément égal à la diagonale : $diag'$ = CANTOR*
4. *Montrer que l'élément n'est pas dans la liste ⇒ Contradiction*
5. *Conclusion :*
   - *Soit on sait que la liste est complète*
     ⇒ CANTOR n'est pas un grand mathématicien (cas utilisé pour démontrer halt).
   - *Soit on sait que CANTOR est un grand mathématicien*
     ⇒ la liste est incomplète (cas utilisé pour la diagonalisation de CANTOR)

Théorème 11 (Diagonalisation de Cantor)**.** *Soit $E = \{x \text{ réel } |0 < x \leq 1\}$, E est non énumérable.*

Démonstration : On va montrer qu'un nombre d' n'est pas dans l'énumération alors qu'on sait que d' est un nombre réel compris entre 0 et 1.

On suppose E énumérable. Donc il existe une énumération des éléments de E, $x_0, x_1, \ldots, X_k, \ldots$. On peut représenter un nombre x_k comme étant une suite de chiffre x_{ki} : $x_k = 0.x_{k0}x_{k1}\ldots x_{kk}\ldots$.

1. On peut donc construire une table infinie :

	1 digit	2 digit	3 digit	...	k+1 digit	...
x_0	x_{00}	x_{01}	x_{02}	...	x_{0k}	...
x_1	x_{10}	x_{11}	x_{12}	...	x_{1k}	...
x_2	x_{20}	x_{21}	x_{22}	...	x_{2k}	...
:	:	:	:	:	:	:
x_k	x_{k0}	x_{k1}	x_{k2}	...	x_{kk}	...
:	:	:	:	:	:	:

2. Sélection de la diagonale (celle-ci est un nombre réel compris entre 0 et 1)
$$d = 0.x_{00}x_{11}...x_{kk}...$$
3. Modification de cet élément d pour obtenir
$$d' = 0.x'_{00}x'_{11}...x'_{kk}...$$
Où $x'_{ii} = 5$ si $x'_{ii} \neq 5$
$x'_{ii} = 6$ si $x'_{ii} = 5$
On a toujours que cet élément d' est compris entre 0 et 1
4. Contradiction, car d' est dans l'énumération, car E est énumérable (par supposition). Il existe donc $x_p = d'$,
$$d' = x_p = 0.x_{p0}x_{p1}...x_{pp}...$$
$$d' = 0.x'_{00}x'_{11}...x'_{pp}...$$
La contradiction vient du fait qu'on a choisi $x'_{pp} \neq x_{pp}$. Donc $d' \neq x_p$ ce qui implique que d' n'est pas dans l'énumération.
5. Conclusion : E n'est pas énumérable

2.4 Conclusion

Les ensembles énumérables sont importants pour la suite du cours et aussi, car en informatique on ne considère que les ensembles énumérables. Dans le cours, on va souvent devoir montrer qu'un ensemble est énumérable/non énumérable. Généralement on va utiliser une des techniques suivantes :

— montrer qu'il y a une bijection avec $\mathbb{N}$ ou $\mathbb{R}$
— montrer que l'ensemble est fini
— utiliser la diagonalisation (cf. Cantor)
— écrire un programme qui énumère l'ensemble

Chapitre 3

Résultats fondamentaux

3.1 Algorithmes et effectivité

Qu'est-ce qu'un algorithme ?

Définition 12 (Algorithme)**.** *C'est une procédure qui peut être appliquée à n'importe quelle donnée et qui a pour effet de produire un résultat. C'est un ensemble fini d'instructions qui peuvent être exécutées. Dans cette partie du cours, on ne prend pas la taille des données, des instructions ni de la mémoire disponible en compte, mais on les considère comme finies.*

Remarque 13. *Un algorithme n'est pas une fonction, mais un algorithme calcule une fonction. De plus dans le cours on se limite aux fonctions de $\mathbb{N}^n$ dans $\mathbb{N}$. Car on peut, montrer que ça revient au même que de considérer de $\mathbb{N}^n$ dans $\mathbb{N}^n$ ($\mathbb{N}^n$ est énumérable et donc au plus de même cardinal que $\mathbb{N}$). On va aussi utiliser Java comme modèle étant donné que c'est plus facile et qu'on va montrer que les modèles complets sont équivalents.*

3.2 Fonctions calculables, ensembles récursifs et récursivement énumérables

3.2.1 Fonction calculable

Définition 14 (Fonction calculable)**.** *Une fonction f est calculable s'il existe un algorithme qui, recevant comme donnée n'importe quels nombres naturels $x_1, \ldots, x_n$ fournit **tôt ou tard** comme résultat $f(x)$ s'il existe.*

S'il ne se termine pas c'est que $f(x) = \perp$.

Remarque 15. *Il faut faire la distinction entre la non-existence d'un algorithme et ne pas être capable de le trouver. (Voir exemple TP et cours : y a-t-il une rose verte sur Mars ou encore x occurrences de 5 dans π).*

Remarque 16. *Une fonction peut-être totale calculable ou partielle calculable.*

3.2.2 Ensemble récursif et récursivement énumérable

Soit $A \subseteq \mathbb{N}$

Définition 17 (Ensemble récursif)**.** *Un ensemble A est récursif s'il existe un algorithme qui recevant un $x \in \mathbb{N}$, fournit **tôt ou tard** comme résultat* $\begin{matrix} 1 \text{ si } x \in A \\ 0 \text{ si } x \notin A \end{matrix}$ *. L'algorithme décide si x est dans A ou non.*

Définition 18 (Ensemble récursivement énumérable)**.** *Un ensemble A est récursivement énumérable ($\in RE$) s'il existe un algorithme qui recevant un $x \in \mathbb{N}$,*

- *fourni **tôt ou tard** comme résultat 1 si $x \in A$.*
- *ne se termine pas ou retourne un résultat $\neq 1$ si $x \notin A$.*

Définition 19 (Ensemble co-récursivement énumérable)**.** *Un ensemble A est* co-récursivement énumérable *si son complément $\bar{A} = \mathbb{N} \setminus A$ est récursivement énumérable.*

Remarque 20. *Le souci lorsque c'est récursivement énumérable, c'est que même si l'algorithme permet de dire si x est dans A, si x n'est pas dans A on ne sait rien dire. En effet, on ne peut pas dire à un moment qu'on arrête l'algorithme et que l'élément n'est pas dans A. Car il est possible que x soit dans A et que l'algorithme ne l'ait pas encore déterminé (l'algorithme retourne tôt ou tard un résultat si x est dans A !).*

Définition importante :

Définition 21 (Fonction caractéristique)**.** *Fonction caractéristique de A : $X_A : \mathbb{N} \to 0, 1$, tel que*

$$X_A(x) = \begin{cases} 1 & si\ x \in A \\ 0 & si\ x \notin A \end{cases}$$

Propriétés 22. *A est un ensemble récursif ssi X_A est une fonction totale calculable.*

On remarque en effet que X_A décide si x appartient à A ou non.

Propriétés 23. *A est un ensemble récursivement énumérable ssi $A = \mathrm{dom}(f)$ et f est une fonction (totale ou partielle) calculable. En effet, tout ensemble récursif est récursivement énumérable (Propriété 25 ci-dessous).*

Démonstration. Comme f est calculable, il existe un programme P_f qui calcule f. On peut alors construire un programme Q qui avec input x, appelle P_f. Si P_f ne se termine pas($P_f(x) = \perp$) alors Q non plus ($Q(x) = \perp$) mais si P_f termine, alors Q renvoie 1. □

Propriétés 24. *A est un ensemble récursivement énumérable ssi*

- *A est vide ou*
- *$A = \mathrm{image}(f)$ et f est une fonction totale calculable.*

Démonstration. f est une fonction d'énumération (elle peut répéter plusieurs fois un élément, mais ce n'est pas gênant). À l'aide d'un programme P_f qui calcule f, on appelle $P_f(0), P_f(1), P_f(2), \ldots$ jusqu'à ce que $P_f(k) = x$ où on retourne 1. Si $x \in \mathrm{image}(f)$, on retournera 1 tôt ou tard. Sinon, on ne terminera jamais, ce qui équivaut à retourner $\perp$. □

Propriétés importantes

Propriétés 25. *A récursif $\Rightarrow$ A récursivement énumérable. (Propriété plus faible)*

Propriétés 26. *A récursif $\Rightarrow$ ($\mathbb{N} \setminus A$) récursif.*
On peut facilement créer un programme qui retourne 1 - le programme qui décide A.

Propriétés 27. *Si A est récursivement énumérable et co-récursivement énumérable ($\overline{A}$ recursivement énumérable) alors A est récursif.*

Démonstration. Il suffit de créer un programme Q qui, pour décider récursivement si $x \in A$, appelle le programme qui décide A et le programme qui décide $\overline{A}$ dans deux threads et renvoie 1 si $x \in A$ et 0 si $xin\overline{A}$. En effet, si A et $\overline{A}$ sont récursivement énumérables, un des deux appels finira tôt ou tard par dire si x appartient à A ou non. □

Propriétés 28. *A fini ⇒ A récursif.*

Démonstration. Si l'ensemble est $a_0, \ldots, a_n$, le programme suivant marche

if $x = a_0$ **then**
  print 1
else if $x = a_1$ **then**
  print 1
  ...
else if $x = a_n$ **then**
  print 1
else
  print 0
end if

□

Propriétés 29. *($\mathbb{N} \setminus A$) fini ⇒ A récursif. (Trivial)*

Exemple 30. *Existe-t-il une fonction f qui renvoie 1 s'il existe au moins x occurrences successives de 5 dans π.*

Oui, c'est soit
print 1

s'il y a une infinité de 5 dans π, soit
if** $x \leq a$ **then
  print 1
else
  print 0
end if

si il y a exactement a occurrences de 5 and π.

Si on remplace "au moins" par "exactement", est-ce toujours calculable ? Oui, il suffit de remplacer $\leq$ par $=$.

Propriétés 31. *Tout programme qui a un nombre fini d'input différent est calculable (ça ne marche pas s'il est infini car le taille du code du programme doit être fini donc on ne peut pas tout hardcoder).*

3.3 Thèse de Church-Turing

Grâce à la définition, on sait montrer qu'une fonction est calculable. Mais comment montrer qu'une fonction n'est pas calculable ? Pour ça on a besoin d'une définition couvrant la totalité des fonctions calculables.

Une autre question est : "Est-ce que prendre un modèle particulier est restrictif ?"

La thèse de Turing répond à ces questions.

1. Aucun modèle de la notion de fonction calculable n'est plus puissant que les Machines de Turing. Version moderne : Une fonction est calculable s'il existe un programme d'ordinateur qui calcule cette fonction.
2. Toute fonction calculable est calculable par une machine de Turing.
3. Toutes les définitions formelles de la calculabilité connues à ce jour sont équivalentes (Théorème, ça a été démontré) (justifie l'utilisation de Java comme modèle).
4. Toutes les formalisations de la calculabilité établies par la suite seront équivalentes aux définitions connues

1, 2 et 4 sont des thèses universellement reconnues comme vraies et la 3 est un théorème.

3.4 Programmes et fonctions

Dans le cours on utilise un langage de programmation, Java, comme modèle.

