Calculabilité & Complexité

Philippe Quéinnec

9 janvier 2025

Plan

1. Machines de Turing
   - Définitions
   - Variantes
   - Machines universelles
   - Fonctions calculables
   - Machines auto-reproductrices
2. Indécidabilité, incalculabilité
   - Indécidabilité de l'arrêt
   - Réduction
   - Autres problèmes indécidables
3. Castor affairé
4. Fonctions récursives
5. Problème de correspondance de Post
   - Définition
   - Langages & grammaires
6. Conclusion

Première partie

Introduction

Exemples de questions en calculabilité / complexité

- Conjecture de Collatz (1937) : le programme suivant termine-t-il pour une valeur donnée de n ? Pour toute valeur de n ?

  while $n > 1$ **do**

  if $(n\%2 = 0)$ **then** $n \leftarrow n/2$ **else** $n \leftarrow 3 * n + 1$

  (pour n donné : semi-décidable ; pour n quelconque : on ne sait pas)
- Les deux codes suivants sont-ils équivalents ?

  $r \leftarrow 1$; **while** $n > 0$ **do** $r \leftarrow r * n$; $n \leftarrow n - 1$; **done**

  let *foo* x = (**if** $x = 1$ **then** 1 **else** $x * foo(x - 1)$) **in** *foo* n

  ($\sim$ équivalents, décidable pour ces instances, indécidable en général)
- Y a-t-il une procédure pour vérifier si toute formule en logique des propositions est vraie ? En logique des prédicats ?

  (propositions : oui, par exemple table de vérité ; prédicats : non)
- Existe-t-il une solution efficace au problème du sac à dos ?

  (que signifie efficace ? a priori problème exponentiel)
- Multiplier deux matrices est-il plus facile que jouer au Go ?

  (que signifie facile ? calculable en temps polynomial pour l'un, en temps exponentiel pour l'autre)

Contenu du cours

Questions abordées

- Qu'est-ce qu'un problème ?
- Qu'est-ce qu'un algorithme ?
- Qu'est-ce qu'un calcul ?
- Peut-on savoir si un problème a une solution = est-il décidable ?
- Que signifie être efficace ?
- Y a-t-il des problèmes plus *difficiles* que d'autres ?
- Un problème possède-t-il un algorithme efficace ?

Ressources

Ce cours est inspiré des cours suivants :

- *Introduction à la calculabilité*, Pierre Wolper, Dunod, 2006
- *Langages formels, Calculabilité et Complexité*, Olivier Carton, éditions Vuibert, 2014
- *Calculabilité et complexité*, Anca Muscholl, 2018
  `http://www.labri.fr/perso/anca/MC.html`
- *Complexité algorithmique*, Sylvain Perifel, Ellipses, 2014
  `https://www.irif.fr/~sperifel/livre_complexite.html`
- *Fondements de l'informatique – Logique, modèles, et calculs*, Olivier Bournez, 2013–2020
  `https://www.enseignement.polytechnique.fr/informatique/INF412`
- *Computational Complexity : A Modern Approach*, Sanjeev Arora and Boaz Barak, Cambridge University Press, 2009
  (draft sur `http://theory.cs.princeton.edu/complexity/`)
- *Mathematics and Computation*, Avi Wigderson, 2019
  `https://www.math.ias.edu/avi/book`

Historique

Programme de Hilbert (1900–1920)

Montrer que les mathématiques sont cohérentes, complètes et décidables (démontrables).

cohérence : propriété d'un système formel dans lequel un énoncé et sa négation ne peuvent être démontrés vrais tous les deux. (notion universelle de la vérité)

complétude : propriété d'un système formel où tout énoncé vrai (dans ce système) est démontrable (dans ce système). (vérité = preuve)

décidabilité : existence, dans un système formel, d'un procédé systématique (algorithme) qui permet de déterminer la véracité/fausseté de tout énoncé démontrable. (automatisation des preuves)

Résultats

Théorème de complétude (Gödel, 1929)

La logique du premier ordre (logique des prédicats) est complète.

Premier théorème d'incomplétude (Gödel, 1931)

Tout système formel « un peu riche » (contenant la théorie des nombres) est soit incohérent, soit incomplet : *Cet énoncé est non démontrable.*

(plus exactement, $\exists P$ tel qu'à la fois $\mathcal{S} \cup \{P\}$ et $\mathcal{S} \cup \{\neg P\}$ sont cohérents)

Second théorème d'incomplétude (Gödel, 1931)

La cohérence d'un système formel (un peu riche) n'est pas démontrable au sein de ce système.

Idée de base : tout énoncé (propriété, démonstration) d'un système formel peut être codé par un nombre.

Résultats – suite

Turing (1935)

Tout système formel ≪ un peu riche ≫ est indécidable.

Théorème de Rice (1951)

Toute propriété sémantique non triviale d'un programme est indécidable.

Thèse de Church-Turing (1936), théorème de Kleene (1938)

Il y a équivalence entre :

- les fonctions *intuitivement* calculables
- les machines de Turing — *ordinateur*
- les fonctions récursives — *langage de prog.*
- le λ-calcul — *langage fonctionnel*
- les langages récursivement énumérables — *ens. de termes*

Deuxième partie

Machines de Turing

Plan

1. Machines de Turing
   - Définitions
   - Variantes
   - Machines universelles
   - Fonctions calculables
   - Machines auto-reproductrices
2. Indécidabilité, incalculabilité
   - Indécidabilité de l'arrêt
   - Réduction
   - Autres problèmes indécidables
3. Castor affairé
4. Fonctions récursives
5. Problème de correspondance de Post
   - Définition
   - Langages & grammaires

Machine de Turing

Septuplet $\mathcal{M} = (Q, X, \Gamma, \delta, q_0, F, \#)$ où :

- Q : ensemble fini d'états
- X : alphabet (fini)
- Γ : alphabet de bande, tel que $X \subset \Gamma$, et $\# \in \Gamma \setminus X$ (le blanc)
- $q_0 \in Q$: l'état initial de l'automate
- $F \subseteq Q$: les états finals (ou terminaux)
- $\delta \in Q \times \Gamma \mapsto Q \times \Gamma \times \{\leftarrow, -, \rightarrow\}$: fonction de transition.

