UCLouvain

Résumé de LINFO1123

compilation du 3 juin 2023

Thomas Debelle

Juin 2023

Table des matières

Préface

Bonjour à toi !

Cette synthèse recueille toutes les informations importantes données au cours, pendant les séances de tp et est amélioré grâce au note du Syllabus. Elle ne remplace pas le cours donc écoutez bien les conseils et potentielles astuces que les professeurs peuvent vous donner. Notre synthèse est plus une aide qui on l'espère vous sera à toutes et tous utiles.

Elle a été réalisée par toutes les personnes que tu vois mentionné. Si jamais cette synthèse a une faute, manque de précision, typo ou n'est pas à jour par rapport à la matière actuelle ou bien que tu veux simplement contribuer en y apportant ta connaissance ? Rien de plus simple ! Améliore la en te rendant ici où tu trouveras toutes les infos pour mettre ce document à jour. (*en plus tu auras ton nom en gros ici et sur la page du github*)

Nous espérons que cette synthèse te sera utile d'une quelconque manière ! Bonne lecture et bonne étude.

Chapitre 1

Concepts

Dans ce chapitre, on s'intéresse aux ensembles, cardinalité et équipotences de ces derniers

1.1 Ensemble

Un ensemble est une *collection* d'objets, *sans répétition*, ces derniers sont appelés *éléments* de l'ensemble. Donc un ensemble peut être des chiffres, des lettres, il peut être vide symbolisé par *void* On peut réaliser des opérations dessus, on peut déterminer des *sous-ensembles d'ensemble* donc des ensembles issus d'ensemble. On a également une notion s'appelant le *complément* d'un ensemble dénoté $\overline{A}$

Langage

Un *langage* n'est autre qu'un mot ou bien un ensemble de caractères d'une taille fixée. Une chaine vide est écrite via le caractère "ϵ".
On forme un langage via un *alphabet* qui n'est autre qu'un ensemble de symboles, on le dénote "Σ". Tout langage est donc une suite de symbole issue de l'*alphabet*.
Σ^* correspond à l'ensemble des langages formés via l'alphabet.

Relations

Lorsque nous avons deux ensembles appelés A et B, on peut établir une relation appelé R qui nous donne un sous-ensemble AXB On peut représenter la relation par une table.

Fonctions

Lorsque nous avons deux ensembles appelés A et B, on peut avoir ce qu'on appelle une *fonction* F. C'est une relation tel que :

$$\exists a \in A : \exists b \in B : \quad < a, b > \quad \in f \tag{1.1}$$

Il n'existe pas plus d'un b pour un a. Si pour un a il n'existe pas de b, on dit que $f(a)$ est indéfini et donc $f(a) = \perp$ ou *bottom*.

Propriétés des fonctions

— un domaine de fonction ou dom(f) $= a \in A | f(a) \neq \perp$
— une image de fonction ou image(f) $= b \in B | \exists a \in A : b = f(a)$
— f est dit fonction totale si dom(f) $= A$
— f est dit fonction partielle si dom(f) $\in A$

— f est surjectif ssi image(f) = B autrement dit, tout élément est associé à minimum 1 élément dans B.
— f est injectif ssi $\forall a, a' \in A : a \neq a' \Rightarrow f(a) \neq f(a')$ autrement dit on ne fait correspondre qu'au plus un élément de A dans B.
— f est bijectif s'il combine *surjectif* et *injectif*

Intéressons nous aux extensions qui est le fait de rajouter une fonction qui ne définit un élément de B pas encore défini.

$$\forall x \in A : g(x) \neq \perp \Rightarrow f(x) = g(x) \tag{1.2}$$

f à la même valeur que g partout où g est défini.

Définition d'une fonction

Comme dit précédemment, une fonction est défini par sa table. On va souvent utiliser une description de la table qui permet que celle-ci soit clair et bien défini. De plus, on n'a pas besoin de savoir comment calculer ceci.
On peut également définir une table via une fonction ou un algorithme.

1.2 Ensemble énumérable

On dit que 2 ensembles ont le même cardinal (A et B) ssi il existe une bijection entre ces 2 ensembles. Donc chaque élément de A correspond à un élément de B.

On dit d'un ensemble qu'il est dénombrable ssi il est fini ou il existe une bijection entre l'ensemble $\mathbb{N}$ et cet ensemble.

Exemples

— L'ensemble $\mathbb{Z}$
— L'ensemble des nombres pairs
— Des paires d'entiers
— L'ensemble des programmes Java

Propriétés

Tout sous-ensemble d'ensemble énumérable est *énumérable*. L'union et l'intersection d'ensembles énumérables est *énumérable*.

En s'intéressant à l'ensemble des programmes informatiques, on se rend compte que c'est une *ensemble énumérable infini*. De plus, les programmes informatiques ne considèrent que des choses *énumérables*.

1.3 Cantor

Le théorème de *Cantor* nous dit que l'ensemble des nombres entre 0 et 1 compris est *non énumérable*.

$$E = x \in \mathbb{R} | 0 < x \leq 1 \tag{1.3}$$

Preuve

Pour prouver cela, on va réaliser une table et on va réaliser une *diagonalisation de Cantor*.

	chiffre 1	chiffre 2	...	chiffre $k+1$	...
x_0	x_{00}	x_{01}	...	x_{0k}	...
x_1	x_{10}	x_{11}	...	x_{1k}	...
...	...	...	...	...	...
x_k	x_{k0}	x_{k1}	...	x_{kk}	...
...	...	...	...	...	...

Ensuite, on va définir notre nombre de la diagonale qui vaut $d = 0.x_{00}x_{11}...x_{kk}$. De cet valeur, on va créer une valeur d' qui a comme propriété $x_{kk} \neq x'_{kk} \forall k$.
Mais, on doit stocker notre valeur d' dans la table. On la stock à p ce qui donne $d' = 0.x'_{p0}x'_{p1}...x'_{pp}$ mais à cause de la construction de $d = 0.x_{00}x_{11}...x_{pp}$. Par construction, $x'_{pp} \neq x_{pp}$ mais cela ne peut être respecté. Donc, **il n'y a pas** de *bijection* des $\mathbb{N}$ vers cet ensemble. Donc cet ensemble est *non énumérable*.

Autre ensemble non énumérable

- L'ensemble des $\mathbb{R}$.
- L'ensemble des sous-ensemble de $\mathbb{N}$.
- L'ensemble des chaines infinies de caractères d'un alphabet fini.
- L'ensemble des *fonctions* de $\mathbb{N}$ dans $\mathbb{N}$.

Chose intéressante à noter, comme on a une infinité non énumérable de fonctions $\mathbb{N}$ dans $\mathbb{N}$ et un nombre de programme informatique *infini énumérable*. On ne peut pas résoudre tous les problèmes informatiques donc.

Chapitre 2

Programmes calculables

2.1 Les algorithmes

Un algorithme est un ensemble *d'instructions* qui a pour but de produire un résultat. Donc un algorithme n'est pas une fonction. Il *calcule* une fonction. Un algorithme n'est pas forcément un *programme*, il peut être un *organigramme*. C'est un *ensemble fini d'instructions*. C'est une sorte de *calculateur*. Ici, on va considérer nos algorithmes comme *n'ayant pas de limite* de :

- Taille de données
- Taille d'instructions
- Taille de la mémoire, mais on a une utilisation finie.

2.1.1 Calculabilité

Avant de continuer, il faut définir la *calculabilité* des algorithmes car sans *formalisme*, les algorithmes sont non rigoureux, non exploitables.

Ici, on base cette notion sur celle des *programmes informatiques*. (plus intuitif). Ainsi, on possède **2 univers** celui des *programmes informatiques* et celui des *problèmes*. Pour être plus précis, on se base sur le langage Java et on se limite au fonction $\mathbb{N} \to \mathbb{N}$. Ainsi pour les fonctions, on aura **1 entrée** et **1 sortie**. (on peut également généraliser ceci en disant que $\mathbb{N}^n \to \mathbb{N}$)

2.2 Fonction calculable

Une fonction est dite *calculable* s'il existe un *programme Java* recevant **1 donnée** étant un nombre $\in \mathbb{N}$ et la fonction va nous retourner la *valeur* de $f(x)$ *si* elle est défini.
Si le programme *ne se termine pas* donc pas défini ou erreur d'exécution on dit que $f(x) = \perp$. On définit bien la notion de calculabilité sur *l'existence d'un programme*. on a 2 types de fonctions

1. Fonction *partielle* calculable : on a *parfois* un résultat
2. Fonction *totale* calculable : on peut *toujours* calculé quelque chose.

2.2.1 Ensemble récursif

Maintenant, on va essayer de déterminer la calculabilité sur *un ensemble de fonctions*. Le principe de décision de *calculabilité* est le principe dit *récursif*.
A est récursif si il existe un programme *Java* qui recevant n'importe quelle donnée sous forme d'un $\mathbb{N}$ fourni comme résultat :

- 1 si $x \in A$
- 0 si $x \notin A$

Donc on est face à un *algorithme* qui calcule si x est dans A ou non. C'est un algorithme complet et se termine toujours. (attention de ne pas confondre *récursif* et *récursivité*)

On dit qu'un ensemble d'algorithme est récursivement énumérable s'il est *récursif* sauf qu'il retourne $\neq 1 \quad x \notin A$ ou ne se termine pas et qu'on puisse énuméré cet ensemble.

Fonctions caractéristiques

Une fonction caractéristique de $A \subseteq N$ et :

$$X_A : N \to N : X_A(x) = 1 \text{ si } x \in A$$
$$= 0 \text{ si } x \in A$$

C'est une autre manière de déterminer si un ensemble est récursif si X_A est une fonction *calculable*. On dit qu'une fonction est récursivement énumérable ssi il existe une fonction f calculable ayant pour domaine A. Ou bien, on dit que A est vide *ou* l'image de f est A ayant une fonction f *totale* calculable.
Un ensemble récursivement *énumérable* est un ensemble dont la bijection des $\mathbb{N}$ est énumérable et calculable.

Propriétés :
- A récursif $\Rightarrow$ A récursivement énumérable
- A récursif $\Rightarrow \mathbb{N}|_A$ récursivement énumérable
- A récursif $\Rightarrow \mathbb{N}|_A$ récursif
- A récursivement énumérable et $\mathbb{N}|_A$ récursivement énumérable $\Rightarrow$ A récursif
- A fini $\Rightarrow$ A récursif
- $\mathbb{N}|_A$ fini $\Rightarrow$ A récursif
- A récursif $\Rightarrow \bar{A}$ récursif

2.3 Thèse de Church-Turing

Comment démontrer qu'une fonction **n'est pas** calculable.
Les 4 grands points de la thèse :

1. Aucun modèle de la notion de fonction calculable n'est plus puissant que les Machines de Turing (ici Java)
2. Toute fonction calculable (au sens intuitif) est calculable par une machine de Turing (ici Java)
3. Toutes les définitions formelles de la calculabilité connues à ce jour sont équivalentes (Théorème)
4. Toutes les formalisations de la calculabilité établies par la suite seront équivalentes aux définitions connues

On établit que Java a accès à une infinité de mémoire (donc physiquement impossible). Ainsi, on a P qui est l'ensemble des programmes Java syntaxiquement corrects, qui reçoivent 1 données *entières* et qui retournent un résultat *entier*.
- P est un ensemble récursif (infini dénombrable)
- $P = P_0, P_1, ..., P_k, ...$ sans répétition donc chaque programme est unique.
- Pour simplifier, $f(k) = P_k$
- f est calculable.
- k et P_k représente le même objet

donc on dit que P_k donne le programme k dans l'ensemble P. on dit que φ_k est la fonction mathématique calculé par P_k. Donc on peut avoir $\varphi_m = \varphi_n$ car réalise le même travail mais sont issues de programmes *différents*. $\varphi_k : N \to N$.

2.4 Non calculabilité

Pour rappel :
— Nombre de fonctions de $\mathbb{N} \to \mathbb{N}$ est **non** dénombrable.
— Nombre de programmes Java est dénombrable.
En programmation, on s'intéresse aux fonctions définies de manière finie, donc on a une infinité dénombrable. Mais si une fonction est définie de manière finie, peut-elle être calculable ?

2.4.1 Problème de l'arrêt

Une fonction prends 2 paramètres : halt : P x N. P est le numéro du programme et N est son entrée.

$$\begin{aligned} \text{halt}(n,x) &= 1 \text{ si } \varphi_n(x) \neq \perp \\ &= 0 \text{ sinon} \\ \text{halt}(n,x) &= 1 \text{ si l'exécution du } P_n \text{ se termine} \\ &= 0 \text{ sinon} \end{aligned}$$

On a donc une table finie, mais décrite de manière finie donc bien définie. Peut-on la calculer ?

Preuve par l'absurde

	0	1	...	k	...
P_0	halt(0,0)	halt(0,1)	...	halt(0,k)	...
P_1	halt(1,0)	halt(1,1)	...	halt(1,k)	...
...	...	...	...	...	...
P_k	halt(k,0)	halt(k,1)	...	halt(k,k)	...
...	...	...	...	...	...

On va sélectionner les valeurs sur la diagonale et stocker cela comme une variable s'appelant "$diag$". On va donc modifier cette valeur et est représenté par "$diag_{mod}$" qui inverse chaque nombre. (donc $0 \to 1$ et $1 \to 0$) Donc $diag_{mod}$ est calculable sous l'hypothèse que la fonction "halt" l'est.

Donc, il existe un programme Java qui calcule cette "$diag_{mod}$" qu'on trouvera en ligne d. Mais à cause de cela, $diag_{mod}(d) \neq diag(d)$ donc ne peut exister par *définition*.

En conclusion, la fonction "halt" **n'est pas** calculable.

Conclusion

— Aucun algorithme ne permet de déterminer pour tout programme P_n et donnée x si $P_n(x)$ se termine ou non
— Seule possibilité serait d'avoir un langage de programmation dans lequel tous les programmes se terminent. La fonction halt est alors calculable pour les programmes de ce formalisme
— halt non calculable ne signifie pas que pour un programme k donné, halt(k,x) est non calculable
Pour le premier point, on ne peut séparer le soucis en 2 algorithmes car on ne peut changer de programmes selon l'input. Un algorithme donne le **bon résultat** en fonction du résultat.

On dit que halt n'est pas calulable dans le sens où il n'existe pas d'algorithmes **généraux**.