Définition 32 (P)**.** *Soit P l'ensemble des programmes Java qui reçoit un ou plusieurs entiers comme donnée(s) et qui imprime/retourne un résultat.*

Propriétés 33. *P est un ensemble infini dénombrable, car c'est chaîne de caractère d'un alphabet fini. P est récursif, car il existe un programme, le compilateur, qui détermine si un programme est un programme Java ou non.*

Définition 34 (Énumération de P)**.** *P = P_0, P_1,... ,P_k,... est l'énumération des programmes Java sans répétition. Ce qui implique qu'on peut numéroter les programmes Java.*

Définition 35 (P_k)**.** *P_k est le programme numéro k dans P*

Définition 36 ($\phi_k^{(n)}$)**.** *$\phi_k^{(n)} : \mathbb{N}^n \to \mathbb{N}$ est la fonction calculée par P_k*

Remarque 37. *$P_{12} \neq P_{47}$ mais ça ne veut* pas *dire que $\phi_{12} \neq \phi_{47}$. En effet, il existe une infinité de manière de coder une fonction.*

Propriétés 38. *Il existe donc une fonction $f : \mathbb{N} \to P$ telle que :*

- *$f(k) = P_k$*
- *f calculable*
- *k (numéro d'un programme) et P_k sont deux représentations distinctes d'un même objet.*

3.5 Existence de fonctions non calculables

Il existe beaucoup de fonctions non calculables, car le nombre de fonctions de $\mathbb{N}$ dans $\mathbb{N}$ est non dénombrable (Exemple 9). Or le nombre de programmes Java est dénombrable. Donc il y a beaucoup de fonctions qui ne sont pas calculables.

On ne va s'intéresser qu'aux fonctions qui sont définies par une table finie ou une table infinie que l'on peut décrire de façon finie (on n'est pas capable d'écrire une définition infinie). Il en existe une infinité dénombrable.

On va maintenant montrer que ce n'est pas parce que la table d'une fonction est définie de manière finie que la fonction est nécessairement calculable (exemple la fonction halt qui détermine si un programme se termine ou non).

3.6 Problème de l'arrêt

Définition 39 (halt)**.** *halt est la fonction : P x $\mathbb{N} \to \mathbb{N}$ telle que*
halt(n, x) = 1 si $P_n(x)$ se termine
halt(n, x) = 0 sinon
ce qui équivaut à
halt(n, x) = 1 si $\phi_n(x) \neq \perp$
halt(n, x) = 0 sinon

Propriétés 40. *halt est une fonction bien définie, totale et sa table est infinie, mais décrite de manière finie. Or on va montrer que halt n'est pas calculable par diagonalisation (comme pour la démonstration de Cantor).*

Remarque 41. *Attention, juste dire qu'on ne sait pas écrire un programme qui calcule halt ne prouve pas qu'on ne sait pas calculer halt. En effet, comme vu dans le chapitre 1, prouver que le programme existe est différent de savoir écrire le programme.*

Théorème 42 (halt)**.** *halt n'est pas calculable*

Démonstration : On suppose halt calculable.

1. On peut donc construire une table infinie définissant la fonction halt :

	0	1	2	...	k	...
p_0	$halt(0,0)$	$halt(0,1)$	$halt(0,2)$	...	$halt(0,k)$	...
p_1	$halt(1,0)$	$halt(1,1)$	$halt(1,2)$	...	$halt(1,k)$	...
p_2	$halt(2,0)$	$halt(2,1)$	$halt(2,2)$	...	$halt(2,k)$	...
:	:	:	:	:	:	:
p_k	$halt(k,0)$	$halt(k,1)$	$halt(k,2)$	...	$halt(k,k)$	...
:	:	:	:	:	:	:

2. Sélection de la diagonale

$$diag : halt(0,0), halt(1,1), ..., halt(k,k), ...$$

$$diag(n) = halt(n,n)$$

3. Modification de cet élément diag pour obtenir $diag'(n) = \begin{matrix} 1 \text{ si halt(n,n) = 0} \\ \perp \text{ si halt(n,n) = 1} \end{matrix}$
   $diag'$ est calculable, car halt est calculable (il y a moyen d'écrire un programme qui calcul $diag'$ en utilisant $diag$)
4. Contradiction :
   Donc il existe un programme P_d qui calcule $diag'$: $diag'(d) = \begin{matrix} 1 \text{ si halt(d,d) = 0} \\ \perp \text{ si halt(d,d) = 1} \end{matrix}$ Mais,
   - Si $diag'(d) = 1$
     $\Rightarrow$ $halt(d,d) = 0$
     $\Rightarrow$ $P_d(d)$ ne se termine donc pas
     $\Rightarrow$ $diag'(d)$ ne se termine pas
     $\Rightarrow$ $diag'(d) = \perp$ or on a supposé que $diag'(d) = 1$
     $\Rightarrow$ Contradiction.
   - Si $diag'(d) = \perp$
     $\Rightarrow$ $halt(d,d) = 1$
     $\Rightarrow$ $P_d(d)$ se termine
     $\Rightarrow$ $diag'(d)$ termine
     $\Rightarrow$ $diag'(d) = 1$ or on a supposé que $diag'(d) = \perp$
     $\Rightarrow$ Contradiction.
5. Conclusion : $diag'$ n'est pas calculable $\Rightarrow$ $diag$ n'est pas calculable $\Rightarrow$ halt n'est pas calculable.

Conclusion Il n'existe pas d'algorithme qui détermine si n'importe quel programme P_n se termine ou non. Mais dans certains formalismes qui ne sont pas des modèles complets la fonction halt de ce formalisme est calculable. Par exemple un langage qui ne permet de calculer que des fonctions totales (exemple java sans aucune boucle), halt est calculable (halt retourne toujours 1).

De plus, il faut faire attention, car ce n'est pas parce que halt n'est pas calculable que pour un programme donné k, $halt(k,x)$ est non calculable. Par exemple $halt(32,123)$ est une fonction constante donc calculable (ce n'est pas pour autant qu'on est capable d'écrire l'algorithme), $halt(32,x)$ peut-être calculable, mais ça dépend du programme 32, par exemple si celui-ci est constant.

Remarque 43. *Cette partie du cours est très importante, car on a trouvé un "trou dans le mur des fonctions calculables" et on va étendre le trou (métaphore de Mr Deville). C'est-à-dire que maintenant on va utiliser halt pour montrer que d'autres fonctions sont calculables/non calculables. Par exemple par réduction par rapport à halt : si on montre qu'une fonction équivalente à halt est calculable, alors halt est calculable.*

Il existe donc au moins un ensemble non récursif K.

Définition 44 (K). $K = \begin{matrix} \{n \mid (n,n) \in HALT\} \\ \{n \mid halt(n,n)=1\} \\ \{n \mid P_n(n) \text{ se termine}\} \end{matrix}$ *K est donc l'ensemble des programmes P_n qui se terminent en recevant comme entrée leur numéro de programme.*

Définition 45 (HALT). *HALT est l'ensemble des programmes k et des valeurs x tel que le programme k se termine avec l'entrée x.*

$$HALT = \{(n,x) | P_n(x) \text{se termine}\}$$

Propriétés 46. *K et HALT ne sont pas récursifs*

Propriétés 47. *K et HALT sont récursivement énumérables (car il suffit de lancer le programme et s'il retourne quelque chose il se termine sinon on ne peut rien dire).*

Propriétés 48. *$\overline{HALT}$ n'est pas récursivement énumérable sinon HALT serait récursif. En effet, on ne peut pas coder un programme qui dirait si un autre programme k, appelé avec la valeur x, ne terminera jamais.*

Propriétés 49. *$\overline{K}$ n'est pas récursivement énumérable*

Voir schéma sur les fonctions et ensembles dans le cours.

3.7 Insuffisance des fonctions totales

Comme on a vu qu'il était possible de calculer halt dans un modèle qui ne permet de calculer que des fonctions totales. On se demande pourquoi on n'utilise pas un tel modèle pour lequel la fonction halt est calculable (pour un tel modèle halt retourne toujours 1. De plus les fonctions "pratiques" sont des fonctions totales. Mais on va montrer qu'un tel modèle nous restreint beaucoup sur ce qu'on peut calculer.

Exemple 50. *Un tel modèle est par exemple BLOOP (défini plus loin) ou encore Java, mais sans boucle donc par intuition on se doute qu'il y a beaucoup de choses qu'on ne peut pas calculer.*

Définition 51. *Posons Q un langage (**non trivial**) dont tous les programmes se terminent (ils ne calculent que des fonctions totales) et pour lequel il existe un interpréteur calculable, interpret(n,x) = ϕ'_n.*

Définition 52. *ϕ'_k est la fonction calculée par le programme Q_k.*

Propriétés 53. *L'interpréteur est une fonction totale.*

Théorème 54 (Hoare-Allison). *interpret(n,x) n'est pas calculable dans Q.*

Démonstration : On suppose interpret calculable.

1. On peut donc construire une table infinie définissant la fonction interpret :

	0	1	2	...	k	...
Q_0	$interpret(0,0)$	$interpret(0,1)$	$interpret(0,2)$	...	$interpret(0,k)$	...
Q_1	$interpret(1,0)$	$interpret(1,1)$	$interpret(1,2)$	...	$interpret(1,k)$	...
Q_2	$interpret(2,0)$	$interpret(2,1)$	$interpret(2,2)$	...	$interpret(2,k)$	...
:	:	:	:	:	:	:
Q_k	$interpret(k,0)$	$interpret(k,1)$	$interpret(k,2)$	...	$interpret(k,k)$	...
:	:	:	:	:	:	:

2. Sélection de la diagonale
$$diag(n) = interpret(n, n)$$
3. Modification de cet élément diag pour obtenir $diag'(n) = interpret(n, n) + 1$ $diag'$ est calculable, car interpret est calculable dans Q (il y a moyen d'écrire un programme qui calcule $diag'$ en utilisant $diag$)
4. Contradiction :
Donc il existe un programme Q_d qui calcule $diag'(d) = interpret(d, d)+1$ (par construction). Mais, par définition $diag'(d) = \phi_d(d) = interpret(d, d)$. En effet, calculer $\phi_d(d)$ revient à interpréter le programme d avec la donnée d.
5. Conclusion : $diag'$ n'est pas calculable $\Rightarrow$ $diag$ n'est pas calculable $\Rightarrow$ interpret n'est pas calculable dans Q.

Remarque 55. *Si les programmes de Q ne calculaient pas que des fonctions totales, alors ça n'aurait pas posé de problèmes, car $interpret(d, d)$ ne se serait pas terminé or un programme qui ne se termine pas + quelque chose, reste un programme qui ne se termine pas.*

3.7.1 Implication du théorème 54, Hoare Allison

Propriétés 56. *Si un langage (**non trivial**) ne permet que le calcul de fonctions totales, alors :*

— *l'interpréteur de ce langage n'est pas calculable dans ce langage.*
— *il existe des fonctions totales non programmables dans ce langage.*
— *ce langage est* **restrictif.**

Propriétés 57. *Si on peut programmer l'interpréteur d'un langage L dans L alors il est impossible de programmer la fonction halt dans L car celle-ci n'est pas calculable.*

Propriétés 58. *Dans un langage de programmation, il est donc impossible que l'interpréteur et la fonction halt puissent être programmés dans ce langage.*

Propriétés 59. *Si on veut pouvoir programmer toutes les fonctions totales dans un langage, le langage doit permettre la programmation de fonctions non totales.*

Propriétés 60. *L'ensemble $\{n \mid \phi_n$ est totale$\}$ n'est pas récursif. (sinon on saurait créer un langage qui calcule toutes les fonctions totales et uniquement les fonctions totales).*

Théorème 61 (fonction universelle)**.** *La fonction universelle (c'est-à-dire l'interpréteur) est $\theta(n, x)$ tel que :*
$$\theta(n, x) = \phi_n(x)$$
est calculable, c'est à dire qu'il existe z tel que P_z calcule l'interpréteur universel, c'est à dire $\phi_z = \theta$.