Une machine de Turing possède une structure de stockage qui est un ruban linéaire *non borné*, et une tête de lecture positionnée sur une case.

Configurations et transitions

Configuration

mot $u\ q\ v$, avec $q \in Q$ et $u, v \in \Gamma^*$
(la tête de lecture est sur la première lettre de v)

···	#	a	b	b	a	b	a	#	···

(tête de lecture q ↓ sur le second a)

$= abb\ q\ aba$
$= \#abb\ q\ aba\#$
$= \#^*abb\ q\ aba\#^*$

Transitions

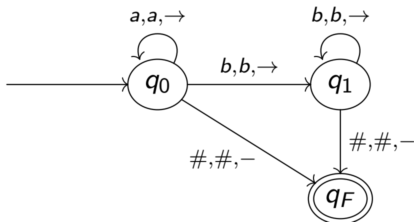
Relation $\vdash$ entre configurations :

$$\begin{array}{ll} uc\ q\ av \vdash uc\ q'\ \mathbf{b}v & \text{si } (q', b, -) = \delta(q, a) \\ uc\ q\ av \vdash uc\mathbf{b}\ q'\ v & \text{si } (q', b, \rightarrow) = \delta(q, a) \\ uc\ q\ av \vdash u\ q'\ c\mathbf{b}v & \text{si } (q', b, \leftarrow) = \delta(q, a) \end{array}$$

$\vdash^*$: fermeture réflexive transitive de $\vdash$

Exemple : $a^* b^*$

	$\#$	a	b	
q_0	$q_F, \#, -$	$q_0, a, \rightarrow$	$q_1, b, \rightarrow$	lit les a
q_1	$q_F, \#, -$		$q_1, b, \rightarrow$	lit les b
q_F				



$q_0 aabb \vdash aq_0 abb \vdash aaq_0 bb \vdash aabq_1 b \vdash aabbq_1 \# \vdash aabbq_F \#$
Langage accepté = mots conduisant à l'état final = $\{a^i b^j \mid i, j \geq 0\}$

Langage accepté et calcul

Langage accepté

L'ensemble des mots (ou suite de mots) qui conduisent à un état final.

$$L(\mathcal{M}) \triangleq \{m \in X^* \mid \exists q_F \in F : q_0 m \vdash^* m' q_F m''\}$$

Valeur calculée

Le contenu du ruban quand la machine s'arrête sur un état final.

$$\mathcal{M}(m) = m'm'' \text{ ssi } \exists q_F \in F : q_0 m \vdash^* m' q_F m''$$

Quand une machine n'a pas de transition possible dans une configuration donnée, elle s'arrête. Pour une entrée donnée, une machine a donc trois comportements possibles : s'arrêter sur un état final, s'arrêter sur un état non final, boucler indéfiniment.

Langage accepté et calcul – variante

On définit aussi $\mathcal{M}$ par un septuplet $(Q, X, \Gamma, \delta, q_0, \varnothing, \#)$

- $\varnothing$: acceptation
- $\delta \in Q \times \Gamma \mapsto (Q \times \Gamma \times \{\leftarrow, -, \rightarrow\}) \cup \{\varnothing\}$

Langage accepté :

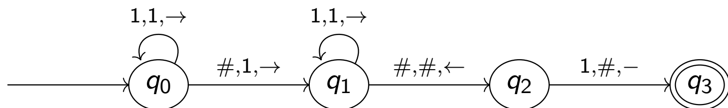
$$L(\mathcal{M}) \triangleq \{m \in X^* \mid q_0 m \vdash^* \varnothing\}$$

Valeur calculée : état du ruban à l'acceptation.

Exemple : addition unaire de $n + m$

Entrée sous la forme $1^n\#1^m$, sortie $= 1^{n+m}$

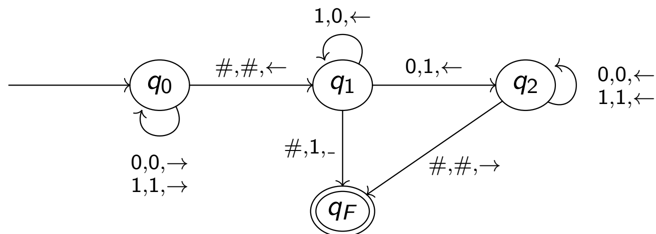
	$\#$	1	
q_0	$q_1, 1, \rightarrow$	$q_0, 1, \rightarrow$	parcourt 1^n et met 1
q_1	$q_2, \#, \leftarrow$	$q_1, 1, \rightarrow$	parcourt 1^m
q_2		$q_3, \#, -$	enlève le 1 de trop
q_3	$\emptyset$		



Exemple : calcul de $n+1$

Initialement sur le ruban : n codé en base 2.

	$\#$	0	1	
q_0	$q_1, \#, \leftarrow$	$q_0, 0, \rightarrow$	$q_0, 1, \rightarrow$	va à la fin
q_1	$q_F, 1, -$	$q_2, 1, \leftarrow$	$q_1, 0, \leftarrow$	incrémente avec retenue
q_2	$q_F, \#, \rightarrow$	$q_2, 0, \leftarrow$	$q_2, 1, \leftarrow$	retourne au début
q_F		$\varnothing$	$\varnothing$	

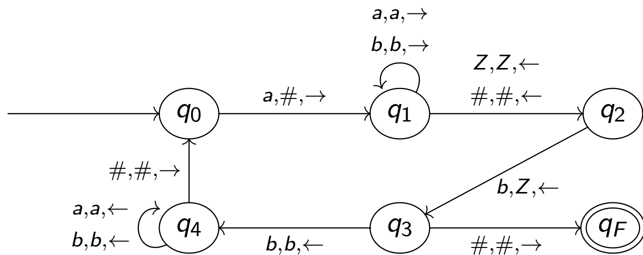


$$L(\mathcal{M}) = \{0,1\}^*$$
$$\mathcal{M}(n) = n+1$$

Exemple : $a^n b^n \rightarrow Z^n$

Machine de Turing définie pour $X = \{a, b\}$ et $\Gamma = \{a, b, Z, \#\}$:

δ	a	b	Z	$\#$	
q_0	$q_1, \#, \rightarrow$				efface le premier a
q_1	$q_1, a, \rightarrow$	$q_1, b, \rightarrow$	$q_2, Z, \leftarrow$	$q_2, \#, \leftarrow$	va à la fin de $a^* b^*$
q_2		$q_3, Z, \leftarrow$			remplace le dernier b par Z
q_3		$q_4, b, \leftarrow$		$q_F, \#, \rightarrow$	vérifie si fini
q_4	$q_4, a, \leftarrow$	$q_4, b, \leftarrow$		$q_0, \#, \rightarrow$	revient au début



Cette machine reconnaît $a^n b^n$ et calcule Z^n.