Exemple non-récursif

$$\begin{aligned} Halt &= \{(n,x)|\ \text{halt}(n,x) = 1\} \\ &= \{(n,x)|P_n(x) \text{ se termine}\} \\ K &= \{n|(n,n) \in \text{HALT}\} \\ &= \{n|halt(n,n) = 1\} \\ &= \{n|diag(n) = 1\} \\ &= \{n|P_n(n) \text{ se termine}\} \end{aligned}$$

Donc K et HALT **ne sont pas** récursifs mais sont récursivement énumérable car si un élément n'appartient pas à K ou HALT, il va boucler mais fournir la bonne solution s'il appartient à ces ensembles.
De plus K est la diagonale de *HALT*.

$\overline{HALT}$ n'est pas récursivement énumérable car on a pas de moyen de prouver qu'un élément probable n'appartient pas à cet ensemble. pareil pour $\overline{K}$

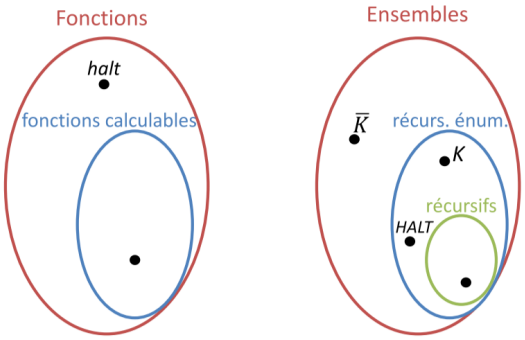


FIGURE 2.1 – Schématisation des fonctions et ensembles de fonctions

Le fait de ne pas pouvoir calculer "halt" nous pose soucis et va ouvrir tout un pan de soucis et limitations.

De plus, on peut faire face à des ensembles **co-récursivement** énumérable. En effet, il existe des ensembles co-récursivement énumérable tel que : $\overline{\mathbf{K}}$ et tous les ensembles *récursifs*.

2.5 Insuffisance des fonctions totales

Pourquoi est-ce utile d'avoir un programme qui tourne en *boucle* ? N'avons-nous pas besoin de fonctions qui donnent un résultat précis tout le temps donc *totale* ?

Imaginons que nous créons un *langage de programmation* qui a que des fonctions totales. Donc, **Halt** est calculable et on a une réponse pour toutes fonctions. Halt serait la fonction constante 1.

Notre langage Q est calculable donc ayant un interpréteur calculable.

2.5.1 Théorème de Hoare-Allison

Donc en résumé de notre langage Q :
— L'interpréteur de ce programme est calculable
— La fonction *halt* est totale et correspond à la fonction constante de 1.
— Mais l'interpréteur n'est pas calculable *dans* Q.

	0	1	...	k	...
Q_0	interpret(0,0)	interpret(0,1)	...	interpret(0,k)	...
Q_1	interpret(1,0)	interpret(1,1)	...	interpret(1,k)	...
...	...	...	...	...	...
Q_k	interpret(k,0)	interpret(k,1)	...	interpret(k,k)	...
...	...	...	...	...	...

La colonne Q correspond à l'ensemble des programmes et la ligne de nombre correspond aux entrées de chaque programme.

Tous les programmes se terminent donc *jamais* $\perp$. On sélectionne la diagonale :

$$diag(n) = interpret(n, n)$$
$$diag_{mod}(n) = interpret(n, n) + 1$$
$$Q_l = diag_{mod}$$

Et donc, on voit facilement que à la ligne l il y aura un souci avec $diag$ et notre entrée à Q_l qui n'est autre que $diag_{mod}$. en effet $diag_{mod}(l) \neq diag(l)$. Donc la fonction *interpret* n'est **pas** calculable en Q.

Le *théorème* nous dit donc que : Si un langage de programmation (non trivial) ne permet que le calcul de fonctions totales, alors :

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

Donc si on peut faire un interpréteur d'un langage dans son langage, la fonction *halt* n'est pas totale. Donc c'est soit programmable par lui-même soit fonction totale de halt.

Si on veut qu'un langage de programmation permette la programmation de toutes les fonctions totales calculables, alors ce langage doit également permettre la programmation de fonctions non totales.

De plus, si on avait une fonction qui regarde si des fonctions sont totales, cela pose problème. Cela est impossible et donc cette fonction $tot(n)$ n'est pas récursif.

2.5.2 Interpréteur

Pour qu'un formalisme soit assez puissant, il faut que ce dernier arrive à programmer son propre interpréteur.

$$\exists z \forall n, x : \varphi_z(n, x) = \varphi_n(x) \tag{2.1}$$

Avec φ_z qu'on appelle la fonction universelle. et P_z est le programme universel. Par convention, on appelle $\theta(n, x)$ la fonction universelle.

2.6 Extension des fonctions partielles

Pour l'instant, nous n'avons vu que des fonctions qui soit donnent le bon résultat soit donne $\perp$ et donc boucle. On va réaliser des *extensions*, c'est-à-dire que nous allons retourner la valeur correcte dans les cas possible et un message ou autre chose pour le reste des entrées.

Un *théorème* nous dit que, *Il existe une fonction partielle calculable g telle qu'aucune fonction totale calculable n'est une extension de g.*
Pour prouver cela, on utilise la fonction *nbstep(n,x)* qui correspond au nombre d'instruction avec l'arrêt de $P_n(x)$. ($P_n(x) = \perp$)La preuve se fait pas diagonalisation comme avant (vidéo).

2.7 Théorème de Rice

2.7.1 Réduction à Halt

Pour montrer que f(x) est non calculable on suppose que :
— f(x) est calculable.
— Sous cette hypothèse la fonction halt(n,x) est alors calculable.
— Comme halt(n,x) est enfaite non-calculable, f(x) est également non-calculable.

Raisonnement

Définissons une fonction qui dit que :

$$\begin{aligned} f(n) &= 1 \quad \text{si } \varphi_n(x) = \perp \\ &= 0 \quad \text{sinon} \end{aligned}$$

On suppose que $f(n)$ est calculable et on veut montrer que halt est calculable. Pour se faire :

1. On construit (pas exécute) un programme qui dit : $P(z) \equiv P_n(x); print(1)$
2. On obtient le numéro de programme : $d =$ numéro de programme $P(z)$
3. On regarde si le résultat de cette fonction calculable pour créer halt :

```
if F(d) = 1 then
        print(0)  //car le programme se termine
else
        print(1)  //car on est bottom et cela boucle
```

4. Donc on en conclue que cette fonction halt est non-calculable

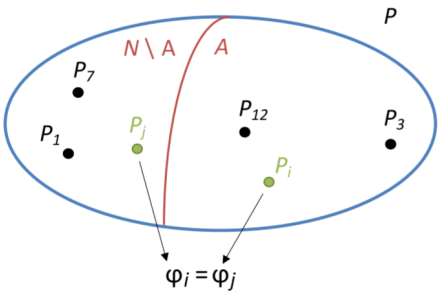
On utilise ce type de démarche pour tout autre fonction du même genre. (on doit impérativement définir le comportement de la fonction)

2.7.2 Théorème

On s'intéresse à comparer des programmes entre eux, savoir si un programme est correct.

Idée de base

Dans l'ensemble des programmes, on va séparer cet ensemble en 2 *sous-ensembles*.



On a que :

$$\begin{aligned} &\text{Soit } A \subseteq \mathbb{N} \\ &\text{Si } A \text{ récursif et } A \neq \varnothing \text{ et } A \neq \mathbb{N} \\ &\qquad \text{Alors } \exists i \in A \text{ et } j \in \mathbb{N} \setminus A : \varphi_i = \varphi_j \end{aligned}$$

Ce qui revient à dire que :

$$\text{Si } \forall i \in A \text{ et } \forall j \in \mathbb{N} \setminus A : \varphi_i \neq \varphi_j$$
$$\text{Alors } A \text{ non récursif } \textit{ou } A = \varnothing \text{ ou } A = \mathbb{N}$$

De plus, si un programme est récursif, on peut savoir de quel ensemble il est.

Compréhension

1. Si une propriété de programmes, vérifiée par certains programmes mais pas tous, est décidable, alors il existe deux programmes équivalents (calculant la même fonction) dont un vérifie la propriété et l'autre pas
2. Si une propriété de la fonction calculée par un programme est vérifiée par certains programmes, mais pas par tous,
   alors cette propriété ne peut être décidée par un algorithme
3. 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

Voyons plus en détail ce que chacun veut dire :

1. C'est une simple traduction du théorème en français.
2. Cela signifie qu'on ne peut pas vérifier si 2 programmes sont équivalents !
3. Autre façon d'énoncer le théorème de *Rice*.

2.7.3 Exemple

Commençons avec : $A_1 = \{i | \varphi_i \text{ est totale}\}$ (P_i s'arrête toujours)
Donc on sait que $A_1 \neq \varnothing$ car on a une fonction $P_k(x) \equiv print(1)$ et on sait ainsi que $k \in A_1$. Ce n'est pas $A_1 \neq \mathbb{N}$ car $P_l(x) \equiv whiletrue$ n'est pas total ! De plus, notre ensemble A a des fonctions totales et l'ensemble $\overline{A}$ des fonctions non-totales, par construction : $\forall i \in A$ et $\forall j \in \mathbb{N} : \varphi_i \neq \varphi_j$

Via le théorème de *Rice*, A_1 est un ensemble non-vide et non égale à $\mathbb{N}$. On a également prouvé que les fonctions ne sont pas égales entre les 2 *ensembles*. On en *conclut* que l'ensemble **n'est pas récursif**.

2.7.4 Analyse du théorème

Bonne nouvelle, on est pas remplaçable (ou presque).

1. Aucune question relative aux programmes, vus sous l'angle de la **fonction** qu'ils calculent, ne peut être décidée par l'application d'un algorithme
2. Les propriétés intéressantes d'un programme concernent la fonction qu'il calcule, non pas la forme (syntaxe) du programme
3. La plupart des problèmes intéressants au sujet des programmes sont non calculables

Mauvaise, on ne peut pas automatiser qu'un programme est correct.

2.7.5 Démonstration

Supposons que $A \neq \varnothing$ et $A \neq \mathbb{N}$. Et on suppose que $\forall i \in A, \forall j \in \overline{A} : \varphi_i \neq \varphi_j$ Donc A ne peut être récursif. On va **démontrer** cela.

Pour prouver cela on va utiliser une réduction à *halt*. On fait les choses suivantes :

1. On suppose que A est récursif.

2. Donc halt est calculable mais ce n'est pas le cas.
3. Donc A est n'est pas récursif.

On va donc séparer notre ensemble des Programmes et on effectue ces manipulations :

1. $\mathbb{N}$ en A et $\overline{A}$. On dit que $P_k(x) \equiv while \quad true$ et que $k \in \overline{A}$
2. On sait que $A \neq \varnothing, \exists m \in A$ (on sait que c'est différent du vide donc il existe au moins 1 programme quelconque)
3. On peut affirmer que (par hypothèse de récursif) $\varphi_k \neq \varphi_m$
4. On crée un programme pour *halt* :
   — On construit : $P(z) \equiv P_n(x); P_m(z)$
   — On assigne un numéro de programme qu'on a construit : $d = \#P(z)$. Cela dépend de ma donnée n et x.
   — on exécute un programme qui regarde si $d \in A$ il print 1 qu'il appartient sinon il imprime 0.
5. On exécute le programme : si $P_n(x)$ se **termine** alors $\varphi_d = \varphi_m$. **Sinon** il boucle et est $\varphi_d = \varphi_k = \perp$
6. Donc aucun programme dans A et $\overline{A}$ n'est équivalent. Donc tester que $\varphi_d = \varphi_m$ est la même chose que tester que $d \in A$ et inversement.
7. Donc, Halt est non calculable car il boucle, donc A est bien **non récursif**.

2.8 Théorème de la paramétrisation

De manière générale, une fonction peut être vu comme $f : \mathbb{N} \to \mathbb{N}$ mais *également* comme $f : P \to P$. Donc $f(a) = b$ et en combinant $P_a = P_B$ notre f devient un **transformateur de programmes**. Pour être plus précis c'est le P_k qui calcule f qui est le *transformateur de programmes.*

2.8.1 Théorème S-m-n via S-1-1

On dit qu'il existe une fonction calculable qui prend 2 arguments tel que :

$$S_1^1 : N^2 \to N \text{ et } \forall k$$
$$\varphi_k^{(2)}(x_1, x_2) = \varphi_{S_1^1(k,x_2)}(x_1)$$

Compréhension

On dit qu'il existe un *transformateur* (S_1^1) qui prend en arguments : - Un numéro de programme à 2 arguments - 1 valeur v_2
Cela donne comme résultat un programme $P_l(x_1)$ qui calcule la même chose que $P_k(x_1, v_2)$.

2.8.2 Via S-m-n

On dit que :

$$\forall m, n \geq 0, \exists S_n^m : N^{m+1} \to N : \forall k$$
$$\varphi_k^{n+m}(x_1, ..., x_n, x_{n+1}, ..., x_{n+m}) = \varphi^{(n)}_{S_n^m(k, x_{n+1}, ..., x_{n+m})}(x_1, ..., x_n)$$

Compréhension

Comme nous avons $m, n \geq 0$, il existe un transformateur de programmes appelés S_n^m qui reçoit : - P_k avec $m + n$ arguments, m valeurs $v_1, ..., v_m$
Cela renvoi comme résultat un programme P à n arguments. Donc $P(x_1, ..., x_n)$ calcule la *même* fonction que $P_k(x1, ..., x_n, x_{n+1}, ..., x_{n+m})$

2.9 Théorème du point fixe

C'est un résultat qui montre que quelque chose est possible.

2.9.1 Définition

Si on a une fonction f qui est **totale calculable**, je sais qu'il existe k tel que :

$$\varphi_k^{(n)} = \varphi_{f(k)}^{(n)} \tag{2.2}$$

Pour tout transformateur (notre fonction totale calculable) de programme T, il existe deux programmes P_k et P_j tels que :

1. P_j est la transformation de P_k via T
2. P_k et P_j calculent la même fonction (qui n'est pas nécessairement totale)

2.9.2 Applications

— Il existe un programme P_n tel que P_n ne s'arrête que pour la donnée n.
— Il existe un programme P_n tel que P_n donne toujours n comme résultat.
— $K = \{n | \varphi_n(n) \neq \perp\}$ n'est pas récursif.
— Transformateur de programmes qui remplaces les + en −.