3.8 Extension de fonctions partielles

Théorème 62. *Il existe une fonction partielle calculable g telle qu'aucune fonction totale calculable n'est une extension de g.*

Démonstration. C'est le cas de diag' avec le tableau $\mathrm{interpret}(i, j)$ (le même que pour la preuve avec Q). S'il existe une extension calculable $f = \phi_s$ de diag', alors si $\mathrm{interpret}(s, s) = \perp$, $\phi_s(s) = \mathrm{diag}'(s) \neq \perp = \mathrm{interpret}(s, s)$, c'est une contradiction et si $\mathrm{interpret}(s, s) \neq \perp$, $\phi_s(s) = \mathrm{diag}'(s) = \mathrm{diag}(s) + 1 = \mathrm{interpret}(s, s) + 1 \neq \mathrm{interpret}(s, s)$, c'est une contradiction.

On en conclut que la fonction universelle n'est pas calculable, ce qui contredit le théorème 61. □

Remarque 63. *Ca implique que si on a une fonction partielle g, il n'est pas toujours possible de créer une fonction totale f qui étend g tel que en dehors du domaine de g, f retourne un code d'erreur. En effet, si cette fonction f existait, halt serait calculable.*

3.9 Théorème de Rice

Soit A $\subseteq \mathbb{N}$

Théorème 64 (Rice). *Si A récursif et $A \neq \emptyset$ et $A \neq \mathbb{N}$*
Alors $\exists i \in A$ et $\exists j \in \mathbb{N} \setminus A$ tels que $\phi_i = \phi_j$

On utilise le plus souvent le théorème de Rice en ayant recourt à sa contraposée :

Théorème 65 (Rice (contraposée)). *Si $\forall i \in A$ et $\forall j \in \overline{A}$ on a $\phi_i \neq \phi_j$*
Alors A non-récursif ou $A = \emptyset$ ou $A = \mathbb{N}$

Dans de nombreux cas, on peut garantir $A \neq \emptyset$ et $A \neq \mathbb{N}$, ce qui permet de se servir de la contraposée pour démontrer qu'un ensemble est non-récursif sans utiliser la preuve par diagonalisation ou par réduction.

Démonstration. On a $\forall i \in A$ et $\forall j \in \overline{A}$ tel que $\phi_i \neq \phi_j$
Supposons A récursif, $A \neq \emptyset$ et $A \neq \mathbb{N}$
On va montrer que HALT est récursif, car on peut construire un programme qui décide HALT, alors qu'on sait que HALT n'est pas récursif.

1. Construisons le programme P_k

```
while true do;
```

   $\forall x\ \phi_k(x) = \perp$
2. $\overline{A} \neq \emptyset$ car $A \neq \mathbb{N}$, supposons $k \in \overline{A}$ (hypothèse sans importance, car montrer que A ou $\overline{A}$ est non récursif revient au même, car A non récursif $\Leftrightarrow$ $\overline{A}$ non récursif)
3. $A \neq \emptyset$ par hypothèse, supposons $m \in A$
4. $\phi_i \neq \phi_j$ et ce $\forall i \in A, \forall j \in \overline{A}$ par hypothèse donc $\phi_m \neq \phi_k$
5. Construisons un programme $P(z)$ qui calcule la fonction $g(z)$, $P(z) \equiv$

```
P_n(x);
P_m(z);
```

   Remarque 66. *On sait que si un programme calcule une fonction $f(x) = \perp\ \forall x$ alors ce programme doit être dans $\overline{A}$ car $f(x) = \phi_k(x)$.*
   Or
   - *soit notre programme $P(z)$ ne se termine pas $\forall z$, car $P_n(x)$ ne se termine pas et donc $g(z) = \phi_k(z)$.*
   - *soit notre programme $P(z)$ calcule $\phi_m(z)$ or on sait que $m \in A$.*

   Donc en résumé si $P_n(x)$ ne se termine pas $P(z)$ sera dans $\overline{A}$ sinon $P(z)$ sera dans A.

   Un programme qui décide HALT $halt(n, x) \equiv$

```
construire P(z);
d <- numero de P(z);
if d in A then print(1);
else print(0);
```

6. Contradiction, car HALT n'est pas récursif.
7. Conclusion : A n'est pas récursif.

□

On est intéressé par l'analyse des propriétés d'un programme. Est-ce que ces propriétés peuvent être déterminées par un algorithme ? On va utiliser le théorème de Rice pour différencier les propriétés qu'il est possible de déterminer par un algorithme de celles pour lesquelles ce n'est pas possible.

Pour ça, considérons A comme étant l'ensemble des programmes qui respectent une propriété. Par exemple $A_1 = \{i | \phi_i \text{ est total}\}$ ou $A_2 = \{i | \phi_i = f\}$,... (il y a plein d'exemples dans le cours).

Analyse

— Si la propriété est vérifiée par certains programmes, mais pas tous, et qu'elle est décidable, alors il existe deux programmes calculant la même fonction, mais l'un vérifie la propriété, et l'autre pas.

— Si la propriété est vérifiée par un programme, mais pas tous, alors cette propriété ne peut-être décidée pas un algorithme.

— S'il existe un algorithme permettant de déterminer si un programme quelconque calcule une fonction ayant cette propriété, alors toutes les fonctions calculables ont cette propriété ou aucune fonction calculable n'a cette propriété.

— Aucune question relative aux programmes, vu sous l'angle de la fonction qu'ils calculent, ne peut être décidée par l'application d'un algorithme.

— Les propriétés intéressantes d'un programme concernant la fonction qu'il calcule, non pas sa forme, sont non calculables.

3.10 Théorème de la paramétrisation

3.10.1 Transformation de programmes

Définition 67 (Transformateur de programme)**.** *On peut voir une fonction $f : \mathbb{N} \to \mathbb{N}$ comme une fonction qui prend le numéro d'un programme et qui retourne le numéro d'un autre programme $f : P \to P$. Donc on peut voir P_k, le programme qui calcule f comme un transformateur de programme. En effet, P_k prend un programme en entrée et retourne un programme qui peut être différent.*

Théorème 68 (S-m-n)**.** *Pour tout $m, n \geq 0$,*
il existe une fonction totale calculable $S_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$
telle que pour tout k (ϕ_k est calculable),

$$\phi_k^{(n+m)}(x_1, ..., x_n, x_{n+1}, ..., x_{n+m}) = \phi^{(n)}_{S_n^m(k, x_{n+1}, ..., x_{n+m})}(x_1, ..., x_n)$$

Démonstration. Pour prouver que S_n^m est totale calculable, on va montrer comment construire un programme qui calcule $S_n^m(k, x_{n+1}, ..., x_{n+m})$.
Tout d'abord, comme ϕ_k est calculable, il existe un programme $P_k(x_1, x_2, ..., x_{n+m})$.
On peut donc construire un programme $Q(x_1, ..., x_n)$ qui calcule $P_k(x_1, x_2, ..., x_{n+m})$.
$x_1, ..., x_n$ restent des arguments du programme, tandis que $x_{n+1}, ..., x_{n+m}$ deviennent des valeurs fixées.
Notre programme qui calcule $S_n^m(k, x_{n+1}, ..., x_{n+m})$ n'a plus qu'à retourner le numéro du programme Q.
En résumé, $S_n^m(k, x_{n+1}, ..., x_{n+m}) \equiv$

```
construire Q(x1,x2,...,xn) = P_k(x1,x2,...xn,...,xn+m);
d <- numero du programme Q;
print(d);
```

□

Théorème 69 (S). *C'est la propriété S-m-n affaiblie.*

$$\forall\, k\ \exists\, S \text{ (totale calculable)} : \phi_k(x,y) = \phi_{S(y)}(x)$$

Démonstration Par le théorème 68 (S-m-n),

$$\exists\, S \text{ (totale calculable)}\ \forall\, k : \phi_k(x,y) = \phi_{S(k,y)}(x)$$

$$\forall\, k\ \exists\, S \text{ (totale calculable)} : \phi_k(x,y) = \phi_{S(k,y)}(x)$$

$$\Rightarrow \forall\, k\ \exists\, S : \phi_k(x,y) = \phi_{S(k,y)}(x)$$

$$\begin{aligned}
\Rightarrow S(k,y) &= \phi_S(k,y) \text{ (S est totale calculable)} && (3.1)\\
&= \phi_{S'(S,k)}(y) \text{ (S-m-n)} && (3.2)\\
&= \phi_{k'}(y) \text{ (renomage de } S'(S,k) \text{ en } k') && (3.3)\\
&= S''(y) && (3.4)
\end{aligned}$$

$$\Rightarrow \forall\, k\ \exists\, S'' \text{ (totale calculable)} : \phi_k(x,y) = \phi_{S''(y)}(x)$$

Remarque 70. *On peut voir le théorème 68, S-m-n, comme :*
Étant donné $m, n \geq 0$
il existe un transformateur de programme, S_n^m, qui recevant comme données : un programme P_k à $n+m$ arguments et m valeurs $v_1, ..., v_m$ fournit comme résultat : un programme P à n arguments tel que $P(x_1, ..., x_n)$ calcule la même fonction que $P_k(x_1, ..., x_n, , v_1, ..., v_m)$ (Ce programme effectue une projection)

Remarque 71. *On peut donc voir la transformation de programmes S_n^m comme un programme qui particularise un autre programme à $m+n$ argument en rendant constant les m derniers paramètres.*

3.11 Théorème du point fixe

Théorème 72 (Point fixe). *Soient $n \geq 0$ et f : fonction totale calculable, il existe k tel que $\phi_k^{(n)} = \phi_{f(k)}^{(n)}$*

Remarque 73. *Le théorème 72 n'est pas très intuitif. Mais on peut le voir comme : quel que soit un transformateur de programme T qui calcule f (n'importe quelle fonction totale calculable peut être vu comme un transformateur de programme), il existe deux programmes P_k et P_j tels que*

- *P_j est la transformation de P_k via T,*
- *P_k et P_j calcule la **même** fonction.*

Démonstration. Pour commencer, on va poser 3 "lapins" :

1. $h(u,v) = \begin{cases} \phi_{\phi_u(u)}(v) & \text{si } \phi_u(u) \neq \perp \\ \perp & \text{sinon} \end{cases}$
   h est calculable

   Remarque 74. *on peut créer un programme qui calcule h*
   Input : u, v
   $a = P_z(u,u)$
   $P_z(a,v)$
   où $\phi_z = \theta$ est la fonction universelle.

2. $h(u,v) = \phi_{S(u)}(v)$
   S est totale calculable, on a donc la propriété S (propriété S-m-n affaiblie).