Une machine de Turing en Lego

https://www.dailymotion.com/video/xrn0yi

https://videotheque.cnrs.fr/doc=3001

Machine de Turing à n rubans

Machine possédant n rubans indépendants (une tête de lecture pour chaque ruban) : septuplet $\mathcal{M} = (Q, X, \Gamma, \delta, q_0, F, \#)$ où :

- Q : ensemble fini d'états
- X : alphabet (fini)
- Γ : alphabet de bande, tel que $X \subset \Gamma$, et $\# \in \Gamma \setminus X$ (le blanc)
- $q_0 \in Q$: l'état initial de l'automate
- $F \subseteq Q$: les états finals (ou terminaux)
- $\delta \in Q \times \Gamma^n \mapsto Q \times (\Gamma \times \{\leftarrow, - \rightarrow\})^n$: fonction de transition.

Note : la machine de Turing à n rubans avec têtes synchronisées $\delta \in Q \times \Gamma^n \mapsto Q \times \Gamma^n \times \{\leftarrow, - \rightarrow\}$ est trivialement équivalente à une machine mono-ruban : travailler sur l'alphabet Γ^n.

Expressivité des MT à n rubans

Les machines multi-rubans ont la même expressivité que les machines mono-ruban : $\forall \mathcal{M}$ MT à n rubans, $\exists \mathcal{M}'$ MT à 1 ruban telle que $L(\mathcal{M}') = L(\mathcal{M})$ et $\forall m : \mathcal{M}(m) = \mathcal{M}'(m)$.

Alphabet de $\mathcal{M}' = (\Gamma \times \{0,1\})^n$: coller les bandes en marquant par 1 la position de chaque tête :

				↓					
⋯	#	a	b	a	a	b	#	#	⋯
⋯	0	0	0	1	0	0	0	0	⋯
⋯	#	#	B	A	A	A	a	#	⋯
⋯	0	0	0	0	0	1	0	0	⋯

La simulation d'une transition de $\mathcal{M}$ consiste à :

1. parcourir le ruban pour noter dans l'état de contrôle de $\mathcal{M}'$ les symboles associés à la marque 1
2. en fin de bande (ou quand n symboles obtenus), choisir la transition de $\mathcal{M}$ correspondant à l'état $\times$ les symboles lus
3. parcours pour écrire les nouveaux symboles et déplacer les marques 1

(simulation quadratique : $\approx 2k^2$ déplacements dans $\mathcal{M}'$ pour simuler k déplacements de $\mathcal{M}$)

Variations

Multi-têtes

Les machines multi-têtes (mono-ruban) ont la même expressivité que les machines mono-tête.

Demi-ruban

Il est équivalent de définir les MT avec un demi-ruban + un symbole de blocage si on va à gauche de la première case.

Alphabet réduit

Il est équivalent que l'alphabet de bande soit réduit à $\{0, 1, \#\}$.

(intuition : coder en base 2 les symboles d'un alphabet plus étendu $\Rightarrow$ transformation logarithmique)

Mouvements

Il est équivalent de n'avoir que $\{\leftarrow, \rightarrow\}$ comme mouvements possibles.

Machine de Turing non déterministe

Sextuplet $\mathcal{M} = (Q, X, \Gamma, \delta, q_0, F)$ où :

- Q : ensemble fini d'états
- X : alphabet (fini)
- Γ : alphabet de bande, tel que $X \subset \Gamma$, et $\# \in \Gamma \setminus X$ (le blanc)
- $q_0 \in Q$: l'état initial de l'automate
- $F \subseteq Q$: les états finals (ou terminaux)
- $\delta \in Q \times \Gamma \times Q \times \Gamma \times \{\leftarrow, -, \rightarrow\}$: relation de transition.
  (ou $\delta \in Q \times \Gamma \mapsto \mathcal{P}(Q \times \Gamma \times \{\leftarrow, -, \rightarrow\}))$

Transitions et langage d'une MT non déterministe

Transitions

$$
\begin{array}{lll}
ucqav \vdash ucq'\mathbf{b}v & \text{si } (q', b, -) \in \delta(q, a) \\
ucqav \vdash uc\mathbf{b}q'v & \text{si } (q', b, \rightarrow) \in \delta(q, a) \\
ucqav \vdash uq'c\mathbf{b}v & \text{si } (q', b, \leftarrow) \in \delta(q, a)
\end{array}
$$

Langage accepté

L'ensemble des mots (ou suite de mots) qui conduisent à un état final par au moins une exécution.

$$L(\mathcal{M}) \triangleq \{m \in X^* \mid \exists q_F \in F : q_0 m \vdash^* m' q_F m''\}$$

Valeur calculée

Les contenus possibles du ruban quand la machine s'arrête.

$$m'm'' \in \mathcal{M}(m) \text{ si } \exists q_F \in F : q_0 m \vdash^* m' q_F m''$$

Expressivité d'une MT non déterministe

Les machines de Turing non déterministes ont la même expressivité que les machines déterministes : $\forall \mathcal{M}$ MT non déterministe, $\exists \mathcal{M}'$ MT déterministe telle que $L(\mathcal{M}') = L(\mathcal{M})$ et $\forall m : \mathcal{M}'(m) \in \mathcal{M}(m)$.