Démonstration pour K

Si on a un programme $\varphi_n(x) = \perp \forall x$ (boucle tout le temps) et un autre $\varphi_m(x) = x \ \forall x$. On définit une fonction :

$$\begin{aligned} f(x) &= n \text{ si } x \in K \\ &= m \text{ si } x \notin K \end{aligned}$$

Donc f est totale et calculable si K est **récursif**. On va appliquer le point fixe : $\exists k : \varphi_k = \varphi_{f(k)}$. Mais $k \in K$?
Si c'est le cas : alors $f(k) = n$ par définition de f. Donc $\varphi_k(k) = \varphi_n(k)$ via le point fixe. Mais $\varphi_k(k) \neq \perp$ car appartient à K mais $\varphi_n(k) = \perp$. Contradiction.
Si ce n'est pas le cas : alors $f(k) = m$. Donc $\varphi_k(k) = \varphi_m(k)$. $\varphi_k(k) = \perp$ car n'appartient pas à l'ensemble et $\varphi_m(k)$ est constant k. Contradiction.

2.9.3 Théorème de Rice via théorème du point fixe

Si $A \neq \varnothing$ et $A \neq \mathbb{N}$ et $\forall i \in A$, $\forall j \in \overline{A}$ qui $\varphi_i \neq \varphi_j$ donc A est non récursif.

Démonstration

Comme on a $A \neq \varnothing : n \in A$ et $A \neq \mathbb{N} : m \in \overline{A}$. On définit :

$$\begin{aligned} f(x) &= m \text{ si } x \in A \\ &= n \text{ si } x \in \overline{A} \end{aligned}$$

Cette fonction f est totale calculable si A est récursif !
En appliquant le point fixe : $\exists k : \varphi_k = \varphi_{f(k)}$. Mais $k \in A$?
Si c'est le cas : alors $f(k) = m$ alors $\varphi_k = \varphi_m$ donc $k \in \overline{A}$ par définition. En effet $m \in \overline{A}$ et grâce à l'hypothèse du théorème de **Rice**.
Si ce n'est pas le cas : alors $f(k) = n$ alors $\varphi_k = \varphi_n$ donc $k \in A$ par définition. En effet $n \in A$ et grâce au théorème de **Rice**.
Donc A est **non récursif** !

2.9.4 Analyse

C'est un théorème important et central. Il utilise **uniquement** la propriété S des langages de programmation. Il implique :

1. Théorème de Rice.
2. La non récursivité de K.
3. Implie la non calculabilité de fonction $HALT$.

2.10 Démonstration du théorème du point fixe

On va faire la technique des 3 lapins. Si on a une fonction f qui est totale calculable, on veut montrer que $\exists k : \varphi_k = \varphi_{f(k)}$.

Lapin 1

$$h(u,x) = \begin{cases} \varphi_{\varphi_u(u)}(x) \text{ si } \varphi_u(u) \neq \perp \\ \perp \text{ sinon} \end{cases} \tag{2.3}$$

Cette fonction h est calculable. On va d'abord exécuter $\varphi_u(u)$ puis on essaye de faire l'autre φ sinon $\perp$.

Lapin 2

$$h(u,x) = \varphi_{g(u)}(x) \tag{2.4}$$

On va appliquer la propriété S. g existe et est totale calculable.

Lapin 3

$$t(x) = f(g(x)) \tag{2.5}$$

t est totale calculable car g l'est et f de mon hypothèse l'est. Ainsi, on a un **numéro de programme** k'.

$$\exists k' : t(x) = \varphi_{k'}(x) = f(g(x)) \tag{2.6}$$

Fin du tour de magie

$$\forall x : h(k',x) = \varphi_{g(k')}(x) \text{ via Lapin 2 (2.4)}$$

$$\begin{aligned} \forall x h(k',x) &= \varphi_{\varphi_{k'}(k')}(x) \text{ via Lapin 1 (2.3)} \\ &= \varphi_{f(g(k'))}(x) \text{ via Lapin 3 (2.6)} \end{aligned}$$

Si je pose que $k = g(k')$ alors $\forall x : \varphi_k(x) = \varphi_{f(k)}(x)$.

2.11 Autres problèmes non calculables

Liste équivalente de mot

Si on a 2 listes comme ci-dessous, il n'existe **aucun** algorithme qui nous permettent de trouver une suite d'indice qui produit 2 mots équivalents depuis deux listes. C'est donc un problème **non calculable**.

Σ = {a , b}
U = {u1=b , u2=babbb , u3=ba}
V = { v1=bbb , v2=ba , v3=a}
Res = {2 1 1 3} //car U : babbb, b, b, ba et V : ba, bbb,bbb, a

Équations diophantiennes

C'est une équation de la forme : $D(x_1, ..., x_n) = 0$. L'équation ne possède que des *coefficients entiers* et on cherche des solutions *entières*.
C'est bel et bien un problème **non calculable**.

$$P(a) \leftrightarrow \exists x_1, ..., x_n[D(a, x_1, ..., x_n) = 0]$$

Ci-dessus, c'est la condition pour être diophantien pour une fonction.

2.12 Codage et représentation

Jusqu'ici, on avait des fonctions $N^n \to N$ et on avait des représentations décimales des nombres. Mais on ne prend pas en compte les autres types de donnés, ... Donc on va créer des fonctions de codage et représentation.

Codage

On va définir une bijection entre notre nouveau type et les entiers. On peut identifier chaque objet par un entier *distinct*. Ce qui nous amène à une propriété très utile :

1. la fonction de codage est bijective
2. la fonction de codage est calculable
3. la fonction de codage inverse est calculable

2.12.1 Les nombres calculables

On sait que tout nombre réel est fini tout comme la limite d'une suite *convergente* vers un rationnel $\mathbb{Q}$ Donc :

$$\forall x \in \mathbb{R}, \exists s : N \to Q : \text{ (s est total)}$$
$$lim_{n\to\infty}|x - s(n)| = 0$$

Définition

On dit qu'un réel x est **calculable** si il existe une fonction totale calculable :

$$s : N \to Q$$
$$\forall n : |x - s(n)| \leq 2^{-n}$$

Donc notre fonction ne fait que s'approcher autant qu'on veut de notre réel.

Propriétés

Donc un $\mathbb{R}$ est calculable s'il existe un programme qui le calcule. Ce qui par induction rend l'ensemble des nombres calculables énumérables.
Il existe des réels donc non-calculables mais qui peuvent être définie de manière finie comme :

$$x = \sum_{0 \geq n \geq \infty} \chi_K(n) 3^{-n}$$

Chapitre 3

Modèles de calculabilité

3.1 Familles de modèles

Il existe 2 familles de modèles :

1. Basé sur le calcul
2. Basé sur le langage (chaine de caractère)

3.1.1 Basé sur le calcul

Leur objectif est de modéliser le concept de fonctions calculables, processus de calcul, algorithme effectif, ... Ex

— Automate fini
— Machine de Turing
— Langages de programmation

Il existe 2 sous-catégories :

1. Modèles déterministes : il n'y a qu'un seule exécution possible d'un programme pour un input donné.
2. Modèles non déterministes : il existe plusieurs exécutions possibles pour un input donné.

3.1.2 Langes de programmation

C'est le modèle basé sur le calcul qui est un exemple de la **calculabilité** qui est le plus *courant*. (on l'a utilisé pour démontrer toutes les théories fondamentales de la *calculabilité*)

Définir un langage

On doit définir la :

1. Syntaxe du langage
2. Sémantique du langage
3. Convention de représentation d'une fonction par un programme

Équivalence des programmes

On sait que les langages de programmation sont équivalents entre eux. Une fonction est calculable en *Rust* alors elle le sera en *C*. On dit que les langages sont Complets.
Bien évidemment, certains problèmes sont plus simples et/ou plus efficace à résoudre dans un langage de programmation spécifique.

BLOOP

BLOOP ou *Bounded loop* qui est Java sans les boucles while, on ne modifie pas la variable de *compteur* dans le corps d'un *for*. Pas de méthodes récursives ni mutuellement récursibles.
Donc BLOOP se termine toujours et ne calcule que des fonctions *totales*. Cependant, il ne calcule pas toutes les fonctions totales (voir 2.5.1). En effet, l'interpreteur n'est pas programmable en BLOOP.
Donc ce **n'est pas** un modèle complet de la calculabilité.

3.1.3 Non-déterministe

On va créer ND-Java qui est une version *non-déterministe* de Java. On lui ajoute la fonction **choose(n)** qui renvoie un entier entre **0 et n** et est donc non déterministe.
On doit considérer n+1 exécution possible à chaque lancement de la fonction.
Donc pour le problème du voyageur de commerce, on va énumérer tous les chemins possibles en swapant un à un les villes. Donc on aura un arbre de décision. Ainsi, on peut regarder la profondeur du chemin pour savoir si le nombre de possibilité est borné.
Il y aura des chemins où ce sera borné et des chemins où ça ne l'est pas.
On dit qu'un programme nous produit une relation plutôt qu'une fonction.

Récursif ND

On dit qu'un ensemble A ($A \in \mathbb{N}$) est ND-récursif si il existe un programme **ND-Java** tel que lorsqu'il reçoit comme donnée n'importe quel nombre naturel x.
— si $x \in A$, alors il existe un exécution fournissant comme résultat 1 (peu importe le temps)
— si $x \notin A$ alors **toutes** les exécutions possibles fournissent comme résultat 0.

ND Récursivement énumérable

On dit que A est ND-récursivement énumérable si il existe un programme ND-Java tel que lorsqu'il reçoit comme donnée n'importe quel nombre naturel x, il existe une exécution fournissant comme résultat 1 **ssi** $x \in A$. (Si $x \notin A$ alors la fonction retourne autre chose que 1 ou est $\perp$)

Propriétés

Un ensemble est **ND-récursif** ssi il est **récursif**. Si nous avons un arbre d'exécutions possibles, il est toujours possible de parcourir tous les **étages** de notre arbre et donc parcourir toutes les exécutions possibles comme si on était en "*déterministe*".
Si la profondeur d'une branche est de taille n, alors mon parcours a une complexité exponentielle tel que b^n.

3.1.4 Basé sur le langage

Se base sur un ensemble de mot et possède une grammaire formelle. On essaye de modéliser une classe de langages. Le langage est donc un ensemble *récursif* ou *récursivement énumérable*. Pas vu dans ce cours

3.2 Automates finis

C'est une modélisation assez simple du concept de *calcul*, il y a :
— Un nombre *fini d'états*
— Une lecture d'une donnée : un mot
— Chaque symbole de la donnée est lu *une et une seule* fois
— On transitionne d'état en état en fonction du *symbole lu*.
— L'état final est l'état après avoir *lu tous les symboles* de la donnée
— On **ne peut pas** mémoriser quelque chose.

Donc l'objectif d'un automate fini est de **décider** si un mot appartient à un langage. Réponse binaire : Oui ou Non. **Pas de boucle**.

Modèle

On a un :
- $\sum$ qui est l'ensemble (*fini*) de symboles
- S qui est l'ensemble (*fini*) d'états
- $s_0 \in S$ qui est l'état initial
- $A \subseteq S$ qui est un ensemble d'état acceptant
- $\delta S\Sigma \to S$ qui est une fonction de *transition*. (C'est une sorte de tableau)

C'est donc un modèle de calcul. On parcourt le mot symbole par symbole et on termine notre exécution à la fin de la lecture de symbole. Pour simplifier tout cela, on va rajouter un état *fail* si on n'a pas tout déterminé pour les multiples possibilités.

Un état acceptant est représenté par un **double cercle** et un état non-acceptant est représenté par un **simple cercle**. Les flèches ou les *arcs* sont les fonction de transition.

Un mot est accepté si l'état final de ce mot est dans l'état *acceptant*. (et vice-versa)
Dans un automate fini, son exécution se finit **toujours** ! Donc cela ne détermine que des ensembles récursifs (mais pas tous les ensembles récursifs (2.5.1))
Il est aussi important de remarquer que si on ne possède pas de transition à un état, on échoue et l'automate échoue !

3.2.1 Extension

On peut étendre les automates aux automates finis non déterministes. On a donc plusieurs transitions possibles pour une paire. Plusieurs exécutions pour une même donnée.
Un mot m est accepté par *NFDA* si il existe **une** exécution de NFDA, où on finit dans un état acceptant.
Pour toute exécution de NFDA, un mot **n'est pas accepté** s'il ne tombe jamais sur un état acceptant.

On peut voir comme si on rassemblait des exécutions ensemble et des ensemble d'états.

Avec transitions vides

On peut faire des transitions sans **rien** consommer via le symbole $+ - \varepsilon$. N'apporte pas de puissances en plus juste un passage sans consommer.

Interprétation

On peut voir ça comme le travail d'un compilateur qui fait une analyse lexical via un découpage de *tokens*.
On peut faire de la recherche de paterne (regex). Donc utile dans un éditeur de texte.
Un automate fini est également utile pour les *interfaces utilisateurs*. Cela se comporte toujours de la même manière peu importe ce qu'il s'est passé auparavant.

3.3 Machines de Turing

C'est une idée inventée par *Alan Turing* en 1936 qui est antérieur aux ordinateurs. C'est un modèle très *simple* mais qui est le plus *puissant* possible.

3.3.1 Fonctionnement

On dispose d'un ruban qui est infini à *droite* et à *gauche*. On a une *tête de lecture* qui peut lire et écrire un caractère à l'endroit qu'elle pointe sur le ruban.
Il y a également un mécanisme de *contrôle* qui gère les actions à l'exécution.

Contrôle

On a un ensemble fini d'états : soit en *état initial* ou un *état d'arrêt*. Le contrôle contient un programme et un *mécanisme* qui exécute les instructions.
Les formes d'instructions sont comme :

$$< q, c > \rightarrow < q_{new}, Mouv, c_{new} >$$

- q : état courant.
- c : caractère sous la tête de lecture.
- c_{new} : caractère à écrire.
- $Mouv$: mouvement vers la *gauche* ou la *droite* de la tête de lecture à effectuer.
- q_{new} : état suivant après instruction.

3.3.2 Modélisation Formelle

Une machine de Turing est composée de :
- $\sum$: ensemble fini de symboles d'entrée
- Γ : ensemble fini de symboles du ruban donc
  1. $\sum \subset \Gamma$
  2. $B \in \Gamma, B \notin \sum$(symbole blanc) Contenu implicite de toutes les cases du ruban sauf de l'input.
- S : ensemble fini d'états
- $s_0 \in S$: état initial
- $stop \in S$: état d'arrêt
- $\delta S\Gamma \rightarrow S$ {G,D} Γ : fonction de transition (*fini*)

Exécution

Au début, on a des données sur le ruban et le reste est rempli du symbole blanc. L'état initial de la tête est sur le premier caractère du ruban.
Le résultat est le contenu sur le ruban à l'état stop.