3. $g(u) = f(S(u))$
   g est totale calculable car S et f le sont (f est la fonction donnée dans le théorème).

$$\exists k' \cdot \phi_{k'}(u) = g(u) = f(S(u))$$

On a que k' est une constante et par le lapin 2 :

$$h(k', v) = \phi_{S(k')}(v)$$

Par le lapin 1 et, car $g = \phi_{k'}$ est une fonction totale calculable, on a :

$$h(k', v) = \phi_{\phi_{k'}(k')}(v)$$

Par le lapin 3 on a que $\phi_{k'}(u) = g(u) = f(S(u))$ donc :

$$h(k', v) = \phi_{f(S(k'))}(v)$$

C'est à dire :

$$\phi_{S(k')}(v) = \phi_{f(S(k'))}(v)$$

Si on pose que $S(k') = k$ on a bien

$$\phi_k(v) = \phi_{f(k)}(v)$$

Ce qui conclut notre démonstration.

Remarque 75. *Le programme créé par $S(k')$ est le suivant*
Input : v
$a = P_z(k', k')$
$P_z(a, v)$

On remarque que comme S et f sont totales, g l'est aussi donc $P_z(k', k')$ se terminera toujours.

Que se passe t'il quand on exécute $S(k')$? Comme on l'a vu, la première ligne se termine et on obtient $a = f(S(k'))$, ensuite on exécute le programme $P_z(a, v) = P_a(f(S(k')), v)$ et output ce qu'il output. Exécuter $S(k')$ revient donc à exécuter $f(S(k'))$! On a donc forcément

$$\phi_{S(k')} = \phi_{f(S(k'))}.$$

Le programme $S(k')$ est donc un point fixe de f.

□

Remarque 76 (Démonstration du théorème de Rice grâce au point fixe)**.** *À l'aide du point fixe, on peut démontrer une version plus forte du théorème de Rice :*

Si A est récursif et $A \neq \emptyset \neq \bar{A}$, alors il existe $n \in A$ et $m \in \bar{A}$,

— *soit $\exists k \in A$ tel que $\phi_m = \phi_k$*
— *soit $\exists k \in \bar{A}$ tel que $\phi_n = \phi_k$.*

Démonstration. Soit un ensemble A récursif tel que $A \neq \emptyset$ et $\bar{A} \neq \emptyset$. Soit $n \in A$ et $m \in \bar{A}$. Soit la fonction

$$f(x) = \begin{cases} m & \text{si } x \in A, \\ n & \text{si } x \in \bar{A}. \end{cases}$$

On remarque que comme A est récursif, f est total calculable. Dès lors, par le théorème du point fixe, il existe k tel que

$$\phi_{f(k)} = \phi_k$$

— Si $k \in A$, ça donne

$$\phi_m = \phi_k$$

or $m \in \bar{A}$.

— Si $k \in \bar{A}$, ça donne

$$\phi_n = \phi_k$$

or $n \in A$.

□

Remarque 77 (Démonstration de K grâce au point fixe)**.** *On peut démontrer que K n'est pas récursif.*

Démonstration. Posons 3 fonctions :

— $\phi_1(x) = \perp \quad \forall\, x$
— $\phi_0(x) = x \quad \forall\, x$
— $f(x) = \begin{cases} 1 & \text{si } x \in \text{K} \\ 0 & \text{si } x \notin \text{K} \end{cases}$

On remarque que par construction pour tout k, $\phi_{f(k)} \neq \phi_k$.

— Si $k \in K$, $\phi_k(k) \neq \perp$, $f(k) = 1$ et $\phi_{f(k)}(k) = \phi_1(k) = \perp \neq \phi_k(k)$ donc $\phi_{f(k)} \neq \phi_k$.
— Si $k \notin K$, $\phi_k(k) = \perp$, $f(k) = 0$ et $\phi_{f(k)}(k) = \phi_0(k) \neq \perp = \phi_k(k)$ donc $\phi_{f(k)} \neq \phi_k$.

Si K est récursif, alors f est total calculable. Par le théorème du point fixe, il existe k tel que $\phi_k = \phi_{f(k)}$. Ce qui est contradictoire.

K n'est donc pas récursif. □

3.12 Autres problèmes non calculables

Définition 78 (Problème de correspondance de Post)**.** *Soit deux listes U et V de mots non vides sur un alphabet Σ :*

— $U = u_1, u_2, ..., u_k$
— $V = v_1, v_2, ..., v_k$

*Le problème consiste à décider s'il existe une suite d'entiers $i_1, i_2, .., i_n$ telle que les mots $u_{i1}, u_{i_2}, ..., u_{in}$ et $v_{i1}, v_{i_2}, ..., v_{in}$ soient **identiques**.*
Pour démontrer l'indécidabilité de ce problème, Post à introduit un nouveau modèle, la machine de Post qui ressemble à une machine de Turing.

Définition 79 (Problème des équations diophantiennes)**.** *Décider si une équation polynomiale de degré supérieur ou égal à 4 possède une solution entière.*

C'est le 10e problème de Hilbert, résolu en 1970 par Matiyasevich ; voir [**?**] pour un description de la solution et des remarques historiques. Ce résultat implique, en particulier, qu'il n'existe pas d'algorithme pour résoudre la programmation quadratique entière ; voir [**?**]. Un bon aperçu des question de non décidabilité pour les équation polynomiale est [**?**].

Il n'existe pas d'algorithme résolvant ces problèmes. Il existe beaucoup d'autres problèmes non calculables. Il y a d'autres exemples sur les grammaires dans le cours.

3.13 Nombres calculables

Définition 80 (Nombre réel)**.** *Un nombre réel est défini comme la limite d'une suite (convergente) de nombres rationnels : $\lim_{n\to+\infty} |x - s(n)| = 0$ ou s'est une fonction totale*

Définition 81 (Nombre réel calculable)**.** *Un nombre réel x est calculable s'il existe une fonction totale calculable s tel que* $\lim_{n\to+\infty} |x - s(n)| \leq 2^{-n}$

Remarque 82. *Donc un nombre est calculable s'il existe un programme qui peut l'approximer aussi près que l'on veut. Par exemple π et e sont calculable*

Propriétés 83. *L'ensemble des nombres réels calculables est énumérable, car on peut énumérer les fonctions totales calculables.*

Propriétés 84. *Il existe des nombres réels non calculables.*

Propriétés 85. *Il existe des nombres réels non calculables qui peuvent être définis de manière finie.*

3.14 Conclusion

Le théorème S-m-n permet de démontrer le théorème du point fixe. Le théorème du point fixe est un résultat central de la calculabilité. Il implique le théorème de Rice, la non-récursivité de K et la non-calculabilité de la fonction halt.

Chapitre 4

Modèles de calculabilité

4.1 Familles de modèles

Il y a deux grandes familles de modèles :

— Modèle de calcul (calcule une réponse)
— Modèle de langages (décide l'appartenance à un ensemble)

4.1.1 Modèle de calcul

L'objectif est de modéliser le concept de fonctions calculables, processus de calcul, algorithme effectif.

On peut encore classer les modèles de calcul en 2 catégories, les modèles déterministes et les modèles non déterministes.

Définition 86 (Modèles déterministes)**.** *une seule exécution possible*

Définition 87 (Modèles non déterministes)**.** *il existe plusieurs exécutions possibles*

On va voir les modèles de calcul suivant :

— Automate fini
— Automate à pile
— Machine de Turing
— Langages de programmation
— Lambda calcul
— Fonction récursive

Mais il en existe beaucoup d'autres.

4.1.2 Modèle de langage

Un langage est défini pas une grammaire formelle. L'objectif est de modéliser une classe de langage. Le langage est alors soit un ensemble récursif ou un ensemble récursivement énumérable.

4.2 Langages de programmation

C'est un modèle possible de la calculabilité. Pour définir un langage de programmation comme modèle de la calculabilité, il faut définir :

— Syntaxe du langage
— Sémantique du langage
— Convention de représentation d'une fonction par un programme

On se pose la question de savoir s'il y a des langages plus puissants que d'autres. On va montrer que tous les langages complets sont équivalents. (Et la plupart des langages sont complets.)

Mais, il existe aussi des langages qui ne sont pas complets comme le langage BLOOP (bounded loop).

Définition 88. *BLOOP : Sous ensemble de Java qui ne calcule que des fonctions totales. (pas de boucle while, boucle for mais sans modification du compteur dans le for, pas de goto en arrière, pas de fonctions récursives)*

BLOOP a donc toutes les propriétés qui découlent du chapitre précédent :

Propriétés 89. *Tous les programmes BLOOP se terminent*
⇔ BLOOP ne calcule que des fonctions totales
⇔ BLOOP ne calcule pas toutes les fonctions totales
⇔ il existe un compilateur des programmes BLOOP (Java)
⇔ l'interpreteur est une fonction totale non calculable en BLOOP (Hoare-Allison)
⇔ BLOOP n'est pas un modèle complet de la calculabilité

4.2.1 Langage de programmation non déterministe

Remarque 90. *Il est difficile d'avoir de l'intuition sur cette partie. On peut voir un programme ND comme un programme qui produit des résultats différents d'une exécution à l'autre. On peut représenter toutes les exécutions possibles du programmes sous forme de branches d'un arbre. Pour analyser la complexité, on ne considère que la profondeur de l'arbre (longueur de la branche la plus longue).*

On va un introduire un nouveau langage ND-Pascal qui est le langage Java auquel on ajoute le non-déterminisme sous la forme d'une fonction prédéfinie $choose(n)$. Celle-ci retourne un entier compris entre 0 et n et elle est non déterministe.

On peut voir un programme ND de 2 manières différentes :

1. Il calcule une relation plutôt qu'une fonction
2. C'est un moyen de décider si un élément appartient à un ensemble

On considère l'approche 2 en calculabilité.

Définition 91 (ND-récursif)**.** *Un ensemble $A \subseteq \mathbb{N}$ est ND-récursif s'il existe un ND-programme tel que lorsqu'il reçoit comme donnée n'importe quel nombre naturel x*
si $x \in A$ alors il existe une exécution fournissant tôt ou tard comme résultat 1
si $x \notin A$ alors toutes les exécutions fournissent tôt ou tard comme résultat 0

Définition 92 (ND-récursivement énumérable)**.** *Un ensemble $A \subseteq \mathbb{N}$ est ND-récursivement énumérable s'il existe un ND-programme tel que lorsqu'il reçoit comme donnée n'importe quel nombre naturel x*
si $x \in A$ alors il existe une exécution fournissant tôt ou tard comme résultat 1
si $x \notin A$ alors les exécutions possibles ne se terminent pas ou retournent un résultat $\neq 1$

Propriétés 93. *On peut simuler les exécutions d'un ND-programme à l'aide d'un programme déterministe. (BFS dans l'arbre d'exécution)*

Propriétés 94. *Un ensemble est ND-récursif ssi il est récursif*

Propriétés 95. *Un ensemble est ND-récursivement énumérable ssi il est récursivement énumérable*

4.3 Automates finis FA

Objectif : Décider si un mot donné appartient ou non à un langage.

Utilisation : Utilisé dans les interfaces pour les humains (par exemple, les distributeurs).