Idée : explorer, en largeur, l'arbre de calcul issu de la configuration initiale.
Numéroter $0..k-1$ les transitions T de $\mathcal{M}$. Un calcul est un mot $k_1 \cdots k_n$.
Ordonner par longueur puis par ordre lexicographique les mots sur T.
(ça revient à ordonner par la valeur numérique du mot en base k)

Construire une machine à 3 bandes :

- la première bande contient le mot d'entrée et n'est jamais modifiée
- la deuxième bande est la bande d'un calcul en cours
- la troisième bande est un calcul : un mot sur $\{0..k-1\}$
- répéter
  - copier la première bande sur la deuxième
  - simuler le calcul de la 3ᵉ bande appliqué à la 2ᵉ : appliquer successivement à la 2ᵉ bande les transitions lues sur la 3ᵉ bande
  - si état terminal $\Rightarrow$ mot accepté
  - si blocage ou calcul épuisé $\Rightarrow$ passer au calcul suivant ($=$ faire $+1$)

Nombre ou codage de Gödel

Une machine de Turing peut être complètement définie avec les neufs symboles # 0 1 G I D $\varnothing$, ; (G, I, D pour $\leftarrow, -, \rightarrow$)

Nombre ou codage de Gödel

Le *nombre* ou *code de Gödel* d'une machine de Turing est l'entier i en base 9 qui représente cette machine, notée $\mathcal{M}_i$.

	#	a	b	
q_0	$q_2, \#, -$	$q_0, a, \rightarrow$	$q_1, b, \rightarrow$	
q_1	$q_2, \#, -$		$q_1, b, \rightarrow$	
q_2	$\emptyset$			q_2 final

Numéroter les états et les coder en base 2, numéroter les symboles de l'alphabet et les coder en base 2, lister les transitions :

Code $= 10I\#, 00D0, 01D1; 10I\#, , 01D1; \emptyset, , ;$

(On pourrait aussi coder la machine en base 2)

Machine de Turing universelle

Machine universelle

Il existe une machine $\mathcal{M}_{univ}$ qui, ayant en entrée le codage $\langle\mathcal{M}\rangle$ d'une machine $\mathcal{M}$ et un mot m, calcule l'application de $\mathcal{M}$ à m.

Soit une machine à trois rubans :

- Premier ruban = codage $\langle\mathcal{M}\rangle$ de $\mathcal{M}$
- Deuxième ruban = simule le ruban de $\mathcal{M}$, initialement m
- Troisième ruban = état courant de $\mathcal{M}$, initialement le q_0 de $\mathcal{M}$

$\mathcal{M}_{univ}$ lit un symbole sur le deuxième ruban, utilise le troisième ruban pour trouver la transition de $\mathcal{M}$ à faire, écrit sur le troisième ruban le nouvel état et sur le deuxième ruban le nouveau symbole en déplaçant la tête de lecture de celui-ci.

Machine de Turing universelle

- Machine de Turing universelle $\approx$ intepréteur de programmes d'un langage $\mathcal{L}$, lui-même écrit en $\mathcal{L}$.
- On connaît des petites machines universelles :
  - 7 états, 4 symboles (Marvin Minsky, 1962)
  - 4 états, 6 symboles, 22 transitions (Yurii Rogozhin, 1996)
  - 2 états, 3 symboles, 6 transitions (Stephen Wolfram & Alex Smith, 2007)
- On connaît des machines universelles efficaces : si $\mathcal{M}$ s'arrête en T pas avec une entrée m, alors $\mathcal{M}_{eff}(\langle\mathcal{M}\rangle, m)$ s'arrête en $C \times T \log T$ pas, où C ne dépend que de la taille de l'alphabet de $\mathcal{M}$ et de son nombre d'états.

Thèse de Church-Turing

Thèse de Church-Turing (1936)

Il y a équivalence entre :

- les fonctions *intuitivement* calculables
- les fonctions calculables par une machine de Turing
- ...

Argumentaires :

1. Que signifie « intuitivement » ?
2. On n'a jamais réussi à prouver le contraire
   (démonstration par intimidation)
3. les machines de Turing sont équivalentes au λ-calcul, aux fonctions récursives, aux langages récursivement énumérables...
   $\Rightarrow$ nombreuses caractérisations distinctes

Turing-complétude

Turing-complet

Un système formel est Turing-complet s'il est aussi puissant que les machines de Turing, c-à-d qu'on peut y décrire toute fonction calculable par une machine de Turing, ou de manière équivalente, avec lequel on peut simuler une machine de Turing universelle.

Turing-équivalence

Un système formel est Turing-équivalent s'il réalise exactement les mêmes fonctions que les machines de Turing.

(On ne connaît pas de système Turing-complet et non Turing-équivalent, c'est-à-dire plus puissant, cf thèse de Church-Turing)

Turing-complétude

Sont Turing-complets :

(à la mémoire non bornée près)

- La plupart des langages de programmation
- Les processeurs généralistes

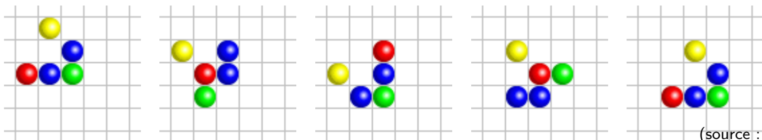
Ne sont pas Turing-complets :

- Expressions régulières / automates à état finis
- Grammaire algébrique / automates à pile
- Réseau de Petri (Turing-complet avec le test à zéro)
- Calcul quantique
- Calcul des constructions (Coq)

Turing-complétude – jeu de la vie

Le jeu de la vie de Conway (1970) : automate cellulaire bi-dimensionnel

- grille bi-dimensionnelle non bornée avec cellule vivante ou morte/vide
- une cellule vivante avec 2 ou 3 voisines vivantes survit, les autres meurent (isolation ou surpopulation)
- une cellule vide avec exactement 3 voisines vivantes devient vivante



(source : wikipedia)

Turing-complétude

Le jeu de la vie est Turing-complet

Indécidabilité du jeu de la vie

Le jeu de la vie est indécidable (p.e. déterminer si une configuration initiale conduit à rien, ou se poursuit à l'infini, ou croît indéfiniment)

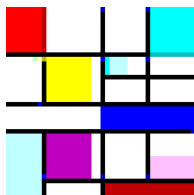
Turing-complétude

- *Rule 110* : automate cellulaire mono-dimensionnel

configuration	111	110	101	100	011	010	001	000
nouvel état central	0	1	1	0	1	1	1	0

- Des processeurs à une instruction : *(substract and branch if nonzero)*, x86 mov instruction (+ 1 jmp)
- Des langages ésotériques :

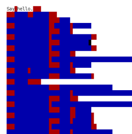
Intercal : `COME FROM`



(Piet)

```
++++++++[>++++[>++>+++>+++>+<<<<-]
>+>+>->>+[<]<-]>>.>---.+++++++..+++.
>>.<-.<.+++.------.--------.>>+.>++.
```

(Brainfuck)



(Whitespace)

(source : wikipedia)

Machines auto-reproductrices ou Quine

"yields falsehood when preceded by its quotation" yields falsehood when preceded by its quotation.