Positionner tête de lecture à droite

état	symb.	état	mouv.	symb.
début	0	début	D	0
début	1	début	D	1
début	B	report	G	B

Addition

état	symb.	état	mouv.	symb.
report	0	stop	G	1
report	1	report	G	0
report	B	stop	G	1

FIGURE 3.1 – Tableau d'exécution pour l'addition binaire

On peut boucler en machine de Turing, par exemple on ne fait qu'aller vers la droite. Donc la machine de Turing est complet.

3.3.3 Fonction T-calculables

Une fonction f est T-calculable si et seulement si il existe une machine de Turing qui, recevant comme donnée (*une représentation décimale de*) n'importe quel nombre naturel x, nous fournit comme résultat $f(x)$ si il est défini. Et ne se termine pas si $f(x) = \perp$
On peut également étendre cela aux fonctions $\mathbb{N}^n \to n$. Ce sont des définitions proche de l'idée de la calculabilité pour *Java*.

3.3.4 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

Le but de ces machines de *Turing* est d'être à l'essentielle. On est à l'essence même des principes de calculabilité. De plus, la complexité est plutôt simple via le concept de transition.

3.3.5 Extensions des Machines de Turing

On va donc modifier le modèle de base des machines de Turing. Mais est-ce plus *puissant* (peut calculer plus de fonctions) et plus *efficace* (moins d'étape à réaliser).

Changement de convention

On peut maintenant faire bouger la tête de plusieurs cases et on a plusieurs états stop. On a donc la même puissance mais une accélération du processus.
On peut aussi limiter l'alphabet en ayant un alphabet binaire. Toujours même puissance et efficacité. Mais cela va *limiter* le nombre d'état possible donc va enfaite limiter la machine.

On peut aussi imaginer un ruban uni-directionnel donc aller que vers la gauche. Toute machine de Turing peut être ré-exprimer dans ce type de machine. On perd en puissance et ralentissement linéaire.
On peut avoir un ruban multi case. Donc même puissance et même efficacité.

On peut avoir plusieurs rubans avec chacun ayant sa propre tête de lecture. Donc chaque tête est indépendante (*thread*). On aura toujours la même puissance mais une accélération quasi quadratique. Ou bien avec plusieurs tête, même résultat que précédemment.

Turing non-déterministe

On peut imaginer des relations au lieu de transitions. Via ce système, on sait déterminer si un élément appartient à un ensemble ou non.

$$\Delta \subseteq S\Gamma S\{G, D\}\Gamma$$

Il y a donc plusieurs exécutions possibles.
Si $A \subset \mathbb{N}$, A est NDT-récursif si 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 qui fournit un résultat $= 1$. Ou bien toutes les exécutions possibles fournissent 0.

A est *NDT-récursivement énumérable* si il existe une ND-machine de Turing telle que lorsqu'elle reçoit comme donnée n'importe quel nombre naturel x, il existe une exécution fournissant 1 si $x \in A$

et un résultat différent de 1 sinon.

Cela a la même puissance qu'une machine de Turing. Donc on peut toujours créer une machine de ND-Turing qui est équivalent à une machine de Turing et vice versa.
Une ND-machine de Turing peut accélérer de manière exponentielle b^n où b est le facteur de branchement. Attention, ce *speedup* ne peut pas être utilisé car aucune machine peut avoir un parallélisme infini !

3 états spéciaux

Il existe la Machine avec oracle :

1. $oracle_{ask}$: on demande si l'entier *représenté* à droite de la tête de lecture appartient à l'ensemble A.
2. $oracle_{yes}$: l'entier appartient à A
3. $oracle_{no}$: l'entier n'appartient pas à A

Donc, l'état suivant n'est pas spécifié dans l'instruction. Le nouvel état sera $oracle_{yes}$ ou $oracle_{no}$
En terme de puissance, on est équivalent **si** A est récursif. Si ce n'est **pas récursif** alors mon modèle est *plus puissant* que la machine de Turing. Je peux programmer la fonction *HALT*. (c'est bien évidemment un modèle abstrait).

C'est pour voir les possibilités. On peut établir les indécidables.

Machine de Turing universelle

On veut établir une machine de Turing qui soit un interpréteur de machines de Turing. On peut arriver à cela juste en utilisant les symboles $\{0, 1\}$. Bien évidemment, cette machine est *extrêmement* complexe. L'ensemble des codes de machine de Turing est un ensemble récursif.

On peut construire l'interpréteur avec 3 rubans :

1. : codage de la machine de Turing interpréteur
2. : les données
3. : résultat intermédiaire de l'interpréteur

On utilise donc 2.5.1, $\exists z \forall n, x : \varphi_z(n, x) = \varphi_n(x)$. $\varphi_n(x)$ est la fonction calculée par la machine de Turing de code n et φ_z la fonction universelle calculable. MT_z est le programme universel (*interpréteur*)

L'arrêt

On peut démontrer, en utilisant le modèle des machine de Turing que le problème de l'arrêt est non calculable. Même méthode de démonstration, mais adaptée aux machines de Turing. La preuve a été faite avec des machines de Turing pour la première fois.

Chapitre 4

Logique

4.1 Logique des propositions

On va décrire la syntaxe des formules logiques.
Son but premier est de :
— Représenter des connaissances
— Raisonner de manière automatique sur ces connaissances
Il existe la logique de **propositions**, des **prédicats**, **temporelle**, **floue**, **modale**, ...

4.1.1 Composition de la logique de propositions

Type	Description
Syntaxe	On détermine la structure accepté dans un langage
Sémantique	On leur donne un sens à ces formules
Raisonnement	On propose une méthode pour manipuler les formules et obtenir des informations pertinentes

4.1.2 Description

C'est la logique "*la plus simple*" et la plus ancienne.
On se base sur le principe de proposition. Ce sont des affirmations qui sont vraies ou fausses. (ex : *le soleil brille*, *le sommet* V_2 *du graphe a la couleur rouge*, ...)

On représente ces affirmations par une **suite de caractères** qu'on peut appeler *variable propositionnelle*
On lie ces variables via des connecteur logique (ET, OU, ...)

Symboles

— *Constante* : True et False.
— *Variables propositionnelles* : nos chaines de caractères.
— *Connecteur logique* : ne pas $\neg$, conjonction $\wedge$, disjonction $\vee$, implication $\Rightarrow$ et équivalence $\Leftrightarrow$
— *Parenthèses*

Formules propositionnelles

Une *constante logique* et une *variable propositionnelle* sont toutes les deux des formules propositionnelles. On peut donc combiner des formules propositionnelles pour en créer de nouvelle.

4.1.3 Convention

On supprime les parenthèses les plus extérieures pour essayer de réduire le nombre de parenthèse.
La précédences des opérateurs est importante et permet de supprimer des parenthèses : 1. $\neg$ 2. $\wedge$ 3. $\vee$ 4. $\Rightarrow$ 5. $\Leftrightarrow$
On a un ordre d'associativité qui définisse l'ordre de deux connecteurs identiques successifs. $\wedge$ et $\vee$ sont associatifs à gauche et $\Rightarrow$ et $\Leftrightarrow$ sont associatifs à droite.

Exemple

$A \wedge B$	$(A \wedge B)$	$\neg A \wedge B$	$((\neg A) \wedge B)$	$A \wedge \neg B$	$(A \wedge (\neg B))$
$A \vee B \vee C$	$((A \vee B) \vee C)$	$A \Rightarrow B \vee C$	$(A \Rightarrow (B \vee C))$	$A \Rightarrow B \Rightarrow C$	$(A \Rightarrow (B \Rightarrow C))$

4.2 Sémantique

Correspond à la signification d'une formule propositionnelles. Le sens d'une formule dépend d'un *contexte* donc des valeurs propositionnelles.
On appelle ce *contexte* une interprétation. Une interprétation d'une formule est une *assignation* d'une valeur de vérité (*true ou false*) à chacune des variables propositionnelles de la formules.
Donc si on a n variables, il y a donc 2^n interprétations possibles.

4.2.1 Valeur de vérité

Selon une interprétation I, on dérive la *valeur de vérité d'une formule* selon les règles suivantes :
- La valeur de vérité des constantes True et False sont respectivement *true* et *false*
- La valeur de vérité d'une variable propositionnelle A est la valeur de cette variable dans l'interprétation I.

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

4.2.2 Modèle

C'est une interprétation dont la formule est vrai.
Le sens d'une formule propositionnelle, donc sa sémantique, est défini par les différents modèles de cette formule.
Donc elle est défini par **tous les modèles de cette formule**. La signification d'une formule propositionnelle est déterminée par l'ensemble de ses modèles.

4.2.3 Formules équivalentes

Deux formules (composés des mêmes variables propositionnelles) sont donc équivalentes si elles ont les mêmes modèles.
Une *sous-formule* peut toujours être remplacée par une *sous-formule* équivalente sans forcément changer sa *sémantique*.

$\neg(\neg p)$	p	$p \wedge q$	$q \wedge p$	$p \vee q$	$q \vee p$
$p \Leftrightarrow q$	$(p \Rightarrow q) \wedge (q \Rightarrow p)$	$p \Rightarrow q$	$\neg p \vee q$	$p \Rightarrow q$	$\neg p \Rightarrow \neg q$
$\neg(p \vee q)$	$\neg p \wedge \neg q$	$\neg(p \wedge q)$	$\neg p \vee \neg q$	$p \vee (q \wedge r)$	$(p \vee q) \wedge (p \vee r)$
		$p \wedge (q \vee r)$	$(p \wedge q) \vee (p \wedge r)$		

On va s'intéresser à l'équivalence de Morgan qui $\neg(p \wedge q) \rightarrow \neg p \vee \neg q$

Démonstration de Morgan

On commence par dénombrer le nombre de possibilités : on a 2 variables donc $2^2 = 4$ possibilités

p	q	$\neg(p \wedge q)$	$\neg p \vee \neg q$
true	true	false	false
true	false	true	true
false	true	true	true
false	false	true	true

4.3 Ensemble SAT

L'ensemble des formules **Satisfaisable**. On veut être en mesure d'effectuer des raisonnements sur différentes formules.
Mais, on a le problème de la satisfiabilité comment savoir si une formule F a un modèle. Le modèle est une "*solution*" qui rend cette formule F vraie.

Et en conséquence logique, si on a notre formule F, quelles sont les conséquences de F.
Quelles formules sont-elles vraies si F l'est.

4.3.1 Satisfiabilité

Une formule propositionnelle est satisfaisable si elle possède au moins un modèle. Si elle n'en possède **aucun** alors elle est non satisfaisable, on appelle cela une contradiction.
L'ensemble SAT est donc l'ensemble des formules propositionnelles satisfaisables

Valeur de vérité

Étant donné une interprétation et une formule propositionnelle, sa valeur de vérité peut être déterminée par un *algorithme*.
Pour une formule avec n variables propositionnelles, il existe donc 2^n interprétations possibles.
L'ensemble **SAT** est donc récursif.
exemple de formules non satisfaisables : $(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (\neg A \vee C)$

Tautologie

C'est une formule qui est toujours vraie dans toutes les interprétations. ex : $A \vee \neg A$.
p est donc une tautologie si et seulement si $\neg p$ est **non satisfaisables**.

4.3.2 Conséquence logique

La formule q est une conséquence logique d'une formule p si la formule $p \Rightarrow q$ est une *tautologie*.

1. Donc tous les modèles de p sont aussi des modèles de q.
2. La formule q est vraie dans tous les modèles de p.

On note cela : $p \models q$

propriété

$p \models q$ ssi $(p \wedge \neg q)$ est *non satisfaisable*
Preuve : $p \models q$ ssi :

- $p \Rightarrow q$ est une tautologie
- $\neg(p \Rightarrow q)$ est non satisfaisable
- $\neg(\neg p \vee q)$ est non satisfaisable
- $(\neg\neg p \wedge q)$ est non satisfaisable
- $(p \wedge \neg q)$ est non satisfaisable

Attention : si q n'est pas une conséquence logique de p, on n'a pas nécessairement que $\neg q$ est une conséquence logique de p.
On peut avoir $p \not\models q$ et $p \not\models \neg q$. Donc $p \Rightarrow q$ et $p \Rightarrow \neg q$ ne sont pas des tautologies. ex :$(A \vee B) \not\models A$ et $(A \vee B)) \not\models \neg A$

4.4 Modélisation

Problème : on veut colorier les noeuds d'un graphe de manière à ce que deux noeuds adjacents aient des couleurs différentes.

4.4.1 Formalisme

On un graphe $G = (V, E)$ où V est l'ensemble des noeuds et E les arrêtes. On a un ensemble C de couleurs.
On définit les variables propositionnelles suivantes :

- couleur $[v, c]$ (distinct) avec $v \in V$ et $c \in C$
- La variable de couleur $[v, c]$ est vraie si le noeud v est de couleur c. Faux sinon.
- Il y a donc $\#V \times \#C$ variables propositionnelles distinctes.

Représentation

Notre modèle est composée d'une **conjonction de formules**.

- Pour chaque arrête $(v, w) \in E$ on impose que les deux noeuds n'aient pas la même couleur via : $\neg(\text{couleur}[v, c] \wedge \text{couleur}[w, c])$
- Pour chaque noeuds $v \in V$, la disjonction des couleurs $[v, c]$ sur toutes les couleurs $c \in C$ nous assure que chaque noeud a au moins une couleur
- Pour chaque noeud $v \in V$ la conjonction va s'assurer que chaque noeud n'a pas plus d'une couleur : $\neg(\text{couleur}[v, c_1] \wedge \text{couleur}[v, c_2])$ on utilise des paires de couleurs $[c_1, c_2]$

Affirmation

Si j'ai un modèle pour une conjonction de formule, j'ai donc une solution (le modèle) pour ce problème.
Si la formule n'est pas satisfaisable alors il n'existe pas de solutions pour ce problème.
Donc comment savoir si le problème est faisable ? Voir si notre formule est **satisfaisable**. Si elle l'est, on peut retourner le modèle.

4.5 Raisonnement

Lorsqu'on veut raisonner, il se pose à nous deux problèmes :

1. $p \models q$ Une formule q est-elle une conséquence logique d'une formule p
2. $p \in SAT$ donc existe-t-il au moins 1 modèle pour cette formule p

On peut réduire le 1. au 2. via : $p \models q$ ssi $(p \wedge \neg q) \notin SAT$

On peut utiliser des *solveurs SAT* qui sont des algorithmes de décision pour SAT.
Ou bien, on peut utiliser l'*inférence* qui est une méthode formelle utilisant des règles d'inférences.