4.3.1 Modèles des automates finis

Un automate fini est composé de :

— Σ : ensemble fini de symboles
— S : ensemble fini d'états
— $s_0 \in S$: état initial
— $A \subseteq S$: ensemble des états acceptants
— $\delta : S \times \Sigma \to S$: fonction de transition

Remarque 96. *On peut aussi représenter un automate fini à l'aide d'un diagramme d'état.*

Fonctionnement

— départ avec un état initial
— parcours des symboles du mot d'entrée, un à un
— à chaque symbole lu, l'état change (fonction de transition δ) en fonction de l'état courant et du symbole lu
— état final est l'état après avoir parcouru tous les symboles en entrée
— l'état final peut-être acceptant ou non

Remarque 97. *Il n'y a donc pas de mémoire. De plus, un automate peut-être simulé par un programme Java.*

Propriétés 98. *Un automate fini définit un ensemble récursif de mots* $= \{m \mid m \text{ est accepté par } FA\}$

Propriétés 99. *Certains ensembles récursifs ne peuvent pas être reconnus par un automate fini. Par exemple* $L = \{a^n b^n \mid n \geq 0\}$ *(il me semble que c'est par ce que ça nécessiterait un nombre infini d'états)*

Propriétés 100. *L'interpréteur des automates finis est calculable, mais ne peut pas être représenté par un automate fini, car ce n'est pas un* ***modèle complet*** *de la calculabilité (Hoare-Allison)*

Définition 101 (Langage régulier)**.** *est un langage défini par une expression régulière.*

Définition 102 (Expression régulière)**.** *Dans le cours, la syntaxe d'une expression régulière est la suivante :*

+ *ou*
. *concaténation*
***** *fermeture de Kleene*[1]
( ) *répétition*

1. définit un groupe qui existe zéro, une ou plusieurs fois

4.3.2 Extension des automates finis

NDFA On étend le modèle en permettant d'avoir plusieurs transitions possibles pour une paire <état,symbole>. Ce qui implique que plusieurs exécutions sont possibles. On a donc plus une fonction de transition mais on a maintenant une relation de transition.

De même que pour un ND programme, un mot est accepté par un NDFA s'il existe au moins une exécution ou l'état final est acceptant. Dans l'autre sens, un mot n'est pas accepté si aucune exécution ne se termine avec l'état final acceptant.

Propriétés 103. *Si un ensemble récursif est défini par un NDFA, alors cet ensemble est défini par un FA.*

Propriétés 104. *Un NDFA définit un ensemble récursif de mots.*

Ajout de transitions vides ϵ On peut encore étendre le modèle NDFA en rajoutant une possibilité de transition sans lire de symbole (transition spontanée). Ça a la même puissance et les mêmes propriétés qu'un NDFA.

4.4 Automate à pile PDA

C'est une extension du modèle des automates finis. On ajoute une mémoire avec la pile de symboles. Les différences principales sont :

— la transition entre états dépend du symbole lu et du symbole au sommet de la pile

— chaque transition peut enlever le sommet de la pile et empiler de nouveaux éléments ou ne pas changer la pile.

Objectif : Décider si le mot donné appartient ou non à un langage.

Utilisation : Utilisé dans les compilateurs.

Composition : On rajoute Γ et on change la fonction de transition en une nouvelle relation de transition.

— Σ : ensemble fini de symboles d'entrée

— Γ : ensemble fini de symboles de pile

— S : ensemble fini d'états

— $s_0 \in S$: état initial

— $A \subseteq S$: ensemble des états acceptants

— $\Delta \subset S \times \Sigma \times \Gamma \times S \times \Gamma^*$: relation de transition (finie)

Propriétés 105. *Tout comme un NDFA, un PDA définit un ensemble récursif de mots (langage récursif).*

Convention :

— Z est le symbole initial de la pile (pile vide)

— ϵ signifie qu'aucun symbole ne doit être lu pour cette transition (symbole "vide")

— A, B / C : A est le symbole lu, B est le symbole au sommet de la pile et C est ce qui va remplacer le sommet de la pile (peut-être un xB pour rajouter x sur la pile, ϵ pour retirer B du sommet de la pile, ou juste B pour ne pas changer le sommet)

Propriétés 106. *Certains ensembles récursifs ne peuvent pas être reconnus par un automate à pile. Ex : L* = $\{a^n b^n a^n | n \geq 1\}$

Propriétés 107. *Les automates à pile sont plus puissants que les automates finis (ils peuvent reconnaitre plus d'ensembles)*

Propriétés 108. *Ce n'est pas un modèle complet de la calculabilité donc par Hoare- Allison, l'interpréteur n'est pas calculable dans le modèle.*

4.5 Grammaires et modèles de calcul

Objectif : Définition d'un langage (ensemble de mots) et à partir de la grammaire on peut générer/dériver les mots du langage.

Utilisation : Utilisé pour la définition de langages de programmation, pour l'analyse du langage naturel...

Composition du modèle :

— Σ : alphabet

— les éléments de Σ sont des symboles terminaux

— autres symboles utilisés durant la dérivation : symboles non terminaux (A,B, ..., <dig>,..)

— S : point de départ de la dérivation (symbole non terminal)

Définition 109 (Règle de production)**.** *On appelle un ensemble de règles de dérivation des règles de production.*

Exemple 110.
— $\Sigma = 0, 1, 2$

— $S \rightarrow < Dig >$

— $< Dig > \rightarrow D$

— $D \rightarrow 0|1|2|\epsilon$ *(ϵ signifie rien)*

TODO Exemple littéraux real en Java

Définition 111 (Dériver)**.** *Appliquer des règles de la grammaire pour vérifier si une chaîne de symbole appartient au langage (on part d'une chaîne de symboles et on vérifie les règles sur celle-ci).*

Définition 112 (Inférer)**.** *Dérivation dans "le sens contraire", c'est-à-dire, on part des règles de grammaire et on génère une chaîne de symboles.*

Définition 113 (Arbre syntaxique)**.** *Un arbre syntaxique permet de représenter la dérivation, chaque noeud correspond à un symbole terminal ou non. Les arêtes correspondent à l'application d'une règle. Il y a plusieurs nœuds enfants si la règle "génère" plusieurs symboles.*

Propriétés 114. *On peut dériver de plusieurs façons équivalentes, leftmost (on dérive toujours le plus à gauche d'abord), rightmost (contraire de leftmost) ou aucun des deux.*

4.5.1 Hiérarchie de Chomsky

Chomsky a défini 4 types de grammaires formelles. On peut les classer selon leur "puissance".

Définition 115 (Puissance d'une grammaire)**.** *Une grammaire A est plus puissante qu'une B si on peut définir plus de langages avec A qu'avec B.*

On peut aussi faire correspondre chaque type de grammaire avec un type de calcul permettant de reconnaitre un langage de cette grammaire.

Type	Type de grammaire	Modèle de calcul
3	régulière	Automate fini
2	hors contexte	Automate à pile
1	sensible au contexte	Machine de Turing à ruban fini
0	récursivement énumérable	Machine de Turing

Chaque type de grammaire est défini par une règle de production A $\rightarrow$ B. Il y a des conditions différentes sur A et B selon le type de grammaire.

4.5.2 Grammaires régulières

Règle de production :

— $A \rightarrow \omega B$
— $A \rightarrow \omega$

Conditions :

— $\omega \in \Sigma^*$, c'est-à-dire ω est une chaîne de symboles terminaux.
— A et B sont des symboles non terminaux.

Exemple 116. $S \rightarrow abS$
$S \rightarrow \epsilon$
Cette grammaire définit différents langages, par exemple $L1 = \{(ab)^n \mid n \geq 0\}$*. Ce langage peut-être aussi défini par une expression régulière :* $L1 = (ab)^*$*.*

4.5.3 Grammaires hors contexte

Cette grammaire est importante, car il suffit de lui rajouter la portée des variables pour définir la syntaxe d'un langage de programmation.

Règle de production :

— $A \rightarrow \beta$

Conditions :

— β est une chaîne de symboles composée de symboles terminaux ou non
— A est un symbole non terminal

Exemple 117. $S \rightarrow aSb$
$S \rightarrow \epsilon$
Un langage défini par cette grammaire est par exemple $L1 = \{a^n b^n | n \geq 0\}$

4.5.4 Grammaires sensibles au contexte

Règle de production :

— $\alpha \to \beta$

Conditions :

— α et β sont des chaînes de symboles composées de symboles terminaux ou non.
— β contient au moins autant de symboles que α.

Exemple 118. $S \to aSBA$
$S \to abA$
$AB \to BA$
$bB \to bb$
$bA \to ba$
$aA \to aa$
Un langage défini par cette grammaire est par exemple $L1 = \{a^n b^n a^n | n \geq 0\}$

4.5.5 Grammaires sans restriction

Règle de production :

— $\alpha \to \beta$

Conditions :

— α et β sont des chaînes de symboles composées de symboles terminaux ou non.

Exemple 119. *Il y a donc moyen de créer des règles qui bouclent :*
$\alpha \to \beta$
$\beta \to \alpha$

4.6 Machines de Turing

Intérêt : Le modèle des machines de Turing est le modèle le plus simple, le plus élémentaire et le plus puissant possible (c'est un modèle complet de la calculabilité). Il permet une définition précise de procédures, d'algorithmes ou encore de calculs.

Composition "abstraite" :

Ruban Suite de cases potentiellement infinie (des 2 côtés), mais à chaque moment, le ruban nécessaire est fini

Tête Une seule tête, sur une case qui peut écrire et lire la case sur laquelle elle est

Contrôle Dirige les actions/opérations

4.6.1 Contrôle

Le contrôleur est composé d'un nombre d'états fini dont un état initial et un final. Il contient un programme (des instructions).

Définition 120 (Une instruction)**.** *est sous la forme*

$$< q, c > \quad \rightarrow \quad < new_q, Mouv, new_c >$$

— q *: état courant*
— c *: symbole sous la tête de lecture*
— new_c *: symbole à écrire sous la tête de lecture*
— $Mouv$ *: G ou D, mouvement que la tête de lecture doit faire*
— new_q *: le nouvel état*

4.6.2 Modélisation

Pour définir une machine de Turing, il faut :

— Σ : ensemble fini de symboles d'entrée
— Γ : ensemble fini de symboles de ruban
— S : ensemble fini d'états
— $s_0 \in S$: état initial
— $stop \in S$: état d'arrêt
— $\delta : S \times \Gamma \rightarrow S \times \{G, D\} \times \Gamma$: fonction de transition (finie)

Il faut aussi que $\Sigma \subset \Gamma$ et que B $\in \Gamma$ mais que B $\notin \Sigma$

Définition 121. *B correspond au symbole blanc.*

4.6.3 Exécution

Au départ il y a juste les données d'entrée sur le ruban. Sur les autres cases, il y a le symbole B. La tête de lecture se trouve sur la première case des données. Tant que c'est possible, on applique des instructions. Il y a 2 cas possibles pour l'arrêt : soit l'état devient stop, soit il n'y a plus d'instruction applicable.

Le résultat est le contenu du ruban à l'état stop. Si la machine ne s'arrête pas sur l'état stop alors il n'y a pas de résultat.

Définition 122 (T-calculable)**.** *Une fonction f est T-calculable s' il existe une machine de Turing qui, recevant comme donnée n'importe quel nombre entier x fourni tôt ou tard comme résultat $f(x)$ si celui-ci existe.*

Définition 123 (T-récursif)**.** *Soit $A \subseteq \mathbb{N}$, A est T-récursif s'il existe une machine de Turing qui, recevant comme donnée n'importe quel nombre naturel x, fournit tôt ou tard comme résultat :* $\begin{cases} 1 & si\ x \in A \\ 0 & si\ x \notin A \end{cases}$

Définition 124 (T-récursivement énumérable)**.** *Soit $A \subseteq \mathbb{N}$, A est T-récursivement énumérable s' il existe une machine de Turing qui, recevant comme donnée n'importe quel nombre naturel x, fourni tôt ou tard comme résultat :* 1 *si* $x \in A$*.*
Si $x \notin A$, la machine renvoie un résultat $\neq 1$, s'arrête avec un état $\neq stop$ ou boucle.