Machines auto-reproductrices

Il existe des machines de Turing qui écrivent leur propre codage.

Programme auto-reproducteur

Dans tout langage de programmation Turing-complet, on peut écrire un programme qui affiche son propre code.

```
int main(){char*a="int main(){char*a=%c%s%c;printf(a,34,a,34);}";
          printf(a,34,a,34);}
```

```
(fun s -> Printf.printf "%s %S;;" s s)
      "(fun s -> Printf.printf \"%s %S;;\" s s)";;
```

```
exec(s:='print("exec(s:=%r)"%s)')
```

Quine : preuve

- Distinguer une machine M et son codage $\langle M \rangle$
- Construire C qui calcule le codage $\langle M \cdot M' \rangle$ de la composition de deux machines données par leur codage $\langle M \rangle$ et $\langle M' \rangle$
- Pour un mot m, considérer $Print_m$ qui écrit m sur le ruban
- Pour un mot m, construire $PPrint(m)$ qui calcule le codage $\langle Print_m \rangle$
- Pour un mot $\langle M \rangle$, construire la machine $R(\langle M \rangle)$ qui calcule le codage $\langle Print_{\langle M \rangle} \cdot M \rangle$ (en utilisant $PPrint(\langle M \rangle)$ et la composition C du résultat avec $\langle M \rangle$)
- Soit la machine $Q = Print_{\langle R \rangle} \cdot R$

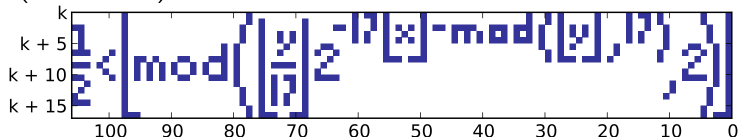
Exécution de Q = exécution de $Print_{\langle R \rangle}$ qui laisse sur la bande $\langle R \rangle$, suivie de l'exécution de R qui laisse sur la bande $\langle Print_{\langle R \rangle} \cdot R \rangle$, ce qui est le codage de Q.

Formule autoréférente de Tupper

Résoudre en (x, y) l'inégalité suivante :

$$\frac{1}{2} < \left\lfloor \mathrm{mod}\left(\left\lfloor \frac{y}{17} \right\rfloor 2^{-17\lfloor x \rfloor - \mathrm{mod}(\lfloor y \rfloor, 17)}, 2 \right) \right\rfloor$$

et l'afficher dans le plan entre $0 < x < 106$ et $k < y < k + 17$ où k est un nombre (bien choisi) de 543 chiffres :



k encode une image, choisie ici pour être le dessin de la formule.

(source : wikipedia)

Troisième partie

Indécidabilité, incalculabilité

Plan

1. Machines de Turing
   - Définitions
   - Variantes
   - Machines universelles
   - Fonctions calculables
   - Machines auto-reproductrices
2. Indécidabilité, incalculabilité
   - Indécidabilité de l'arrêt
   - Réduction
   - Autres problèmes indécidables
3. Castor affairé
4. Fonctions récursives
5. Problème de correspondance de Post
   - Définition
   - Langages & grammaires

Incalculabilité

Existence de fonctions non calculables

Il existe des fonctions non calculables.

Démonstration :

1. L'ensemble des machines de Turing est dénombrable (codage de Gödel)
2. L'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ n'est pas dénombrable (théorème de Cantor : autant que de réels)
3. cqfd...

Problème de décision

Problème de décision

Un problème de décision est la donnée d'un ensemble E d'instances et d'un sous-ensemble $P \subseteq E$ des instances positives pour lesquelles la réponse est oui.

- Nombres premiers : $E = \mathbb{N}$ et $P = \{n \in \mathbb{N} \mid n \text{ est premier}\}$
- Mots acceptés par $\mathcal{M}$: $E = X^*$ et $P = L(\mathcal{M})$
- Acceptance : $E = \{\langle \mathcal{M}, m \rangle \mid \mathcal{M} \text{ MT}, m \in X^*\}$ et $P = \{\langle \mathcal{M}, m \rangle \mid \mathcal{M} \text{ accepte } m\}$
- Vérification : $E = \{\langle Prog, F \rangle \mid Prog \text{ programme}, F \text{ formule LTL}\}$ et $P = \{\langle Prog, F \rangle \mid Prog \models F\}$

Décidabilité

Décidabilité (algorithmique)

Un problème de décision est décidable s'il existe un algorithme (= une machine de Turing) qui termine en temps fini et répond oui / non selon si l'entrée est vraie (= est une instance positive).

Semi-décidabilité

Un problème de décision est semi-décidable s'il existe un algorithme (= une machine de Turing) qui, si l'entrée est vraie, termine en temps fini et répond oui.

(si l'énoncé est faux, la machine peut aussi bien répondre non que boucler ; elle ne peut pas répondre oui)

Construction d'un prédicat indécidable

Soit $T(i, a)$ le prédicat sur $\mathbb{N} \times X^*$ qui retourne vrai si l'exécution de la machine de codage i appliquée à a retourne un résultat (ie $\mathcal{M}(a)$ s'arrête, avec $\langle\mathcal{M}\rangle = i$), et faux sinon.

Supposons T décidable. Il existe $\mathcal{M}_T$ qui décide T.