4.5.1 Solveur SAT

C'est un modèle élémentaire basé sur le **model checking**. Il prend en *input* une formule p qui a n variables propositionnelles.
L'algorithme consiste à énumérer parmi les 2^n interprétations possibles pour p.
Si p est vraie pour une interprétation, on retourne $< true, I > (p \in SAT)$(I étant l'interprétation)
Si on ne trouve aucune interprétation correcte, on retourne $< false, \varnothing > (p \in SAT)$

Solveur efficace

Ce type de solveur travaille sur une forme simplifiée des formules propositionnelles : les formules sous forme normale conjonctive (CNF) :

- Négation uniquement devant une *variable propositionnelle*
- Que des disjonctions et des conjonctions
- La formule est une suite de conjonctions dont chaque élément est une disjonction de variables ou de négation de variables.

ex : $(\neg A \vee \neg C) \wedge (\neg A \vee D) \wedge (\neg B \vee \neg C) \wedge (\neg B \vee D) \wedge (A \vee B \vee C \vee \neg D)$ Donc entre chaque conjonction, on a des disjonctions de variables.
Nomenclature :

- *Literal* : est une variable propositionnelle A ou sa négation $\neg A$
- *Clause* : est une disjcontion de littéraux
- *CNF* : est une conjonction de *clauses* $c_1 \wedge ... \wedge c_n$
- *Formule CNF* : on représente une telle formule comme : $\{c_1, ..., c_n\}$

Transformation en CNF

On peut toujours représenter une formule sous forme *CNF*. il faut suivre ces étapes :

1. Éliminer les $\Leftrightarrow$ et $\Rightarrow$ en utilisant les équivalences (voir 4.2.3)
2. Mettre les symboles de négation dans des littéraux en utilisant la double négation et les équivalences de Morgan
3. Utiliser la distributivité sur $\wedge$ et $\vee$

Exemple :$(A \vee B) \Leftrightarrow (\neg C \wedge D)$ $(\neg(A \vee B) \vee (\neg C \wedge D)) \wedge (\neg(\neg C \wedge D) \vee (A \vee B))$
$((\neg A \wedge \neg B) \vee (\neg C \wedge D)) \wedge (C \vee \neg D \vee A \vee B)$ $(\neg A \vee \neg C) \wedge (\neg A \vee D) \wedge (\neg B \vee \neg C) \wedge (\neg B \vee D) \wedge (A \vee B \vee C \vee \neg D)$

4.5.2 Résolution

On transforme la formule propositionnelle en appliquant des règles de transformations successives. On réalise des règles d'inférence à chaque étape et on garantit que chaque formule résultante est équivalente à la formule initiale.

Exemple de règles

Nom	prémices	prémices	conclusion
Équivalence	$p \Leftrightarrow q$	$q \Leftrightarrow r$	$p \Leftrightarrow r$
Modus ponens	$p \Rightarrow q$	p	q

Si on a un ensemble de formules propositionnelles F qui représentante une Formules. Si on a des prémices qui appartiennent à cet ensemble, on peut ajouter des conclusions pour en faire F' et dire que $F = F'$

Résolution

Les règles d'inférence unique généralisant le *modus ponens*. Via cela, on peut décider si un ensemble F de clauses est satisfaisables ou non. Forme la plus simple :$p \vee q \qquad \neg p \vee r \rightarrow q \vee r$.
Si on réalise une table de vérité, on peut voir que *tout modèle des prémices est aussi un modèle de la conclusion.*

Règles

Si on a 2 clauses prémices C_1 et C_2, la première contenant $L_1 = V$ et la seconde $L_2 = \neg V$. On a une clause conclusion contenant tous les littéraux de C_1 et de C_2 sauf les littéraux L_1 et L_2
Exemples :

$$\begin{array}{cccc} A \vee B & A \vee B \vee \neg C & A \vee B & A \\ \neg A & \neg D \vee \neg B \vee F & \neg A \vee C & \neg A \\ \hline B & A \vee \neg C \vee \neg D \vee F & B \vee C & false \end{array}$$

Exécution

À partir d'une formule CNF sous la forme de clause. on va à chaque étape :

1. Appliquer la règle de résolution sur deux clauses pour lesquelles la règle est applicable et dont la conclusion n'est pas une clause existante
2. Ajouter la clause conclusion à la formule CNF

Si à une certaine étape, on ajouter la clause *false* alors la formule n'est pas satisfaisable.
Si la règle de résolution ne peut plus s'appliquer alors la formule n'est pas satisfaisable.

Exactitude

On sait qu'à chaque étape, on a une formule *CNF* qui est équivalente à la formule précédente.
On a un nombre de clauses distinctes, la formule est fini et donc son exécution se termine toujours.
Si une clause est *false* alors la formule n'est pas satisfaisables.
Si fin sans clause *false* alors la formule est non satisfaisable.

Résultat

La résolution nous indique si une formule *CNF* appartient ou n'appartient pas à SAT. Si elle appartient à SAT, la résolution ne donne pas de modèle comme résultat.
On l'utilise pour prouver que $p \models q$ On transforme $(p \wedge \neg q)$ en *CNF* :

— *false* : alors $(p \wedge \neg q) \notin SAT$ donc $p \models q$
— *se termine* : alors $(p \wedge \neg q) \in SAT$ donc $p \not\models q$

Exemple : $p = \{A, \neg B, \neg A \vee \neg C \vee D, \neg E \vee C, B \vee C\}$ mais $p \models D$? On applique la résolution à $p \cup \{\neg D\}$:

$$\begin{array}{cccc} \neg A \vee \neg C \vee D & \neg A \vee \neg C & \neg C & \neg B \\ \neg D & A & B \vee C & B \\ \hline \neg A \vee \neg C & \neg C & B & false \end{array}$$

Donc $p \models D$

Chapitre 5

Complexité algorithmique

5.1 Notions

On peut demander qu'un programme est juste ou encore qu'il soit **efficace**. C'est la calculabilité et complexité respectivement. On regarde de manière pratique.

5.1.1 Mesure

On a deux types de complexité

1. Complexité spatiale, l'espace mémoire.
2. Complexité temporelle, le temps d'exécution.

La complexité spatiale est *toujours* inférieur à la complexité temporelle.

Définition

La complexité d'un problème est la complexité de l'algorithme le plus efficace résolvant ce problème.

5.1.2 Comment mesurer ?

La complexité dépend de la taille des données, de la représentation des données et du modèle de calcul utilisé

On a 3 grandes mesures :

1. Meilleur cas
2. Cas moyen
3. Pire cas

5.1.3 Calcul de complexité

Pour la *complexité d'un algorithme*, on peut soit utiliser des *benchmarks* soit des analyses d'algorithmes.

Pour la *complexité d'un problème* on utilise la complexité d'un algorithme le résolvant ou l'*induction*.

Benchmark

On mesure le *temps d'exécution* d'un certains programmes avec des certaines données sur une même machine. Ainsi, on peut comparer les machines et programmes entre eux

Analyse mathématique

On compare la complexité du programme à une fonction mathématique. On trouve une fonction $T(n)$ exprimant la complexité. $T(n)$ correspond au temps d'exécution du programme et est obtenu en analysant le texte du programme.

Réduction

On peut résoudre des problèmes en ayant un algorithme qui en résout un autre.

5.2 Notation Grand O

Tout temps d'exécution peut être résumé à une fonction polynomial. On regarde le meilleur et pire cas ainsi que le cas moyen.

On se contente souvent que de la complexité dans le pire des cas. Cela nous donne une borne supérieure.

5.2.1 grand O

Cela nous permet une caractérisation concise de classes de complexité. Cette notation est indépendante des évolutions technologiques.
On peut négliger les termes étant différent du terme à l'exposant le plus élevé. On ignore également le coefficient devant l'inconnu.

5.2.2 Définition

$O(g(n))$: le nombre d'étapes exécutés par le programme pour une donnée de taille n n'est pas plus élevé que $cg(n)$ pour des tailles suffisamment grande.

$$T(n) \text{ est } O(g(n)) ssi \exists c, n_0 \forall n \geqslant n_0 : |T(n)| \leqslant c|g(n)| \tag{5.1}$$

5.2.3 Problèmes

La notation grand O n'est pas très précise. $O(n^3)$ n'indique pas si on peut borner par $O(n^2)$ par exemple.

Borne Inférieure

La borne inférieure est $\Omega(g(n))$. Si $f(n) = O(g(n))$ alors $g(n) = \Omega(f(n))$

Borne Exacte

La borne exacte est $\Theta(g(n))$. Si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$ alors $g(n) = \Theta(g(n))$

5.3 Hiérarchie de complexités

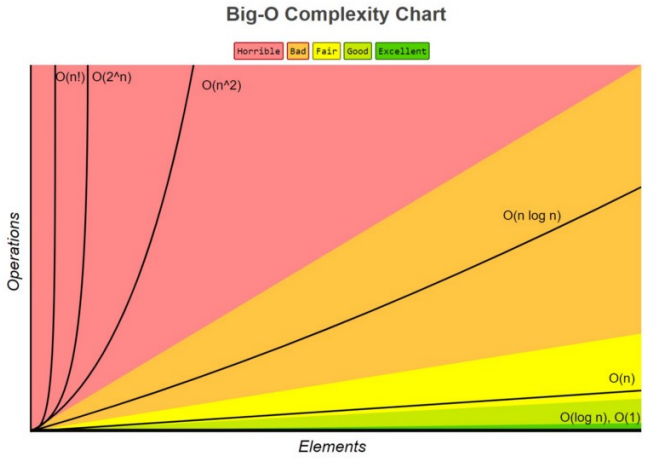


FIGURE 5.1 – Graphe de notation grand O

Algorithmes quadratiques :

1. Tri par échange
2. Bubble sort
3. Addition de matrices $n \times n$

Algorithmes cubiques :

1. Multiplication de matrices $n \times n$
2. Perspective d'une image 3D

Algorithmes exponentiels

1. Voyageur de commerce
2. Coloration de graphes
3. Problèmes de planification
4. Problèmes d'ordonnancement

5.4 Problème intrinsèquement complexe

Ce sont les problèmes, qui étant calculables, ne peuvent être résolu à part pour des entrées très petites.

Si un problème n'a pas de complexité *polynomiale*, alors il est pratiquement infaisable. Sinon il est pratiquement faisable.

Un problème est intrinsèquement complexe *ssi* il **n'existe pas** d'algorithme de complexité polynomiale. Un problème intrinsèquement complexe est donc pratiquement infaisable.

5.4.1 Influence du modèle de calcul

Si un algorithme, exprimé dans un modèle de calcul particulier, est de complexité polynomiale, alors il sera également de complexité polynomiale dans un autre modèle de calcul (spatiale et temporelle).

Hypothèse de modèle de calcul réaliste (pas de non déterminisme).

La classe de problème intrinsèquement complexes sont indépendants du modèle de calcul !

5.4.2 Influence de la représentation des données

Elle n'influence en aucun cas avec la propriété de problèmes intrinsèquement complexes.

Chapitre 6

Classes de complexité

6.1 Réduction et ensemble complet

6.1.1 Réduction

Le but est de déduire un algorithme pour un problème P à partir d'un algorithme résolvant P'.

Ici, on se limite au problème de décision pour un ensemble E à partir d'un algorithme de décision pour E'.

Via cette définition, on va pouvoir :

1. Retourner des informations sur la calculabilité :
   (a) Permet de prouver la calculabilité
   (b) Permet de prouver la non calculabilité
   (c) Permet d'analyser le degré de non calculabilité
2. Retourner des informations sur la complexité :
   (a) Permet de déduire la complexité
   (b) Permet d'analyser le degré de complexité

Ainsi, on va utiliser 3 méthodes de réduction :

1. La réduction algorithmique : permet de déduire des propriétés sur la calculabilité
2. La réduction fonctionnelle : permet de déduire des propriétés sur la complexité
3. La réduction polynomiale : permet de déduire des propriétés sur la complexité

6.1.2 Algorithmique

Un ensemble A est algorithmiquement réductible à un ensemble B si à partir d'un *algorithme* permettant de reconnaitre B, on peut construire un algorithme permettant de reconnaitre A.

Un ensemble A est algorithmiquement réductible à un ensemble B ($A \leqslant_a B$) si en supposant B récursif, alors A est récursif.

$\leqslant_a$ est une relation réflexive et transitive. Cela induit des classes d'équivalence.

6.1.3 Ensemble complet

Un ensemble complet d'une classe d'ensembles est l'ensemble le "*plus difficile*" à décider parmi tous les ensembles de la classe. Si on trouve un algorithme pour cet ensemble, alors on peut reconnaitre tous les autres ensembles de la classe. C'est le problème point rouge.

Définition

Ce point rouge d'une classe A est un problème E qui est **A-complet** par rapport à une relation de réduction :

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

Un problème E est **A-difficile** par rapport à une relation de réduction : $\forall B \in A : B \leqslant E$ il peut être en dehors de l'ensemble A.

6.2 Propriétés

— Si $A \leqslant_a B$ et B récursif, alors A récursif
— Si $A \leqslant_a B$ et A non récursif, alors B non récursif
— $A \leqslant_a \overline{A}$
— $A \leqslant_a B \Leftrightarrow \overline{A} \leqslant_a \overline{B}$
— Si A récursif alors pour tout B, $A \leqslant_a B$
— Si $A \leqslant_a B$ et B récursivement énumérable, alors A pas nécessairement récursivement énumérable.

Exemples

$$DIAG \leqslant_a HALT \qquad\qquad HALT \leqslant_a DIAG$$

6.3 Fonctionnelle

6.3.1 Définition

Un ensemble A est fonctionnellement réductible à un ensemble B ($A \leqslant_f B$) ssi il existe une fonction totale calculable f tq :

$$a \in A \Leftrightarrow f(a) \in B \tag{6.1}$$

Pour savoir si un élément appartient à A il suffit de calculer $f(a)$ et tester si $f(a) \in B$.

6.3.2 Propriétés

— Si $A \leqslant_f B$ et B récursif alors A récursif
— Si $A \leqslant_f B$ et A non récursif alors B non récursif
— Si $A \leqslant_f B$ récursivement énumérable, alors A récursivement énumérable
— $A \leqslant_f B \Leftrightarrow \overline{A} \leqslant_f \overline{B}$
— Si A récursif, alors pour tout $B, A \leqslant_f B$
— Pas nécessairement $A \leqslant_f \overline{A}$
— Si $A \leqslant_f B$ alors $A \leqslant_a B$
— $A \leqslant_a B$ n'implique pas nécessairement $A \leqslant_f B$

Cela est plus restrictif car on doit voir s'il y a une appartenance à B tel que $f(a) \leftarrow B$ et on doit l'exécuter.