4.6.4 Thèse de Church-Turing

1. Toute fonction T-calculable est calculable
2. Toute fonction calculable est T-calculable

3. Tout ensemble T-récursif est récursif
4. Tout ensemble récursif est T-récursif
5. Tout ensemble T-récursivement énumérable est récursivement énumérable
6. Tout ensemble récursivement énumérable est T-récursivement énumérable

Les points 1, 3 et 5 sont des théorèmes. Les autres sont des thèses.

4.6.5 Extension du modèle

On peut modifier le modèle pour changer sa puissance et son efficacité.

Définition 125 (Puissance d'une MT)**.** *La puissance d'une MT se mesure en fonction du nombre de fonctions qu'elle peut calculer.*

Définition 126 (Efficacité d'une MT)**.** *L'efficacité d'une MT se calcule en fonction du nombre d'instructions à exécuter (on ne tient pas compte de la taille d'un mot mémoire).*

Changer les conventions On peut par exemple permettre de se déplacer de plusieurs cases à la fois ou encore de permettre plusieurs états $stop$.

Influence :

— Même puissance
— Speedup linéaire (pour aller 20 cases à gauche on doit plus exécuter 20 instructions se déplacer à gauche)

Réduire les symboles Par exemple, ne plus avoir que 0 et 1 comme symboles dans Σ.

Influence :

— Même puissance
— Même efficacité, car même s' il y a un facteur logarithmique, en calculabilité on le néglige

Limiter le nombre d'états Cela implique qu'il y a seulement un nombre fini de machines de Turing différentes.

Influence :

— Moins puissant

Autres rubans Ruban unidirectionnel, c'est-à-dire limité d'un coté (à priori à gauche).

Influence :

— Même puissance

— Slowdown linéaire : il faut faire plus de déplacement, en effet, avant les cases étaient numérotés

$$-\infty, ..., -2, -1, 0, 1, 2, ..., +\infty$$

alors que maintenant ce sera

$$0, -1, 1, -2, 2, ..., -\infty, +\infty$$

Ruban multicases La tête lis plusieurs cases en parallèle, ce qui implique que la taille de l'alphabet augmente ($\Sigma \times \Sigma \times ...$).

Influence :

— Même puissance

— Même efficacité

Plusieurs rubans Chaque ruban à sa propre tête. On doit changer la relation de transition, car un état est défini par les positions de toutes les têtes. Le relation doit maintenant prendre l'état (E) et plusieurs symboles ($s_1, ..., s_n$) et retourner un état (E'), plusieurs symboles ($s'_1, ..., s'_n$) à écrire et plusieurs directions différentes, une pour chaque tête ($d_1, ..., d_n$).

$$< s_1, ..., s_n >, E \ \rightarrow \ E', < s'_1, ..., s'_n >, < d_1, ..., d_n >$$

Influence :

— Même puissance

— Speedup quadratique

4.6.6 Machine de Turing non déterministe NDT

Tout comme pour les automates non déterministes, on permet plusieurs transitions possibles pour une paire <état, symbole>. La fonction de transition devient une relation de transition, ce qui implique qu'il y a plusieurs exécutions possibles.

Remarque 127. *On utilise les NDT uniquement pour décider un ensemble.*

Remarque 128. *Cette partie est importante pour la partie concernant la complexité.*

Définition 129 (NDT-récursif)**.** *Soit $A \subseteq \mathbb{N}$, A est NDT-récursif s'il existe une ND-machine de Turing telle que lorsqu'elle reçoit comme donnée n'importe quel nombre naturel x :*
Si $x \in A$, alors il existe une exécution fournissant tôt ou tard comme résultat 1.
Si $x \notin A$, alors toutes les exécutions fournissent tôt ou tard comme résultat 0.

Définition 130 (NDT-récursivement énumérable)**.** *Soit $A \subseteq \mathbb{N}$, A est NDT-récursivement énumérable s'il existe une ND-machine de Turing telle que lorsqu'elle reçoit comme donnée n'importe quel nombre naturel x :*
Si $x \in A$, alors il existe une exécution fournissant tôt ou tard comme résultat 1.
Si $x \notin A$, toutes les exécutions possibles retournent soit un nombre $\neq 1$,
soit ne se terminent pas, ou encore s'arrêtent avec un état $\neq$ stop.

Influence :

— Même puissance, car il existe une machine de Turing qui interprète les NDT.

— Speedup exponentiel, car on "descend" directement au bon endroit dans l'arbre. Mais comme en pratique on doit simuler l'exécution non déterministe par un parcours en largeur de l'arbre d'exécution, ça ne change rien.

4.6.7 Machine de Turing avec Oracle

On ajoute 3 états spéciaux : soit $A \subseteq \mathbb{N}$

— $oracle_{ask}$: demander si l'entier représenté à droite de la tête de lecture appartient à l'ensemble A

— $oracle_{yes}$: l'entier appartient à A

— $oracle_{no}$: l'entier n'appartient pas à A

Puissance : Elle dépend de A. Si A est récursif, ça n'apporte rien et on garde la même puissance, car on peut remplacer l'oracle par un programme qui décide A.

Par contre, si A n'est pas récursif, alors c'est un modèle plus puissant (on pourrait déterminer halt). Mais il n'est pas possible d'exécuter un tel programme.

Remarque 131. *Utilité : permet d'établir une hiérarchie parmi les problèmes indécidables. Quels problèmes seraient encore indécidables si K était récursif ?*

4.6.8 Machine de Turing Universelle

Objectif : Construire une machine de Turing qui soit un interpréteur de machines de Turing

Remarque 132. *On définit un encodage de 0, 1 qui permet de représenter une MT*

Une telle machine est possible à construire. Il y a plusieurs façons différentes de faire. Une façon de faire est d'utiliser 3 rubans :

— codage de la MT à interpréter

— donnée

— résultat intermédiaire de l'interpréteur

4.7 Fonctions récursives

Ce modèle de calcul se base sur la définition mathématique de fonction. On va s'intéresser aux fonctions de $\mathbb{N}^k \rightarrow \mathbb{N}$.

Il y a 2 grandes classes de fonctions récursives :

— Fonctions primitives récursives, on se limite aux fonctions totales (équivalent au langage BLOOP)

— Fonction récursives, c'est un modèle complet, on peut calculer toutes les fonctions calculables

Fonctions de bases Ce sont des fonctions qui vont être utilisées pour construire nos fonctions.

Fonctions constantes
$$\begin{array}{|l} a : \mathbb{N}^0 \to \mathbb{N} \\ a() = a \end{array}$$

Fonctions successeur
$$\begin{array}{|l} s : \mathbb{N} \to \mathbb{N} \\ s(n) = n + 1 \end{array}$$

Fonctions de projection
$$\begin{array}{|l} p_i^k : \mathbb{N}^k \to \mathbb{N} \\ p_i^k(x_1, .., x_i, ...x_k) = x_i \end{array}$$

Il existe aussi 2 "règles" importantes :

Composition
$$\begin{array}{|l} h_1, h_2, ..., h_m : \mathbb{N}^k \to \mathbb{N} \\ g : \mathbb{N}^m \to \mathbb{N} \\ \overline{x} = x_1, ...x_k \\ f(\overline{x}) = g(h_1(\overline{x}), h_i(\overline{x}), ..., h_m(\overline{x})) \end{array}$$

Récursion primitive
$$\begin{array}{|l} h : \mathbb{N}^{k+2} \to \mathbb{N} \\ g : \mathbb{N}^k \to \mathbb{N} \\ \overline{x} = x_1, ..., x_k \\ f(\overline{x}, 0) = g(\overline{x}) \quad \text{(Cas de base)} \\ f(\overline{x}, n+1) = h(\overline{x}, n, f(\overline{x}, n)) \quad \text{(Cas récursif)} \end{array}$$

Remarque 133. *Lors de l'utilisation de la récursion primitive, il faut faire attention. Le cas de base ne peut pas faire appel à f et on passe toujours de $n+1$ à n, car il ne peut pas y avoir de récursion infinie.*

4.7.1 Fonctions primitives récursives

Ce modèle ne permet d'utiliser que les fonctions de base et les fonctions obtenues suite à l'application de composition ou de récursion primitive.

Propriétés 134. *Les fonctions primitives récursives ne sont pas un modèle de complet de la calculabilité. En effet, il ne peut pas y avoir de récursion infinie. Donc on ne peut calculer avec ce modèle que des fonctions totales calculables. De plus, on sait par le théorème de Hoare-Allison que, comme ce n'est pas un modèle complet, son interpréteur n'est pas calculable dans le modèle.*

Exemple 135. *La fonction d'Ackermann est une fonction calculable **non** primitive récursive :*

$$ack(0, m) = m + 1 \tag{4.1}$$
$$ack(n + 1, 0) = ack(n + 1) \tag{4.2}$$
$$ack(n + 1, m + 1) = ack(n, ack(n + 1, m)) \tag{4.3}$$

Cette fonction à une croissance plus rapide que n'importe quelle fonction primitive récursive.

4.7.2 Fonctions récursives

On va étendre les fonctions primitives récursives en ajoutant une règle :

Minimisation
$$\begin{array}{|l} h : \mathbb{N}^{k+1} \to \mathbb{N} \\ f : \mathbb{N}^k \to \mathbb{N} \\ \overline{x} = x_1, ..., x_k \\ f(\overline{x}) = \mu_n(h(\overline{x}, n) = 0) \\ \mu_n \text{ est le plus petit } n \text{ tel que } h(\overline{x}, n) = 0 \end{array}$$

Propriétés 136. *Les fonctions récursives sont un modèle complet de la calculabilité. Toute fonction calculable est une fonction récursive et vice versa.*

4.8 Lambda calcul

Remarque 137. *C'est encore un modèle peu intuitif. Je pense que c'est important de refaire l'exercice sur le vrai ou faux en lambda calcul ou encore la représentation des entiers dans le cours. Mais c'est un modèle complet et qui contient la base de la programmation fonctionnelle.*

Définition 138 (Symboles de base)**.** *Soit une variable : $a, b, c, ...y, z, ...$ ou un symbole spécial : $\lambda, (,)$*

Définition 139 (Expression lambda)**.** *est l'une des 3 choses suivantes :*

- *une variable*
- *λxB si B est une expression lambda et que x est une variable.*
  *λx correspond à la définition d'une variable **liée***
  λxB correspond à la définition d'une fonction à un paramètre, x.
- *(FA) si F et A sont des expressions lambda. On dit que F est l'opérateur et A est l'opérande. Ça représente l'application de F à A.*

Définition 140 (Variable liée)**.** *est une variable qui suit un λ ou qui apparait dans M et qu'on a λxM.*

Définition 141 (Variable libre)**.** *est une variable qui n'est pas liée.*

4.8.1 Réduction

Objectif : appliquer les fonctions (opérateur) à un opérande, jusqu'à ce qu'il n'y ait plus de fonction à appliquer. On obtient alors une forme réduite.