Soit la machine de Turing $\mathcal{M}$ prenant un argument a et définie par : si $T(a, a)$ est vrai alors $\mathcal{M}$ boucle, sinon $\mathcal{M}$ s'arrête.
(construction : $\mathcal{M}$ duplique son argument a – lire le premier symbole, aller à la fin, l'écrire, revenir au début, etc –, puis exécute $\mathcal{M}_T$ qui laisse 0 ou 1 sur le ruban, et boucle ou termine selon cette valeur)

$\mathcal{M}$ possède un code de Gödel j. Pour ce j, si $T(j, j)$ est vrai alors $\mathcal{M}(j)$ doit boucler, donc $T(j, j)$ doit être faux. Si $T(j, j)$ est faux alors $\mathcal{M}(j)$ doit s'arrêter donc $T(j, j)$ doit être vrai. Contradiction.

La machine $\mathcal{M}$ est impossible, donc $\mathcal{M}_T$ n'existe pas,
donc T est indécidable. $\square$

Indécidabilité de l'arrêt

Indécidabilité de l'arrêt des machines de Turing

Le prédicat $T(i, a)$ est indécidable : étant donné une machine $\mathcal{M}$ et un argument a, il est a priori impossible de savoir si $\mathcal{M}(a)$ va s'arrêter sur un état final ou boucler.

$\Rightarrow$ savoir si un programme, une boucle, une récursivité vont se terminer est indécidable : il n'existe pas de méthode ou algorithme vérifiant cela pour tout programme. Mais on peut le faire pour des cas particuliers : variants, décroissance bornée...

Semi-décidabilité de l'arrêt *(théorème inutile)*

Le problème de l'arrêt est semi-décidable.

(laisser tourner la machine !)

Indécidabilité de l'arrêt sur entrée vide

Indécidabilité de l'arrêt sur entrée vide

Étant donné une machine $\mathcal{M}$ et un ruban blanc, il est indécidable de déterminer si $\mathcal{M}$ va s'arrêter sur un état final ou boucler.

Réduction de l'arrêt : soit $\mathcal{M}'$ une machine et a un argument. Construire $\mathcal{M}$ qui écrit a sur un ruban vide puis se comporte comme $\mathcal{M}'$. Si on peut décider de l'arrêt de $\mathcal{M}$, alors on peut décider de l'arrêt de $\mathcal{M}'(a)$. Contradiction avec l'indécidabilité de l'arrêt. $\square$

Une machine qui termine... ou pas

	$\#$	1
q_0	$q_1, 1, \rightarrow$	$q_0, 1, \leftarrow$
q_1	$q_2, \#, \rightarrow$	$q_2, 1, \leftarrow$
q_2	$q_3, 1, \rightarrow$	$q_0, \#, \leftarrow$
q_3	$q_4, 1, \rightarrow$	$q_1, 1, \rightarrow$
q_4	$q_2, 1, \leftarrow$	$q_F, 1, \rightarrow$
q_F	$\varnothing$	$\varnothing$

On pense que cette machine ne termine pas mais on n'a pas la preuve (en 2021).

Démonstration de l'indécidabilité par réduction

Pour démontrer qu'un problème B est indécidable, sachant que le problème A est indécidable, on réduit A à B :

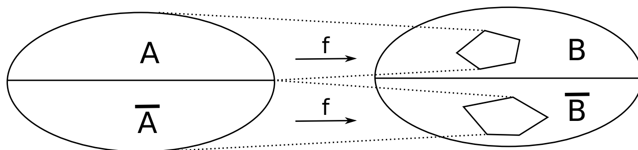
- Montrer que si on sait résoudre B, alors on peut résoudre A :
  - Supposer qu'il existe $\mathcal{M}_B$ qui décide B ;
  - En utilisant $\mathcal{M}_B$ et d'autres machines, construire une MT $\mathcal{M}_A$ qui décide A.
- A étant indécidable, $\mathcal{M}_A$ ne peut pas exister.
- Conclure que l'hypothèse d'existence de $\mathcal{M}_B$ est fausse $\rightarrow$ B est indécidable.

Réduction de problèmes

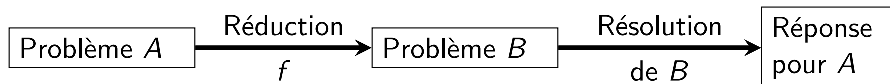
Réduction $A \leqslant B$

- Soit A et B deux problèmes
- Une réduction de A vers B est une fonction calculable f telle que $x \in A \Leftrightarrow f(x) \in B$

(B suffit à résoudre A ; on dit que A se réduit à B).



Réduction de problèmes



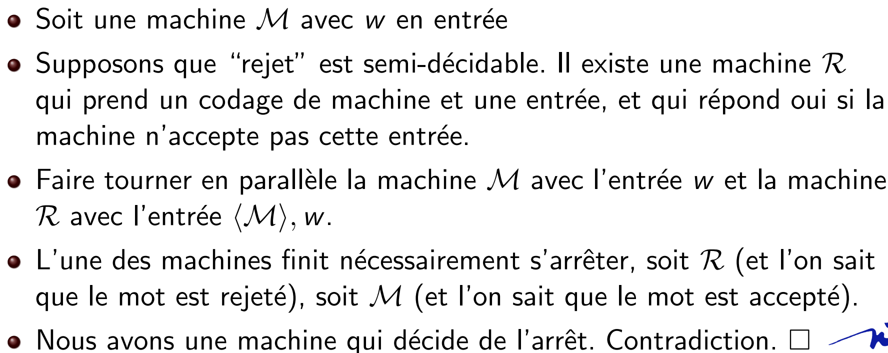
Réduction et décidabilité

- Si $A \leqslant B$ et B est décidable, A l'est aussi
- Si $A \leqslant B$ et A est indécidable, B l'est aussi

Indécidabilité du rejet

Indécidabilité du rejet

Savoir si une machine n'accepte pas un mot donné est un problème ni décidable, ni semi-décidable.



Réduction du problème de l'arrêt.