6.3.3 Exemples

$$DIAG \leqslant_f HALT \qquad\qquad HALT \leqslant_f DIAG$$

6.3.4 Différences entre algorithmique et fonctionnelle

En $\leqslant_a$ on a un point de vue calculabilité et pour $\leqslant_f$ est un point de vue complexité. On doit donc appliquer un schéma d'algorithme suivant pour la *complexité*. On a un input a, $a' := f(a)$ et si $a' \in B$ alors l'output est $a \in A$. On utilise en dernier lieu et une seule fois le test $f(a) \in B$. On ne fait rien après ce test.

6.4 Modèle de calcul

On théorie "*classique*" la complexité se mesure via le modèle de machine de Turing. On regarde le nombre de *transitions* nécessaires pour la complexité temporelle et le nombre de cases utilisées pour la complexité spatiale. La notion de *transition* n'est pas très adapté pour les langages modernes.

On va utiliser le modèle des programmes Java. Donc déterministes et non déterministes. (on peut parfois des écarts pour utiliser le modèle le plus adapté)

Thèse

Tous les modèles de complexité ont entre eux des complexités spatiales et temporelles reliées de façon polynomiale.

Si un problème est *pratiquement faisable* dans un modèle alors il l'est aussi dans un autre.

6.5 Classes de complexité

6.5.1 Modèle déterministe

On va déterminer des *ensembles* de classe de problèmes qu'on regroupe par leur O.

DTIME(f) : la famille des ensembles récursifs pouvant être décidés par un programme Java de complexité temporelle $O(f)$.

DSPACE(f) : la famille des ensembles récursifs pouvant être décidés par un programme Java de complexité spatiale $O(f)$.

En complexité, on **n'a pas** de programme qui boucle.

Attention : $DTIME_{Java}(n^2) \neq DTIME_{Turing}(n^2)$ on sait juste qu'un complexité polynomiale en Java **sera** polynomiale en Turing mais pas égale !

6.5.2 Modèle non-déterministe

NTIME(f) : la famille des ensembles ND-récursifs pouvant être décidés par un programme Java non déterministe de complexité temporelle $O(f)$. On considère la complexité de la branche d'exécution **la plus longue**. Donc toutes les branches sont finies et de complexité $O(f)$.

NSPACE(f) : la famille des ensembles ND-récursifs pouvant être décidés par un programme Java non déterministe de complexité spatiale $O(f)$.

6.5.3 Classe

Classe P : $P = \cup_{i \geqslant 0} DTIME(n^i)$ qui regroupe donc tous les programmes récursifs pouvant être décidé par une complexité temporelle polynomiale. Le fait que ce soit Java ou Turing ne change pas la classe.

Classe NP : $NP = \cup_{i \geqslant 0} NTIME(n^i)$ qui est la famille des ensembles **ND-récursifs** pouvant être décidés par un programme Java *non-déterministe* de complexité temporelle polynomiale.

6.6 Relation entre les classes

6.6.1 Rapport déterminisme et non-déterminisme

Si $A \in NTIME(f)$ alors $A \in DTIME(c^f)$. Il faut s'imaginer la façon dont on parcours l'arbre, de gauche à droite et de haut en bas. C'est une perte exponentielle d'efficacité.
Si $A \in NSPACE(f)$ alors $A \in DSPACE(f^2)$.

6.6.2 Rapport temps et espace

$si A \in NTIME(f)$ alors $A \in NSPACE(f)$ $\quad$ $si A \in DTIME(f)$ alors $A \in DSPACE(f)$
$si A \in NSPACE(f)$ alors $A \in NTIME(c^f)$ $\quad$ $si A \in DSPACE(f)$ alors $A \in DTIME(c^f)$

6.6.3 Hiérarchie de complexités

Pour toute fonction totale calculable f, il existe un ensemble A récursif tel que $A \notin DTIME(f)$. On prouve cela par diagonalisation et même résultat pour NTIME, DSPACE et NSPACE.

Il existe toujours des problèmes plus complexes qu'une complexité donnée.

Chapitre 7

NP-complétude

Les problèmes intrinsèquement complexes ont une complexité *non polynomiale*.

On veut savoir si il existe un algorithme non déterministe polynomiale, existe-t-il alors un algorithme déterministe polynomial résolvant ce même problème ?

Si un ensemble A appartient à NP, A appartient-il alors à P ?

$$P = NP \tag{7.1}$$

7.1 NP-complétude

On sait que $P \subseteq NP$. Mais comment démontrer que $NP \subseteq P$ ou pas. On peut prendre un élément **E** qui est "*représentatif*" de NP et essayer de démontrer tel que :

— $E \in P$ alors on démontre que $P = NP$

— Ou $E \notin P$ alors $P \neq NP$

Il faut choisir un élément qui soit NP-complet par rapport à une relation de réduction $\leq$

— $E \in NP$

— $\forall B \in NP : B \leq E$

7.1.1 Réduction algorithmique

Si on réalise $\leq_a$. Si $A \leq_a B$ en supposant que A et B sont récursifs.

Si on arrive à montrer que $E \in P$ on ne sait pas si $P = NP$. Car pour les autres éléments de NP, la réduction ne permet pas d'affirmer que ceux-ci sont dans P.

7.1.2 Réduction fonctionnelle

Si $A \leq_f B$ si il existe une fonction totale calculable f telle que $a \in A \Leftrightarrow f(a) \in B$. Si on arrive à montrer que $E \in P$ on ne sait pas si $P = NP$. Car pour les autres éléments de NP, la réduction ne permet pas d'affirmer que ceux-ci sont dans P . (le calcul de $f(a)$ peut prendre un temps non polynomial). On doit raffiner la réduction fonctionnelle.

7.1.3 Réduction polynomiale

Un ensemble A est polynomialement réductible à un ensemble $B(A \leq_p B)$. Si il existe une fonction totale calculable f de complexité temporelle polynomiale telle que :

$$a \in A \Leftrightarrow f(a) \in B \tag{7.2}$$

Propriétés

1. $\leq_p$ est une relation réflexive et transitive
2. Si $A \leq_p B$ et $B \in P$ alors $A \in P$
3. Si $A \leq_p B$ et $B \in NP$ alors $A \in NP$

7.1.4 NP-complétude

Un problème E est NP-**complet** (par rapport $\leq_p$) si

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

Un problème E est NP-**difficile** (par rapport $\leq_p$) si

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

7.1.5 Propriétés

Soit E est un ensemble NP-complet.
— $E \in P$ ssi $P = NP$
— $E \notin P$ ssi $P \neq NP$
— Si $E \leq_p B$ et $B \in NP$ alors est NP-complet.

7.2 Théorème de Cook

SAT : satisfaction des formules propositionnelles
Soit $W(A_1, ..., A_n)$ une formule propositionnelle dont les variables sont $A_1, ...A_n$.
Si on a la formule $(\neg A \vee B) \Rightarrow (A \wedge B)$ cette formule est satisfaisable avec A = B = true.

7.2.1 Formalisation

On utilise les connecteurs logiques, on a des variables A_i. On réalise des formules qui s'écrivent $W(A_1, ..., A_m)$ qui sont satisfaisables si il existe des valeurs logiques qui permettent que cette formule soit vrai.

Si on a une longueur d'une formule W, on a n occurrence de variables, alors la taille de $W = O(nlogn)$.

7.2.2 Théorème SAT NP-complet

Donc $SAT \in NP$. Il existe un programme *non déterministe* de complexité polynomiale capable de reconnaitre si une formule $W(A_1, ..., A_m)$ est satisfaisable.

1. On génère une séquence de m valeurs logiques. $O(m)$ tel que $m \leqslant n$.
2. On substitue les occurrences des variables A_i par leur valeur. $O(nlogn)$
3. On évalue l'expression via la méthode de *réduction*. Complexité *polynomiale*.

$\forall B \in NP : B \leqslant_p SAT$

Soit $B \in NP$. Il existe une **machine de Turing** non déterministe M qui reconnait en un temps polynomial $p(n)$ si un élément x appartient à B :
— p est un polynôme
— n est la longueur de la donnée x
Donc la machine M dans toutes les branches d'exécutions possibles, s'arrêtent avant $p(n)$ mouvements.
La donnée x est transformée en une formule *propositionnelle* W_x :

— La taille de W_x : $O(p(n))$ symboles
— Transformation polynomiale : la construction de W_x peut se faire en un temps *polynomial* par rapport à n.
— $x \in B \Leftrightarrow W_x \in SAT$

On va construite la formule W_x. À partir d'une donnée x et d'une machine de Turing non déterministe M, d'une formule W_x dont les variables caractérisent le fonctionnement de la machine M sur la donnée x.

7.3 Problèmes NP-complets

Comment **prouver** qu'un problème est NP-complet. On peut soit

1. Réaliser une preuve comme à celle pour SAT (assez complexe) (il faut montrer que tous les problèmes peuvent s'y réduire)
2. Utiliser la propriété : $siE \leqslant_p B$ avec E NP-complet et $B \in NP$ alors B est **NP-complet**. (méthode la plus utilisée)

Tous les problèmes considérés dans cette classe sont des problèmes de *décision.*

En pratique, on est intéressé par des *algorithmes* qui construisent des solutions.

Problème du circuit Hamiltonien HC

HC = ensemble des *graphes* possédant un circuit hamiltonien (parcourant une seule fois tous les sommets).

Le problème d'un circuit eulérien est lui dans P.

Problème du voyageur de commerce TS

Si nous avons n noeuds et qu'on les relies via des arcs pondérés, Existe-t-il un circuit reliant les n noeuds (villes) qui soit de longueur inférieur ou égal à un entier.

$HC \leqslant_p TS$.

Chemin le plus long entre deux sommets

Si on a un graphe avec dans les noeuds a et b. On veut savoir s'il existe un chemin reliant a et b qui est supérieur ou égal à un entier. (*l'inverse aurait été P*)

3SAT

Problème de la satisfaisabilité de formules de la forme $C_1 \wedge C_2 \wedge ... \wedge C_n$ avec :

$$C_i = L_{i1} \wedge L_{i2} \wedge L_{i3} \qquad L_{i1} = A_k \text{ ou } L_{i1} = \neg A_k \qquad A_k = \text{ variable propositionnelle}$$

Forme normale conjonctive avec 3 variables par clause.

Couverture de sommets VC

Si on a un graphe et un entier positif B. Existe-t-il un sous-ensemble de sommets de taille $\leqslant B$ qui couvre tous les arcs du graphe.

Le problème de la couverture des arcs est dans P.

Problèmes de la clique

Si on a un graphe et un entier positif B. Existe-t-il un sous-ensemble des sommets de taille $\geqslant B$ qui soit une clique (chaque paire de noeuds de ce sous-ensemble est reliée par un arc)

Autres

1. Nombre chromatique d'un graphe
2. Problème de partition

7.4 P = NP

On a énormément chercher à prouver qu'il existe des algorithmes polynomiaux pour les problèmes NP-complets. Donc on pense que la solution est $P \neq NP$.

7.4.1 En pratique

Si un problème est *NP-complet*. On sait que le problème n'a pas de solution algorithmique polynomiale. Mais il s'agit de la complexité dans le *pire des cas*. On pourrait avoir une solution polynomiale pour certaines données.

Résoudre

— Changer le problème en un problème plus simple. Particulariser le problème pour certaines instances
— Utiliser un algorithme exponentiel si la plupart des instances à résoudre sont de complexité polynomiale
— Utiliser la technique d'exploration, mais en se limitant à un nombre polynomial de cas (heuristique). Algorithme incomplet.
— Si problème d'optimisation, calculer une solution approximative.

Transition de phase

Beaucoup de problèmes NP-complets sont caractérisés par un *paramètre* de contrôle. Cela induit 3 zones

1. Instances où presque tout est solution (quasi toujours *oui*)
2. Transition entre les deux régions
3. Instances où presque tout est solution *non*.

Le **pic de difficulté** se trouve dans la *transition* et est **indépendant** de l'algorithme qui est choisi.

7.4.2 Autres classes de complexité

Classe EXPTIME

Famille des ensembles *récursifs* pouvant être décidées par un programme de complexité temporelle $O(2^p)$ où p est un polynôme.

Classe LOGSPACE

Famille des ensembles *récursifs* pouvant être décidées par un programme de complexité spatiale $O(log)$.

Classe PSPACE

Famille des ensembles *récursifs* pouvant être décidées par un programme de complexité spatialle $O(p)$ où p est un polynôme.

Classe NPSPACE

Famille des ensembles *ND-récursifs* pouvant être décidées par un programme de complexité spatialle $O(p)$ où p est un polynôme.

Mais : $PSPACE = NPSPACE$.

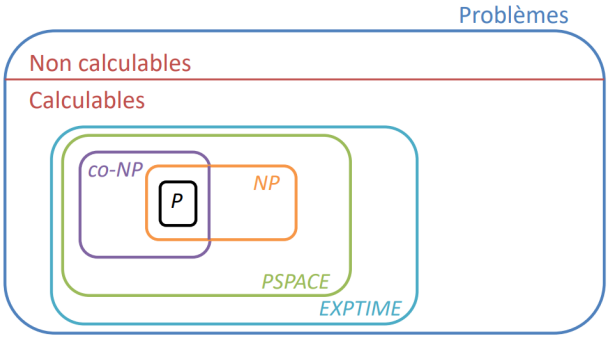


FIGURE 7.1 – relation entre classe

On a une classe intéressante qui est $NPI = NP \setminus NPC \setminus P$ qui est une sorte de *no man's land*.

Chapitre 8

Analyse et perspectives

8.1 Thèse de Church Turing

Rappel : toute fonction calculable par une machine de Turing est donc calculable (logique). Toute fonction effectivement *calculable* est calculable par une machine de Turing.

La deuxième partie est considérée vraie par de nombreuses évidences.

8.1.1 Évidences heuristiques

La définition des machines de Turing s'approche de la notion intuitive de procédé effectif. On sait que toutes fonctions particulières s'avérant effectivement calculables ont été montrées être calculables par une machine de Turing.

Il y a énormément de démonstration pour tout plein de classe de problème. On a toujours échoué à prouver qu'une fonction *intuitivement calculable* ne soit pas calculable sur une machine de Turing.

8.1.2 Équivalences des formalismes

Les différentes définitions des mêmes formalismes de calculabilité ont conduit à la définition de la **même classe** de fonction calculables.

8.1.3 Mécanisme de Calcul

On a montré que tous les mécanismes de toutes les machines constructibles par la mécanique **newtonienne** ne peuvent calculer que des fonctions programmables.