Définition 142 (Application de fonction)**.** *Si on a une expression lambda (FA) où F est une fonction λxB, on remplace toutes les occurrences liées de x dans B par A.*

Remarque 143. *Il faut faire attention, car lorsqu'on réduit une expression on ne peut pas introduire de conflit de nom, donc il faut renommer les variables.*

Remarque 144. *Il est possible d'avoir des réductions infinies, par exemple :*
$(\lambda x\ (xx)\ \lambda x\ (xx)) \rightarrow (\lambda x\ (xx)\ \lambda x\ (xx))$

Remarque 145. *Il est important de voir qu'il y a plusieurs façons de réduire, ça dépend de l'ordre dans lequel on applique les réductions.*

Propriétés 146. *Une expression lambda est non définie si, peu importe le choix de réduction, on n'arrive pas à une forme réduite.*

Théorème 147 (Church-Rosser)**.** *Si 2 séquences de réductions d'une expression lambda conduisent à une forme réduite, alors les expressions obtenues sont équivalentes.*

Propriétés 148. *Si une forme réduite existe, le choix de réduire l'expression la plus à gauche amène toujours à une forme réduite. (Donc, privilégier la réduction la plus à gauche.)*

Remarque 149. *Il existe 2 types de réduction la plus à gauche : la moins imbriquée (semblable au passage par nom en programmation) et la plus imbriquée (semblable au passage par valeur).*

Chapitre 5

Analyse de la thèse de Church-Turing

Dans ce chapitre on va principalement voir ce qu'est un bon formalisme.

5.1 Fondement de la thèse

La forme originale ne contient que 2 parties :

1. Toute fonction calculable par une MT est effectivement calculable
2. Toute fonction effectivement calculable est effectivement calculable par une MT

La partie 1 est démontrée et la 2 est supposée vraie. On la suppose vraie, car on a des évidences heuristiques (il y a eu beaucoup d'essais pour trouver une fonction qui ne respectait pas la thèse) et qu'on a montré l'équivalence entre tous les formalismes créés à ce jour (on a montré que toute machine constructible par la mécanique de Newton ne calcule que des fonctions calculables).

5.2 Formalismes de la calculabilité

On va se poser la question de ce qui fait un bon formalisme de la calculabilité. Il faut un formalisme qui vérifie les fondements de la thèse... Plus précisément, on va étudier des caractéristiques (des propriétés) nécessaires et suffisantes pour avoir un bon modèle de la calculabilité.

Dans ce chapitre, on va considérer un formalisme de la calculabilité D.

Caractéristiques

SD Soundness [1] des descriptions

CD Complétude des descriptions

SA Soundness algorithmique

CA Complétude algorithmique

U Description universelle

S Propriété S-m-n affaiblie

Remarque 150. ***D*** *correspond à description, c'est-à-dire une description de fonction.*
A *correspond à algorithme, c'est-à-dire à une description exécutable.*

1. Cohérence

Remarque 151. ***Soundness*** *signifie que, si on a une description dans notre modèle D alors celle-ci est cohérente, correspond bien à une fonction calculable.*
Complétude *signifie que notre modèle est complet, qu'il n'existe pas de fonction calculable qui ne le soit pas dans notre modèle.*

Caractérisiques plus en détails

SD Toute fonction D-calculable est calculable (première partie de la thèse de Turing)

CD Toute fonction calculable est D-calculable (deuxième partie de la thèse de Turing)

SA L'interpréteur de D est calculable (on peut exécuter une description de programme de D)

CA Il existe un compilateur qui étant donné un programme p dans un formalisme respectant SA produit une description de programme $d \in D$ tel que p et d calculent la même fonction.

U L'interpréteur de D est D-calculable (sinon on sait par Hoare Allison que ce n'est pas un formalisme complet).

S Il existe un transformateur de programme calculable, qui recevant comme entrée un programme $d \in$ D à 2 arguments et une valeur x fournit comme résultat un programme d' tel que $d'(y)$ calcule la même fonction que $d(x, y)$.

Un bon formalisme de la calculabilité possède toutes ces propriétés. En pratique, il suffit d'en montrer certaines, car certaines propriétés en entraînent d'autres.

Propriétés 152. *SA ⇒ SD, car si on peut trouver une description exécutable, celle-ci existe.*

Propriétés 153. *CA ⇒ CD, car si on a un compilateur qui compile un programme de P en une description dans D, alors un programme de P est calculable dans D.*

Propriétés 154. *SD et U ⇒ SA car si on sait qu'il existe une description D calculant une fonction et qu'on a un interpréteur de D D-calculable, alors la description est exécutable.*

Propriétés 155. *CD et S ⇒ CA, considérons n'importe quel autre formalisme Q qui a la propriété SA (ce qui signifie que sont interpréteur, $interp_Q(n, x)$ est calculable). Donc par la propriété CD, il existe un programme $p \in P$ tel que $\phi_p = \phi_{interp_Q}(n, x)$. Ensuite par la propriété S, il existe T, un transformateur de programme tel que : $\phi_{interp_Q}(n, x) = \phi_{T(n)}(x)$. On peut donc voir T comme un programme qui compile/transforme une description $q \in Q$ en une description $p \in P$. Ce qui implique qu'on a bien la propriété CA.*

Un bon formalisme doit donc posséder soit SA et CA ou soit SD, CD, U et S ou soit SA, CD et S ou encore CA, SD et U, car

— SA et CA $\iff$ SD, CD, U et S

— SA, CD et S $\iff$ SD, CA et U

5.3 Techniques de preuve

Pour prouver qu'un problème est non calculable, on peut utiliser :

— Le théorème de Rice

— La démonstration directe de la non-calculabilité par diagonalisation ou par preuve par l'absurde. **A priori le plus dur !** C'est plus pratique de réutiliser les problèmes pour lesquels on a déjà montré la non-calculabilité.

— La méthode de réduction

5.4 Aspects non couverts par la calculabilité (Section pas très importante pour l'examen)

La calculabilité se limite au calcul de fonctions. Or certains problèmes de la vie de tous les jours ne correspondent pas à une fonction.

Exemple 156. *Système d'exploitation*

Exemple 157. *Système de réservation aérienne*

Exemple 158. *Certains problèmes utilisent des caractéristiques de l'environnement comme des données en provenance de sonde.*

5.5 Au-delà de la calculabilité

Est-il possible d'imaginer un modèle plus puissant que les modèles qu'on a aujourd'hui ? Est-ce que les humains sont "plus puissants" que les machines dans le sens où ils pourraient calculer des fonctions non calculables ?

Ce sont des questions qui font débat. Certain ne veulent même pas poser la deuxième question c'est-à-dire est-ce que H-calculable = T-calculable.

Définition 159 (H-calculable)**.** *Une fonction est H-calculable si un être humain est capable de calculer cette fonction.*

À la question "les machines pensent-elles ?", Turing a proposé un raisonnement :

— **Hypothèse :** les machines pensent.
— Si on ne peut réfuter l'hypothèse alors celle-ci est vraie
— Envisager toutes les objections (théologique, émotions, mathématiques...) et toutes les réfuter.

Chapitre 6

Complexité

Lors de l'étude de la complexité, on ne va considérer que la borne supérieure (notation big O). De plus, on ne va considérer que les fonctions totales et donc la décision d'ensembles récursifs. En effet, si la fonction n'est pas totale, l'algorithme peut boucler. Donc on ne sait pas étudier l'efficacité.

Définition 160 (Complexité d'un problème)**.** *Complexité de l'algorithme le* ***plus efficace*** *résolvant ce problème.*

Définition 161 (Problème pratiquement faisable)**.** *S'il existe un algorithme de complexité polynomiale qui résout ce problème, alors, celui-ci est pratiquement faisable.*

Définition 162 (Problème intrinsèquement complexe)**.** *S'il n'existe pas d'algorithme de complexité polynomiale qui résout ce problème, alors, celui-ci est intrinsèquement complexe.*

Remarque 163. *Quelle est la différence entre intrinsèquement complexe et pratiquement infaisable ?*

6.1 Influence du modèle de calcul

Si un algorithme est de complexité polynomiale dans un modèle complet alors il sera polynomial dans un autre modèle de calcul. Il y aura juste un facteur polynomial entre les deux, car il existe un compilateur du premier modèle vers le second qui a une complexité polynomiale.

6.2 Influence de la représentation des données

Le choix de représentation de données induit une variation polynomiale du temps d'exécution et de l'espace.

Remarque 164. *Certains problèmes sont intrinsèquement complexes juste pour certaines données (simplexe). Celles-ci sont souvent des cas particuliers et peu rencontrées en pratique.*

Chapitre 7

Classes de complexité

On va se "limiter" au problème de décision.

Remarque 165. *On ne se limite pas vraiment, car on peut facilement transformer un problème en un problème de décision.*

7.1 Réduction

Objectif : Déduire un algorithme pour un problème P' à partir d'un algorithme P permet :

- de prouver la calculabilité/non calculabilité
- d'analyser le degré de non-calculabilité
- de déduire la complexité
- d'analyser le degré de complexité

Définition 166 (Relation de réductibilité)**.** *$A \leq B$: A est réductible à B. Cette relation induit des classes d'équivalence. On peut comprendre ça comme A est plus "simple" que B.*

Il existe plusieurs méthodes de réduction qui ont des propriétés différentes. Mais dans ce cours on va en étudier que 3 :

- réduction algorithmique
- réduction fonctionnelle
- réduction polynomiale

Définition 167 (A-complet)**.** *Soit A une classe de problème, un problème E est A-complet* ***par rapport*** *à une relation de réduction $\leq$ si*

1. $E \in A$
2. $\forall B \in A \ : \ B \leq E$

Remarque 168. *Le problème E appartient à la classe de problème A et est A-difficile.*

Définition 169 (A-difficile)**.** *Soit A une classe de problème, un problème E est A-difficile* ***par rapport*** *à une relation de réduction $\leq$ si*

1. $\forall B \in A \ : \ B \leq E$

Remarque 170. *N'importe quel problème de A peut être réduit au problème E, mais E n'appartient pas nécessairement à A.*

7.1.1 Réduction algorithmique

Ce type de réduction n'apporte aucune information au niveau de la complexité, car on peut répéter autant de fois qu'on veut l'algorithme qui décide B. On peut aussi faire un calcul avec une complexité très grande en plus d'utiliser B.

Définition 171 (Réduction algorithmique)**.** *Un ensemble A est algorithmiquement réductible à un ensemble B ($A \leq_a B$) si en supposant B récursif, A est récursif.*

Remarque 172. *C'est à dire qu'en supposant qu'on connait un algorithme qui décide B, on peut construire un algorithme qui décide A.*

Propriétés 173. *Si $A \leq_a B$ et B récursif, alors A récursif (par définition)*

Propriétés 174. *Si $A \leq_a B$ et A non récursif, alors B non récursif (par définition)*

Propriétés 175. $A \leq_a \overline{A}$

Propriétés 176. $A \leq_a B \iff \overline{A} \leq_a \overline{B}$

Propriétés 177. *Si A est récursif alors peu importe le B, $A \leq B$*

Propriétés 178. *Si $A \leq_a B$ et B récursivement énumérable alors A n'est* ***pas nécessairement récursivement énumérable***

7.1.2 Réduction fonctionnelle

Ce type de réduction n'apporte aucune information au niveau de la complexité, car tout dépend de la complexité de la fonction f.