- Soit une machine $\mathcal{M}$ avec w en entrée
- Supposons que "rejet" est semi-décidable. Il existe une machine $\mathcal{R}$ qui prend un codage de machine et une entrée, et qui répond oui si la machine n'accepte pas cette entrée.
- Faire tourner en parallèle la machine $\mathcal{M}$ avec l'entrée w et la machine $\mathcal{R}$ avec l'entrée $\langle \mathcal{M} \rangle, w$.
- L'une des machines finit nécessairement s'arrêter, soit $\mathcal{R}$ (et l'on sait que le mot est rejeté), soit $\mathcal{M}$ (et l'on sait que le mot est accepté).
- Nous avons une machine qui décide de l'arrêt. Contradiction. $\square$

Machine minimale

Indécidabilité de la minimalité

Savoir si une machine de Turing est la plus petite qui résout un problème est indécidable.

- Supposons la décidabilité du prédicat qui indique la minimalité d'une machine
- Soit E une machine qui énumère les MT en ne gardant que les machines minimales (p.e. en comptant par ordre croissant du codage de Gödel)
- Soit une machine universelle $\mathcal{M}_{univ}$ quelconque
- Soit la machine C qui :
  - appelle E jusqu'à obtenir une machine D de taille $> |C|$
    (il y a en nécessairement car le nombre de MT minimales est infini – considérer les problèmes « y a-t-il n 1 sur le ruban ? »)
  - puis simule D avec $\mathcal{M}_{univ}$
  - $(|C| = |E| + |\mathcal{M}_{univ}| + |\text{test et boucle}|)$
- C est équivalente à D mais plus petite, contradiction. □

Machine à langage vide

Indécidabilité du test à zéro

Savoir si une machine n'accepte aucun mot est indécidable.

Preuve par réduction du problème de l'arrêt.

Pour une MT $\mathcal{M}$ et un mot m, construire la machine $\mathcal{M}_m$, qui pour une entrée u, teste si $m = u$, puis, si c'est vrai, elle simule $\mathcal{M}$ avec m, sinon elle rejette le mot u.

La machine $\mathcal{M}_m$ ne peut accepter au mieux que le mot m. Selon que son langage accepté est vide ou pas, on déduit que $\mathcal{M}$ accepte m ou pas. Tester si le langage est vide fournit donc une solution au problème de l'arrêt. Contradiction. $\square$

Machines équivalentes

Indécidabilité de l'équivalence

Savoir si deux machines de Turing sont équivalentes (i.e. acceptent le même langage et calculent la même fonction) est indécidable.

Preuve par réduction du test à zéro.

Construire $\mathcal{M}_\emptyset$ qui n'accepte aucun mot (machine sans état final).
Alors tester si $\mathcal{M}$ est équivalente à $\mathcal{M}_\emptyset$ répond au test à zéro. $\square$

Autres problèmes indécidables

Équations diophantiennes (dixième problème de Hilbert)

Soit $p(x_1, \ldots, x_n)$ un polynôme à coefficients entiers. Déterminer si l'équation $p(x_1, \ldots, x_n) = 0$ possède des solutions entières est un problème indécidable. (théorème de Matiyasevich, 1970)

Arithmétique

La validité d'une formule arithmétique (avec $+$ et $*$) est indécidable.

(l'arithmétique de Presburger (arithmétique entière sans multiplication ou avec multiplication par des constantes) est décidable)

Algèbre linéaire

Étant donné un nombre fini de matrices 3×3 à coefficients entiers, déterminer si un produit multiple permet d'annuler la composante (i, j) est indécidable.

Impossibilité de l'intelligence artificielle

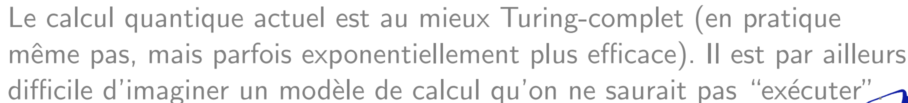
Impossibilité de l'intelligence artificielle

Il est impossible de concevoir, à base d'ordinateurs actuels, une *intelligence artificielle* qui puisse faire plus que ce qu'on sait déjà faire.

L'intelligence artificielle (en 2025) est la composition d'algorithmes (= de fonctions calculables) : elle construit une fonction calculable, donc l'IA est au mieux Turing-complet. □

Limitation de la simulation

Si un modèle de calcul, un processeur, un langage est simulable sur un ordinateur actuel, alors il est au mieux Turing-complet.



Le calcul quantique actuel est au mieux Turing-complet (en pratique même pas, mais parfois exponentiellement plus efficace). Il est par ailleurs difficile d'imaginer un modèle de calcul qu'on ne saurait pas "exécuter".

Pas de panique !

Instances finies

Tout problème ayant un nombre d'instances finies est décidable.

(il suffit d'énumérer les instances pour les vérifier une à une)

Instances particulières

Le fait qu'un problème soit indécidable / qu'une fonction soit incalculable ne signifie pas que des instances particulières ne sont pas décidables / calculables.

Approximation

En absence d'algorithme, on peut approximer un problème (ou des instances particulières d'un problème) pour se ramener à un problème décidable.

Existence de problèmes décidables

Il existe des problèmes décidables non triviaux :

- Arithmétique de Presburger (arithmétique entière sans multiplication ou avec multiplication par des constantes)
- Égalité de deux formules LTL
  (égalité = même ensemble de modèles)
- SAT (satisfiabilité de formule en logique des propositions)
- Accessibilité d'un marquage dans les réseaux de Petri
- Typage dans Ocaml (sans les modules ; semi-décidable avec)
- Égalité des fonctions calculables dans $\{0,1\}^\omega \to \mathbb{N}$
  (alors que l'égalité des fonctions dans $\mathbb{N} \to \mathbb{N}$ est indécidable)
- Les fonctions *décroissantes* de $\mathbb{N} \to \mathbb{N}$ sont toutes calculables.
  (elles sont dénombrables)

Quatrième partie

Castor affairé (*Busy beaver*)

Plan

1. Machines de Turing
   - Définitions
   - Variantes
   - Machines universelles
   - Fonctions calculables
   - Machines auto-reproductrices
2. Indécidabilité, incalculabilité
   - Indécidabilité de l'arrêt
   - Réduction
   - Autres problèmes indécidables
3. Castor affairé
4. Fonctions récursives
5. Problème de correspondance de Post
   - Définition
   - Langages & grammaires

Castor affairé

Busy Beaver

Le jeu du castor affairé à n états consiste à concevoir une machine de Turing avec n états + un état final, un alphabet $\Gamma = \{\#, 1\}$, et qui partant d'un ruban intégralement blanc écrit le plus de 1 possible **avant de s'arrêter** (variante équivalente : fait le plus de transitions possible).