8.2 Formalismes de Calculabilité

Qu'est-ce qu'un **bon** formalisme ? Il doit être :

1. **SD : Soundness des défintions :** Toute fonction D-calculable est calculable. Si il existe une description $d \in D$ calculant une fonction f alors cette fonction est calculable. (*première partie de C-T*)
2. **CD : Complétude des définitions :** Toute fonction calculable est D-calculable. Si une fonction f est calculable, alors il existe une description $d \in D$ calculant cette fonction f. (*deuxième partie de C-T*)
3. **SA : Soundness algorithmique :** L'interpréteur de D est calculable. Toute description $d \in D$ peut être effectivement exécutée sur des données. Le formalisme D est algorithmique. Cette caractéristique assure que D décrit bien des procédés effectifs.

4. **CA : Complétude algorithmique :** Si P est SA, alors il existe un compilateur (*calculable*) tel que étant donné $p \in P$, ce compilateur appliqué à P produit une description $d \in D$ avec p équivalent à d. Cela assure une équivalence des formalismes.
5. **U : description universelle :** L'interpréteur de D est D-calculable. Le formalisme doit permettre de décrire son propre interpréteur. (*Hoare-Allison*)
6. **S : propriété S-m-n affaiblie :** $\forall d \in D \quad \exists S : d(x,y) = [S(x)](y)$. Il existe un transformateur de programme (*total calculable*), qui recevant comme données un programme d à 2 arguments et une valeur x, fournit comme résultat une programme à 1 argument tel que les deux fonctions calculent la même chose.

8.2.1 Propriétés

$SA \Rightarrow SD$	Il existe un interpréteur de D qui est calculable SA.Donc calculable via SD
$CA \Rightarrow CD$	
SD et $U \Rightarrow SA$	propriétés U rajoutées
CD et $S \Rightarrow CA$	
SA et $CD \Rightarrow U$	
CA et $SD \Rightarrow S$	
S et $U \Rightarrow S-m-n$	
SA et $CA \Rightarrow SD$ et CD et U et S	
SA et CD et $S \Rightarrow CA$ et SD et U	

On peut avoir une partie des formalismes mais cela rend notre formalisme mauvais. Bloop était SD, SA et S mais pas le reste.

8.3 Techniques de raisonnement

Comment démontrer la **non calculabilité** d'une fonction. Voici un schéma de raisonnement :

1. Essayer de trouver un algorithme qui résout le problème
2. Se demander si suspicion possible de non *calculabilité* du problème. (aucun algorithme)
3. Essayer de prouver la non calculabilité
4. Si c'est le cas, essayer de définir une version du problème approchée qui est calculable.

8.3.1 Techniques de preuve

On va utiliser Rice : s'intéresser à la fonction du programme conduit souvent à la non calculabilité.
Démonstration directe par la non calculabilité via une diagonalisation ou une preuve par l'absurde.
Méthode de réduction :

1. Réduire la solution d'un problème A à celle d'un autre problème A' (A est un cas particulier de A')
2. À partir d'un algorithme pour A' construire un algorithme pour A.
3. Conclusion, si A' est calculable alors A l'est aussi. Si A est non calculable alors A' est non calculable.

8.3.2 Problème intrinsèquement complexe

Une complexité exponentielle est juste une borne supérieure et peut avoir des algorithmes polynomiaux.

On peut essayer de faire une réduction d'un problème NP à notre problème.

Existe-t-il un algorithme ND polynomial. Si oui, le problème est dans NP.

Résoudre

On utilise un algorithme exponentiel si la plupart des instances à résoudre sont de complexité polynomiale.

Changer le problème en un problème plus simple (polynomial) qui est proche du problème initial.

Utiliser une technique d'exploration, mais en se limitant à un nombre polynomial de cas (*heuristique*). L'algorithme est incomplet. Il peut donner une réponse erronée ou approximative. Mais c'est parfois mieux que pas de réponse du tout.

Si problème d'optimisation, calculer une solution approximative.

8.4 Aspects non couverts par la calculabilité

Une grande parties des programmes **ne peuvent pas** être résumées au calcul de fonction : IoT, OS, app, ...

La définition de la calculabilité cherche si il existe un programme qui calcule cette fonction. On veut voir les **conséquences** de la non calculabilité.

8.4.1 Ressources

Une fonction calculable en *théorie* peut être non calculable en *pratique* (trop de ressources, ...). La calculabilité ne détermine pas ça.

8.5 Au delà de la calculabilité

8.5.1 Calculabilité et intelligence humaine

Si un être humain particulier est capable de calculer une fonction particulière (*en un temps fini*). Alors on dit que la fonction est *H-calculable*. Cela est-il équivalent à la calculabilité d'un ordinateur ?

8.5.2 Thèse : version procédés publics

Si une fonction est H-calculable et que l'être humain calculant cette fonction est capable de décrire (à l'aide du langage) à un autre être humain sa méthode de calcul de telle sorte que cet autre personne soit capable de calculer cette fonction, alors la fonction est T-calculable Cette version n'exclut pas l'existence de fonctions H-calculables, mais non calculables

8.5.3 Thèse CT : version réductionniste

Si une fonction est H-calculable, alors elle est T-calculable. Le comportement des éléments constitutifs d'un être vivant peut être simulé par une machine de Turing. Tous les processus cérébraux dérivent d'un substrat calculable

8.5.4 Thèse CT : version anti-réductionniste

Certains types d'opérations effectuées par le cerveau, mais pas la majorité d'entre elles, et certainement pas celles qui sont intéressantes, peuvent être exécutées de façon approximative par les ordinateurs. Certains aspects seront cependant toujours hors de portée des ordinateurs.

$$H - calculable \neq T - calulable \tag{8.1}$$

8.5.5 Calculabilité et intelligence artificielle

Il y a 2 types :
- **IA forte :** simulation du cerveau humain à l'aide d'un ordinateur en copiant le fonctionnement. On veut des résultats similaires.
- **IA faible :** programmation de méthodes et techniques de raisonnement en exploitant les caractéristiques propres des ordinateurs.

IA forte $\approx$ version réductionniste de CT.

Puissance cerveau humain

Un cerveau humain fait 10^{18} instructions par seconde. Le plus puissant fait 442 PFlops (10^{15}). Qui correspond à environ 10^{18}.

Machine et la pensée

On peut réaliser le test de Turing. On pose des questions au pc et à l'être humain par écrit. Si la personne qui pose les questions ne peut déterminer la différence entre les deux alors on atteint la singularité et l'ordinateur réussie le **test de Turing** (perdre 1 fois sur 2) et donc les machines peuvent penser.

8.6 Au delà des modèles de calculs

La frontière entre calculabilité et non calculabilité est indépendante du progrès technologiques.

Mais si on a des nouvelles technologies, peut-on dépasser ces limitations ?

8.6.1 Ordinateur quantique

Il ne donne pas un seul résultat mais une **superposition de résultat** avec un probabilité.

Une nouvelle classe est **BQP** ou Bounded-error Quantum Computing. Si il existe un algorithme *quantique* polynomial qui donne la bonne réponse au moins 2 fois sur 3.

L'espace BQP ne comprend pas les problèmes NP-Complet mais dépasse dans les problèmes PSPACE et NP.

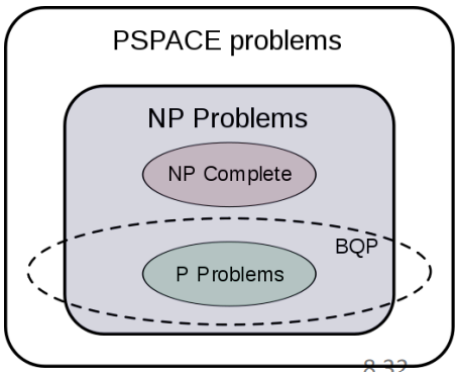


FIGURE 8.1 – Espace *supposée* des relations BQP et NP

Chapitre 9

Questions Test d'entrée

9.1 TP1

1. Effectivement, il existe une bijection entre les N et les nombres impairs positifs → en somme il existe une fonction qui transforme les N en impair positif
2. J'imagine qu'il y a une bijection mais je ne vois pas quel formule passant de N aux impairs existent car c'est le propre des nombres impairs
3. Même raisonnement que la question 1
4. La fameuse formule qui lie N et Q car Q est juste une paire de N
5. Effectivement, sachant la diagonalisation de Cantor il est simple de le prouver
6. Pour N dans N il en existe une infinité et l'ensemble d'arrivé ne change pas grand-chose car on s'intéresse au nombre de fonction.
7. Effectivement, on a un nombre fini de langage donc de mot. Cela est dû grâce à l'alphabet fini et la longueur fixe. Donc on sait énumérer
8. Question typique vu au cours. En effet comme on a une infinité et il n'existe aucune bijection depuis les naturels etc.
9. Même cardinalité = bijection, il ne peut y avoir de bijection entre un ensemble non énumérable et énumérable
10. Une infinité de nombre mais effectivement même cardinalité car tous peuvent être ramené aux naturels.

9.2 TP2

1. Effectivement, on ne doit pas être en capacité de coder l'algorithme pour que la fonction soit calculable.
2. Nombre premier est récursif car on a le crible d'eratosthène.
3. Si un ensemble **X** est récursif (donc donne 1 ou 0) alors $\bar{X}$ l'est également car il inverse les 1 et 0.
4. Si **X** est récursivement énumérable (donne 1 ou quelque chose d'autre mais pas 1) et que son opposé $\bar{X}$ est récursivement énumérable. Alors bien évidemment **X** est récursif.
5. Oui, trivial.
6. Non, juste au simple fait que si **X** est récursivement énumérable alors $\bar{X}$ ne peut être récursivement énumérable.
7. Vrai car on a une fonction calculable car ce sont des combinaisons linéaires de calculables.
8. Voir sous-section 2.2.1.

9. Oui car être énumérable c'est dire qu'on peut compter tous les résultats même si ça prend un temps infini.
10. Voir sous-section 2.2.1.

9.3 TP3

1. En effet, si on a que des fonctions totales, on sait qu'on aura toujours une réponses pour n'importe quelle input.
2. Oui, le théorème de *Hoare-Allison* ne dit pas l'inverse et pour le sous langage :

   ```
   P = return 1;
   ```

3. Mais si on peut avoir la fonction halt en L, on ne peut avoir sa fonction *interpret*
4. Mais, on peut calculer cette fonction *interpret* avec un langage de programmation qui n'est pas restrictif.
5. Effectivement on ne peut pas calculer toutes les fonctions totales avec L.
6. Il existe un langage qui peut calculer sa fonction halt et son interpréteur (le langage vide trivial)
7. Effectivement, ne pas être récursif n'empêche pas d'être récursivement énumérable.
8. Faux exemple : $\overline{\mathbf{K}}$. (voir 2.4.1)
9. Non, on peut imaginer 2 fonctions non récursives qui se "*complètent*" et comblent les lacunes de chacune. Exemple : K et $\overline{K}$
10. Non, on peut imaginer une intersection qui ne comportent que des entrées avec une réponse.

9.4 TP4

1. Faux, l'ensemble des programmes qui ont pour fonction $2n$ est non récursif car notre ensemble à toutes les fonctions qui ont ce résultat.
2. Faux, car on possède toutes les fonctions qui renvoient 0 pour n'importe quelle entrée.
3. Vrai, car certains programmes peuvent hors de notre ensemble ont la même fonction. C'est le *1000 instructions* qui posent soucis. Car des programmes qui en ont + peuvent retourner le même résultat.
4. Faux, en effet un programme qui prend 3 entrées (donc hors de notre ensemble) a le même résultat qu'un faisant partie de notre ensemble. Et ceci est valable pour **tous** programmes dans notre ensemble.
5. Cela dépend de la fonction f, en effet, même sorte de raisonnement que pour le 4. donc cela dépend. On peut aussi avoir des fonctions non calculables.
6. Il faudrait tester toutes les entrées pour en être sûr donc $\mathbb{N}$.
7. Oui, voir l'idée avec halt, ...
8. C'est l'ensemble formé par Halt donc pas récursif.
9. Mais c'est bien récursivement énumérable comme Halt.
10. Non, il faut faire une réduction à halt.

9.5 TP5

à ajouter

9.6 TP6

pas présent

9.7 TP7

Les 5 premières questions sont des applications *calculatoires* de machine de Turing. Simplifiez vous la vie en utilisant ce site et utilisez ce code.

```
input: 'bbba'
blank: ' '
start state: start
table:
  start:
    'a'    : {write: 'x', R: seekB}
    'b'    : {write: 'x', R: seekA}
    'x'    : {write: ' ', R: start}
    ' '    : {write: 1, L: done}
  seekA:
    'a'         : {write: 'x', L: restart}
    ['b', 'x']: {R: seekA}
    ' '         : {L: falseFun}
  seekB:
    'b'         : {write: 'x', L: restart}
    ['a', 'x']: {R: seekB}
    ' '         : {L: falseFun}
  restart:
    ['a', 'b', 'x'] : {L: restart}
    ' '               : {R: start}
  falseFun:
    ['a', 'b', 'x'] : {write: ' ', L: falseFun}
    ' '               : {write: 0, L: done}
  done:
```

1. Vrai c'est le principe même et le but d'une machine de Turing, montrer qu'un langage de programmation est récursif.
2. Faux, on peut penser à un tableau simple de Turing qui bouge le ruban une fois à droite puis à gauche.
3. Faux, non-déterministes donc plusieurs applications et "futures" possibles. Cependant, cela n'avantage en aucun sa puissance, juste sa rapidité et facilité d'implémentation.
4. Faux, on n'a pas besoin d'un oracle qui *voit une case devant* pour savoir si une entrée est paire. L'ensemble des nombres pairs est **récursifs** donc oracle inutile.
5. Vrai, car on serait incapable sans oracle de savoir si un programme ne boucle pas donc on veut avoir **HALT**.

9.8 TP8

pas fait oopss

9.9 TP9

1. On a une complexité de $O(n^2)$ car on a une double boucle for. Pas la borne exacte mais la plus proche.
2. On a une complexité de $O(ln(n))$ car on divise le travail en deux à chaque itération.
3. On a une complexité de $O(n)$ car on divise et double le travail donc on reste à n.
4. Petit indice dans son nom "*explosion*", on a $O(2^n)$ car à chaque entrée on dédouble le tout et on est récursif.
5. Vrai puisque les bornes donnent une limite supérieure.
6. Faux car on peut avoir un $g(n)$ d'un degré inférieur que $f(n)$ ce qui par l'*Hospital* nous donnerait une limite *infinie*.
7. Faux c'est en $O(n^3)$
8. Faux, être intrinsèquement complexe est une complexité *non-polynomiale*.
9. Faux. En effet, pour de faibles tailles de données on peut facilement trouver le résultat.
10. Faux. C'est l'utilité de cette notation. Un pc 100 fois plus puissant nous permettrait que d'augmenter la taille de l'entrée de 2 ou 3.