Définition 179 (Réduction fonctionnelle)**.** *Un ensemble A est fonctionnellement réductible à un ensemble B ($A \leq_f B$) s'il existe une fonction* ***totale calculable*** *f telle que*

$$a \in A \iff f(a) \in B$$

Remarque 180. *Donc pour décider si $a \in A$ il suffit de calculer $f(a)$ et décider si $f(a) \in B$. Pour trouver une réduction fonctionnelle, il faut trouver une fonction qui transforme un problème de A en un problème de B.*

Propriétés 181. *Si $A \leq_f B$ et B récursif, alors A récursif (par définition)*

Propriétés 182. *Si $A \leq_f B$ et A non récursif, alors B non récursif (par définition)*

Propriétés 183. $A \leq_f B \iff \overline{A} \leq_f \overline{B}$

Propriétés 184. *Si A est récursif alors peu importe le B, $A \leq B$*

Propriétés 185. *Si $A \leq_f B$ et B récursivement énumérable alors A est* ***nécessairement récursivement énumérable***

Propriétés 186. $A \leq_f B \Rightarrow A \leq_a B$ *(Attention ce n'est pas toujours vrai dans l'autre sens)*

7.1.3 Différence entre $\leq_a$ et $\leq_f$

La principale différence, c'est que $A \leq_a B$ est plus du point de vue de la calculabilité. On s'intéresse au fait que ça soit possible, on peut utiliser autant de fois que l'on veut le fait que B soit récursif.
Alors que $A \leq_f B$ est plus du point de vue de la complexité. On est obligé d'utiliser un certain schéma d'algorithme :

```
input a
// some work
a2 := f(a)
// some work
if a2 in B then 1
else 0
```

On est donc limité à utiliser qu'une fois le test $f(a) \in B$ **en dernier lieu**.

7.2 Modèles de calcul

D'habitude, on utilise les machines de Turing pour avoir une définition précise de la complexité. Mais c'est peu intuitif et on s'intéresse aux frontières. Or la différence de complexité entre différents modèles est un facteur polynomial (c'est une thèse) ce qui n'a pas d'influence sur les frontières.

7.3 Classes de complexité

Classes basées sur le modèle déterministe

Définition 187 (DTIME(f))**.** *Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité temporelle $\mathcal{O}(f)$*

Définition 188 (DSPACE(f))**.** *Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité spatiale $\mathcal{O}(f)$*

Classes basées sur le modèle non déterministe

Définition 189 (NTIME(f))**.** *Famille des ensembles récursifs pouvant être décidés par un programme non déterministe Java de complexité temporelle $\mathcal{O}(f)$*

Remarque 190. *On considère juste la complexité de la branche d'exécution la plus longue. Toutes les branches sont donc finies.*

Définition 191 (NSPACE(f))**.** *Famille des ensembles récursifs pouvant être décidés par un programme non déterministe Java de complexité spatiale $\mathcal{O}(f)$*

Définition 192 (Classe P)**.**

$$P = \bigcup_{i \geq 0} DTIME(n^i)$$

Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité temporelle polynomiale.

Remarque 193. *Les classes ne dépendent pas du modèle de calcul*

Définition 194 (Classe NP)**.**

$$NP = \bigcup_{i \geq 0} NTIME(n^i)$$

Famille des ensembles récursifs pouvant être décidés par un programme Java non déterministe de complexité temporelle polynomiale.

Remarque 195. *Si on savait faire du non-déterminisme, on aurait une complexité polynomiale, mais pour le moment on ne peut que le simuler donc on a une complexité exponentielle.*

7.4 Relations entre les classes de complexité

Déterministe vs non-déterministe

Propriétés 196. $A \in NTIME(f) \Rightarrow A \in DTIME(c^f)$
f est la profondeur maximale de l'arbre. Si on simule le ND, alors on doit faire un bfs dans un arbre de profondeur f. c est le facteur de branchement.

Propriétés 197. $A \in NSPACE(f) \Rightarrow A \in DSPACE(f^2)$
C'est une borne sur le nombre de nœuds d'un graphe de profondeur f. Théorème de Savitch.

Time vs Space

Propriétés 198. $A \in DTIME(f) \Rightarrow A \in DSPACE(f)$
Le programme ne peut utiliser qu'au maximum un emplacement mémoire par instruction. Donc l'espace utilisé est limité par le nombre d'instructions.

Propriétés 199. $A \in NTIME(f) \Rightarrow A \in NSPACE(f)$
C'est la même chose que pour le cas déterministe (propriété précédente).

Propriétés 200. $A \in DSPACE(f) \Rightarrow A \in DTIME(c^f)$

Démonstration. On sait que le programme se termine car l'ensemble est récursif. La mémoire a un nombre exponentiel de configuration possible (stack, heap, program counter, ...). Si la complexité temporelle que le nombre de configuration possible, par le principe de tiroirs, on passe deux fois par la même configuration. On aura bouclé une fois. Mais comme la configuration détermine entièrement l'état du programme, on refera cette boucle une infinité de fois et le programme ne se terminera pas, ce qui contredit l'hypothèse. □

Propriétés 201. $A \in NSPACE(f) \Rightarrow A \in NTIME(c^f)$
C'est la même chose que pour le cas déterministe (propriété précédente).

Hiérarchie de complexité

On peut prouver qu'il existe pire qu'une complexité exponentielle.

7.5 NP-complétude

La question fondamentale de la complexité est la suivante : **s'il existe un algorithme non déterministe polynomial, existe-t-il un algorithme déterministe polynomial résolvant ce même problème ?** C'est-à-dire, est-ce que $P = NP$ **?** On sait que $P \subseteq NP$ mais on n'a pas encore montré si oui ou non $NP \subseteq P$.

Pour démontrer ça, on essaye de montrer qu'un élément de NP, le plus difficile, est dans P. On choisit un élément qui soit NP-complet par rapport à une relation de réduction. Ainsi, si on y arrive, ça implique que tous les autres éléments de NP sont dans P aussi.

La question est maintenant de choisir la relation de réduction. Les 2 relations de réduction définies précédemment ne suffisent pas. Ces réductions ne permettent pas d'affirmer quelque chose sur la complexité. On introduit donc une nouvelle réduction, la réduction polynomiale.

Définition 202 (Réduction polynomiale)**.** *Un ensemble A est polynomialement réductible à un ensemble B, $A \leq_p B$ s'il existe une fonction* ***totale calculable*** *f de* ***complexité temporelle polynomiale*** *telle que*

$$a \in A \Leftrightarrow f(a) \in B$$

Remarque 203. *On ajoute à la réduction fonctionnelle une contrainte de complexité sur la fonction f. Cette réduction nous permet donc de tirer des conclusions sur la complexité de A connaissant la complexité de B.*

Propriétés 204.

$$A \leq_p B \text{ et } B \in P \Rightarrow A \in P$$

Ce qui est logique étant donné qu'on a la complexité de f qui est polynomiale + la complexité de la décision de B qui est aussi polynomiale.

Propriétés 205.

$$A \leq_p B \text{ et } B \in NP \Rightarrow A \in NP$$

Ce qui est logique étant donné qu'on à la complexité de f qui est polynomiale + la complexité de la décision de B qui est non déterministe polynomiale.

Définition 206 (NP-complétude)**.** *Un problème E est NP-complet (par rapport à $\leq_p$) si :*

1. $E \in NP$
2. $\forall B \in NP : B \leq_p E$

Propriétés 207.

$$E \leq_p B \text{ et } B \in NP \Rightarrow B \text{ est } NP\text{-complet}$$

Ce qui est logique puisque ça veut dire que B est NP-difficile et dans NP.

TODO schéma sur les classes de problèmes

On va maintenant essayer de trouver des problèmes NP-complets et de trouver des propriétés intéressantes sur P.

7.5.1 Problème de décision

On va définir différemment la classe NP. On va considérer des problèmes de décision. Pour un ensemble A cela consiste à dire si oui ou non une donnée x appartient à A. On peut voir cela comme un prédicat. Par exemple $SAT(x)$: la formule x est-elle satisfaisable ?

Redéfinition de P et NP en problème de décision.

Définition 208 (Classe P)**.** *La classe P est la classe des problèmes de décision pouvant être décidés par un algorithme polynomial.*

Définition 209 (Classe NP)**.** *La classe NP est la classe des problèmes de décision $A(x)$ pouvant s'exprimer sous la forme $\exists y\ B(x, y)$ tel que :*

- *$B(x, y) \in P$ (Il est donc facile de vérifier une solution)*
- *le domaine de y est fini (taille polynomiale en x) et peut être généré, de manière non déterministe, en un temps polynomial*

Remarque 210. *On peut se représenter ça comme si le non-déterminisme permettait de générer tous les y "en même temps" ou que le non-déterminisme choisissait le bon y. Il est rapide de tester une solution, mais pas d'en trouver une.*

Définition 211 (Calcul d'un problème NP)**.** *Pour décider $A(x)$*

1. *calculer y (de manière non déterministe)*
2. *déterminer $B(x, y)$*

7.6 Théorème de Cook : SAT est NP-complet

Pour pouvoir trouver des problèmes NP-complet en les réduisant par rapport à un problème NP-complet, il faut trouver un premier problème NP-complet.

On va montrer que SAT est NP-complet en 2 parties :

1. $SAT \in NP$
2. $\forall B \in NP \ : \ B \leq_p SAT$

7.6.1 Le problème SAT

Le problème $SAT(x)$ est de décider si la formule propositionnelle x est satisfaisable ou non. C'est-à-dire : est-ce que $x \in SAT$?

La longueur d'une formule x est $O(n \log n)$, où n est le nombre d'occurrences des variables. Ça se justifie par le fait qu'en utilisant un codage d'Huffman, le code pour une variable prendra $\log n$ (c'est important pour définir la complexité du problème).

7.6.2 $SAT \in NP$

Il existe un programme ND polynomiale capable de décider si $x \in SAT$. On pose que m est le nombre de variables de x et n est le nombre d'occurrences de variables ($m \leq x$). Étapes de l'algorithme avec leur complexité :

— Générer une séquence de m valeurs logiques de façon non déterministe : $\mathcal{O}(m)$
— Substituer les occurrences des variables par leur valeur : $\mathcal{O}(n \log n)$
— Évaluer l'expression : Complexité polynomiale par une technique de réduction.

7.6.3 $\forall B \in NP \ : \ B \leq_p SAT$

Comme $B \in NP$, on a une NDMT qui décide B en un temps polynomial $p(n)$. On va montrer qu'on sait transformer en un temps polynomial la NDMT par rapport à n en une formule propositionnelle et que cette formule à une longueur qui dépend de $p(n)$ ($\mathcal{O}(p(n))$ symboles). Ce qui prouve que $B \leq_p SAT$.

Idée de la transformation Il me semble qu'on ne doit pas la connaitre pour l'examen. Mais l'idée est de représenter le ruban comme un tableau de variables booléennes (chaque ligne représente le ruban à un instant), même chose pour les états, le curseur, l'alphabet,...

7.7 Quelques problèmes NP-complets

— Problème du circuit hamiltonien HC (trouver un chemin qui passe une seule fois par tous les sommets)
— Problème du voyageur de commerce TS (trouver un chemin qui relie tous les sommets et de longueur $\leq B$)
— Chemin le plus long entre 2 sommets dans un graphe
— $3SAT$ (forme conjonctive avec 3 variables par clause)

— Programmation entière (simplexe)
— ...