9.10 TP10

1. Faux, pas forcément !
2. Vrai, une réduction $\leqslant_a$ peut permettre de prouver qu'un ensemble est récursif
3. Faux, mais ne peut pas prouver que ce n'est pas récursif
4. Vrai, on peut prouver qu'un ensemble est récursif via $\leqslant_f$
5. Vrai
6. Faux, car on ne sait pas si C n'est pas le problème le plus compliqué de l'ensemble.
7. Faux, non on ne sait pas obtenir une borne exacte
8. Vrai
9. Faux, car la complexité est plus élevé sur une machine de Turing
10. Faux, Non il y a d'autres problèmes qui ne peuvent pas être résolu par des algorithmes non polynomiaux qui ne sont pas dans les **NP**.

Chapitre 10

Question cours

10.1 S1

10.2 S2

Une densité du nombre de rationnel Un langage est un ensemble de chaine de caractère ! Attention de ne pas confondre langage le mot vulgaire et langage en informatique.
Fonction totale s'intéresse au domaine.
Fonction surjective s'intéresse à l'image.

— Injective max une flèche.
— Surjective min une flèche.
— Étendre une fonction est rajouté une flèche.

10.3 S3

Attention à la notation entre$\rightarrow$ $P_{12} \neq P_{47}$ car P sont des chaines de caractères bien déterminés et donc différentes $\varphi_{12} = \varphi_{47}$ effectivement car phi est créé depuis des P différents mais peuvent créer des fonctions donnant le même résultat

$Halt(n, 37)$ n'est pas calculable car si ce serait, halt serait calculable
Print(1) car tous les Entiers appartiennent à $\mathbb{N}$

Recursif : calculable Récursivement énumérable : calculable et ne dit pas s'il ne l'est pas.

10.4 S4

Différence entre Q et JAVA, toutes les fonctions se terminent et rendent une valeur. Java lui peut rendre une fonction bottom ! Donc notre démonstration ne fonctionne plus car $interpret(k, k) = \perp$ et donc $diag_{mod} = \perp + 1 = \perp$

Est-ce le fait qu'un interpréteur ne puisse être écrit en Q problématique ? Un interpréteur prend en entrée un code et met des mots-clés. L'interpréteur va créer un arbre d'exécution.
SQL est un langage qui ne boucle jamais. $\rightarrow$ Donc SQL ne peut être utiliser pour faire des programmes complexes.

Impossible de détecter les cas sans réponses. (fonction HALT en exemple)

10.5 S6

1. Pourquoi la propriété S est-elle une conséquence du théorème S-m-n ? Avec le théorème S-1-1 Il existe une fonction totale calculable $S_1^1 : N^2 \to N$
2. Lesquelles de ces affirmations sont vraies
   (a) Si f est une transformateur de programmes (f fonction totale calculable), alors il existe deux programmes P1 et P2 tels que $f(P1) = P2$ ainsi que P1 et P2 calculent la même fonction Le théorème de u point fixe
   (b) Si f est une transformateur de programmes (f fonction totale calculable), alors il existe deux programmes P1 et P2 tels que f(P1) = P2 ainsi que P1 et P2 calculent la même fonction totale → le totale nous force à avoir quelque chose qui se termine toujours
3. À quoi sert le théorème du point fixe :
   (a) Faire un programme qui sans paramètre donne son propre code en sortie standard
4. Post → impossible même avec Bruteforce. Diophantienne → l'utilité est minime.
5. Nombre réels non-calculables est-elle une conséquence immédiate du fait que R est non énumérable. Donne un output fini aussi précis que je le voudrais. R n'est pas énumérable et j'ai un nombre énumérable de programme donc je ne peux pas représenter tous les R.

10.6 S7

— Sémantique : signification du programme et sa fonction calculé
— Syntaxique : comment écrire correctement le programme et qu'il soit compréhensible par l'ordinateur

Si un ensemble A est ND-récursif, les différentes exécutions du programme ND-Java qui décide cet ensemble doivent-elles nécessairement se terminer pour chaque input possible.→ ce sont des arbres de décisions (accepter les résultats si oui ou non etc)

Tant qu'une façon d'avancer dit oui même avec une boucle infini, on peut accepter.
Quel est le but d'un langage non déterministe car pas exécutable sur un pc ? Car sinon on va saturer et on n'aura pas assez de cœur et on passera en séquentiel. (ex : parcours en largeur) : C'est pour modéliser des problèmes

Avec BLOOP je peux calculer des choses mais pas toutes les fonctions totales et calculables (hoare allison)

Automate fini chose simple qui répète toujours la même chose en boucle sans dépendre du passé.

Par hoare allisson que les automates sont toujours finis et s'arrête

Si on a une façon acceptante, on peut toujours l'accepter et dire oui.

On ne peut pas programmer toutes les fonctions calculables en automate fini

Chapitre 11

Vrai ou Faux cours

Cette section regroupe toutes les réponses et raisonnements des questions **wooclap** posées au cours.

11.1 S1

Introduction donc pas de QCM.

11.2 S2

Besoin de contributeurs Je n'ai pas noté :((

11.3 S3

TODO ajouter les questions pas que les réponses

1. Tout language n'est pas récursif car c'est un ensemble de fonction dont Halt qui n'est pas récursif
2. Tout ensemble énumérables n'est pas récursif.
3. Un ensemble fini est récursif
4. Le complément d'un ensemble récursif est récursif
5. L'ensemble des rationnels est récursivement énumérable
6. Un sous-ensemble infini d'un ensemble récursivement énumérable n'est pas récursivement énumérable
7. Un sous-ensemble fini d'un ensemble récursivement énumérable n'est pas récursivement énumérable
8. L'union d'une infinité énumérable d'ensembles récursivement énumérable n'est pas récursivement énumérable : Exemple avec l'ensemble K
9. Le complément d'un ensemble récursivement énumérable n'est pas récursivement énumérable : Exemple avec K et inverse de K
10. Une fonction dont la table est infinie est calculable
11. Un algorithme ne calcule qu'une et une seule fonction
12. Il existe des ensembles non récursivement calculable
13. Une fonction calculable est peut être calculée par une infinité de programmes
14. L'ensemble HALT n'est pas récursivement énumérable

15. Il existe des ensembles récursifs qui sont récursivement énumérables
16. Il existe des ensembles récursifs qui ne sont pas énumérables
17. Si le domaine d'une fonction est finie, alors cette fonction est calculable : on peut faire une liste de tous les cas possibles
18. Si le domaine de fonction est infini, alors cette fonction est calculable

11.4 S4

1. Il existe un langage non trivial dans lequel la fonction halt est calculable. → on sait que tout ce fini en Q donc oui on sait calculer halt car c'est la fonction constante 1 (non trivial → pas les réponses facile type langage vide)
2. Il n'existe pas un langage de programmation (non trivial) dans lequel on peut programmer le fonction halt ainsi que l'interpréteur de ce langage
3. Si un langage de programmation (non trivial) permet de programmer son interpréteur, alors la fonction halt n'est pas calculable dans ce langage
4. Il existe de langage de programmation (non trivial) dans lequel toutes les fonctions calculées sont totales
5. Il n'existe pas un langage de programmation ne permettant de calculer que des fonctions totales, mais toutes les fonctions totales calculables → l'interpréteur ne peut pas être calculé en Q
6. il n'existe pas une fonction totale calculable qui n'est l'extension d'aucune fonction partielle calculable → effectivement car on a une fonction calculable totale elle est d'office l'extension d'une fonction partielle calculable. Il existe une fonction partielle calculable telle qu'aucune fonction totale calculable n'est une extension de cette fonction partielle –> énoncé du théorème, pas de messages d'erreur.

11.5 S5

1. L'ensemble des programmes Java calculant une fonction f telle que $f(10) = 10$ n'est pas récursif (car ici on caractérise que fait le programme !)
2. L'ensemble des programmes Java calculant une fonction f telle que $f(10) = 10$ est un ensemble récursivement énumérable. (n'a aucun rapport avec Rice ! ! !)
3. Toute propriété relative aux programmes est calculable (théorème de rice parle des propriétés calculés par la fonction pas lié à la fonction
4. Si A est un sous-ensemble (strict et non vide) récursif de programmes Java, alors toute fonction calculée par un programme de A n'est pas aussi calculée par un programme du complément de A (car il existe pas qu'il existe pour tout ! ! !)
5. Soit la fonction $revenu1_y de(n)$ = le revenu imposable de Yves Deville à l'année n. Si n est inférieur à 1960 ou supérieur à 2060, le résultat de cette fonction est $\perp$. Par hypothèse, une personne décédée a un revenu de 0. Cette fonction est-elle calculable ? (Car n'est définie que pour 101 inputs. Calculable car nb fini de point à calculer. Domaine finie == calculable au sens théorique du terme)
6. Soit la fonction $revenu2_y de(n)$ = le revenu imposable de Yves Deville à l'année n. Par hypothèse, une personne pas encore née ou décédée a un revenu de 0. Cette fonction est-elle calculable ? (car est constante sauf pour un certain input. Ici on n'est jamais bottom donc il existe toujours une fonction)
7. Un programme Java étant fini, l'exécution de ce programme sera aussi finie (car il peut boucler) Soient les programmes $P_3 2$

8. (n) dont le code est « print(1) » et $P_5 7(n)$ dont le code est « print(0) ». Cochez les affirmations correctes (on peut pas varier de l'un à l'autre)
9. Une extension d'une fonction partielle calculable est toujours calculable (faux car il existe des fonctions qui ne peuvent être étendue ex : HALT)
10. L'ensemble des sous-ensembles des entiers est énumérable (car autant que les réelles, on a besoin d'une chaine infinie de caractère)
11. Il existe un ensemble infini de chaînes finies de caractères (A-Z) qui est non énumérable (on peut créer des chaines de caractères de chiffres et ce sera fini)
12. Il existe un ensemble infini de chaînes finies de caractères (A-Z) qui est non récursivement énumérable (ensemble non récursif → complément K → c'est une représentation
13. La fonction $halt(18, x)$ est calculable (cela dépend de la numérotation choisie des programmes Java)
14. L'ensemble des fonctions non calculables est énumérable (Faux car infinité)
15. Soit A est un ensemble (infini) récursivement énumérable. Si $B \subseteq A$, alors B est aussi récursivement énumérable. (Faux car le sous-ensemble peut prendre que des choses bottoms)
16. Il existe des langages de programmation (non triviaux) dans lesquels toutes les fonctions calculées sont totales (Vrai, mini-Java ne calcule que des fonctions totales et calculables. Toutes les fonctions sont totales)
17. Si une fonction f est calculable, alors toute fonction g dont f est une extension est calculable (Faux, → le théorème de l'extension parle pas de ça. Car g peut être bottom et donc ne pas être calculable (genre 1 si élément appartient au complément de K ou $\perp \rightarrow$ fonction pas calculable et f est une extension))
18. Soit le programme numéro n calculant la fonction factorielle ($f(x) = x!$ si x non négatif et $f(x) = \perp$ si x négatif,). Cochez les affirmations correctes :
    (a) $\varphi_l(n)$ calcule bien quelque soit l'énumération choisie pour les programmes
    (b) Pas énumération car n est positif
    (c) Une infinité de programmes calculent la même fonction
    (d) Par hasard
    (e) Pas injectif car ne fait d'office correspondre au max 1 une fois. (penser à 0! Et 1! Même réponse)

11.6 S6

1. La propriété S-m-n affirme que tout numéro de programme calculable peut être transformé en un numéro équivalent, mais avec moins de paramètres → Faux, on transforme un programme en un autre programme. Les fonctions sont équivalentes.
2. Les propriétés S-m-n et S sont équivalentes → Faux, S-m-n implique S mais pas dans l'autre sens.
3. Tous les langages de programmation (standards) satisfont la propriété S-m-n, on peut spécialiser un programme par ses arguments
4. Le théorème du point fixe est une conséquence du théorème de Rice → Faux c'est l'inverse
5. Si deux programmes P1 et P2 calculent la même fonction, alors il existe un transformateur f de programmes (f fonction totale calculable) , tel que f(P1)=P2 → Vrai, ce n'est pas le théorème du point fixe. On a 2 programmes qui calculent la même fct, on une fct qui transforme P1 et P2 (return P2)
6. Si f est un transformateur de programmes (f fonction totale calculable), alors il existe deux programmes P1 et P2 tels que f(P1)=P2 ainsi que P1 et P2 calculent la même fonction → Vrai énoncé du point fixe mais un peu moins math.

7. Si f est un transformateur de programmes (f fonction totale calculable), alors il existe deux programmes P1 et P2 tels que f(P1)=P2 ainsi que P1 et P2 calculent la même fonction totale → Faux, attention au totale, on n'exige pas que les programmes s'exécutent toujours.
8. Le théorème du point fixe permet de démontrer que la fonction halt est non calculable → Vrai, pt-fixe donne halt
9. La non calculabilité du problème de Post est une conséquence du théorème de Rice → Faux, c'est une autre famille de problème non calculables, c'est lié au problème des chaines de caractères
10. L'ensemble des nombres réels calculables est énumérable → Vrai car on l'approche via un programme et le nombre de programme est calculable
11. Il existe une infinité non énumérable de nombres réels non calculables → Vrai les réels calculables sont énumérables donc l'ensemble opposé doit être non énumérable sinon ça voudrait dire que les réels sont énumérables.

11.7 S7

1. Toutes les fonctions calculables par les programmes du langage BLOOP sont totales → Vrai toujours terminé donc tot
2. Le langage BLOOP (Bounded Loop) ne permet de programmer que des fonctions totales. Il existe donc des fonctions totales calculables qui ne peuvent être programmées dans ce langage. → Vrai hoare allison
3. Les langages non déterministes permettent d'écrire des algorithmes plus efficaces. → Faux, plus simple de les exécuter mais on ne peut pas exécuter tout en parallèle
4. Tout ensemble ND-récursivement énumérable est récursif → Faux, ND et déterministe est la même chose. Cela n'apporte rien en calculabilité. ND facilite la description de problème. Donc est-ce un ensemble récursivement énumérable récursif ? Faux bien évidement
5. Il existe des ensembles récursifs ne pouvant être décidés par un automate fini → Vrai, HOARE ALLISON ! Tous les programmes se terminent
6. Tout automate fini non déterministe peut être transformé en un automate fini déterministe équivalent → Vrai, un programme non déterministe Java peut être transformé en déterministe en faisant étape par étape