Score $\Sigma(n) \triangleq$ nombre de 1 sur la bande à l'arrêt
Score $S(n) \triangleq$ nombre de transitions effectuées
Nombre de machines de Turing à étudier : $(4(n+1))^{2n}$

- n=1, $\Sigma = 1$, $S = 1$: $q_0 \xrightarrow{\#,1,\rightarrow} q_F$
- n=2, $\Sigma = 4$, $S = 6$: q_0 —(#,1,→ ; 1,1,←)→ q_1 —(1,1,→)→ q_F ; q_1 —(#,1,←)→ q_0

$\#q_0\# \vdash \#1q_1\# \vdash \#q_0 11 \vdash \#q_1\#11 \vdash \#q_0\#111 \vdash \#1q_1 111 \vdash 11q_F 11$

Busy beaver : résultats connus

n	$\Sigma(n)$	$S(n)$	$nb\ MT$	
1	1	1	64	
2	4	6	20736	
3	6	21	6777216	
4	13	107	$256 * 10^8$	(1983)
5	4098	47176870	$\approx 6.3 * 10^{14}$	(2024)
6	$\geq 3.5 * 10^{18267}$	$\geq 7.4 * 10^{36534}$	$\approx 2.3 * 10^{18}$	(2010)
7	$\geq 10^{10^{10^{10^{18705353}}}}$	...	$\approx 1.2 * 10^{22}$	(2014)

Busy beaver 5 : suspecté depuis 1990, prouvé en 2024 (avec Coq)

Trouvé en 2022 : $S(6) \geq 10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10}}}}}}}}}}}}}}$

(source : wikipedia)

Incalculabilité de $\Sigma(n)$

f croît plus vite que g, noté $f(x) \gg g(x)$, $\triangleq \exists x_0 : \forall x > x_0 : f(x) > g(x)$.
Σ croît plus vite que toute fonction calculable :
$\forall f$ calculable : $\Sigma(n) \gg f(n)$.

Preuve informelle : soit une fonction calculable quelconque. On peut construire une fonction calculable qui croît plus vite (p.e. $2 * f(x)$) et une machine de Turing qui écrit autant de 1 que cette fonction. Le score du busy beaver est au moins aussi grand que celui de cette MT.

Corollaire : Σ n'est pas calculable.

S n'est pas calculable.

Le nombre de transitions $S(n)$ est supérieur au nombre de 1 $\Sigma(n)$.

Incalculabilité de $\Sigma(n)$

1. Soit f une fonction calculable. Considérons $F(x) = \Sigma_{i=0}^{x}(f(i) + i^2)$.
2. F est calculable. Soit $\mathcal{M}_F$ la MT à C états qui calcule F.
3. Construisons $\mathcal{M}^{(x)}$ la MT à x états qui écrit x 1 consécutifs sur un ruban vierge, et $\mathcal{M}_F^{(x)} \triangleq \mathcal{M}^{(x)} \to \mathcal{M}_F \to \mathcal{M}_F$.
4. $\mathcal{M}_F^{(x)}$ a $x + 2C$ états. Elle écrit x 1, puis $F(x)$ 1 puis $F(F(x))$ 1.
5. Score$(\mathcal{M}_F^{(x)}) = F(F(x))$ donc $\Sigma(x + 2C) \geq F(F(x))$.
6. Or $F(x) \geq x^2$ et $x^2 \gg (x + 2C)$ donc $F(x) \gg (x + 2C)$.
7. F est monotone par construction donc $F(F(x)) \gg F(x + 2C)$.
8. Donc $\Sigma(x + 2C) \gg F(x + 2C)$. Comme $F(y) \geq f(y)$, on obtient $\Sigma(x + 2C) \gg f(x + 2C)$.
9. Donc $\Sigma(n) \gg f(n)$ avec f quelconque. $\square$

Incalculabilité de $S(n)$

Preuve par réduction du problème de l'arrêt

Supposons que $S(n)$ est calculable. Il existe alors une machine de Turing $\mathcal{A}$ qui calcule S et nous allons construire une machine qui résout le problème de l'arrêt.

Construisons une machine $\mathcal{M}$ qui prend en entrée le codage d'une machine $\mathcal{T}$ quelconque. La machine $\mathcal{M}$ détermine le nombre d'états de $\mathcal{T}$ (compter les « , » et « ; ») puis utilise $\mathcal{A}$ pour calculer $S(n)$.

Ensuite $\mathcal{M}$ simule $\mathcal{T}$ en comptant les transitions de $\mathcal{T}$.
Si la simulation de $\mathcal{T}$ s'arrête en moins de $S(n)$ transitions $\Rightarrow$ $\mathcal{M}$ décide que $\mathcal{T}$ s'arrête.
Si la simulation prend plus de $S(n)$ transitions $\Rightarrow$ $\mathcal{M}$ décide que $\mathcal{T}$ boucle.

$\mathcal{M}$ résout le problème de décision de l'arrêt d'une machine quelconque.
Contradiction avec l'indécidabilité de l'arrêt sur entrée vide. $\square$

Incalculabilité d'une borne sup à $S(n)$

Il n'existe pas de fonction calculable qui donne une borne supérieure à $S(n)$ (un majorant).

Preuve par réduction : comme précédemment, contradiction avec l'indécidabilité de l'arrêt.

Preuve constructive : énumérer les machines de Turing à n états (en nombre fini), simuler chacune au plus jusqu'à la borne supérieure et garder celle qui fait le plus de transitions (l'une de celles s'il y en a plusieurs). C'est un *busy beaver* à n états. L'exécuter pour calculer $S(n)$. Contradiction avec l'incalculabilité de $S(n)$. $\square$

$S(n)$ croît plus vite que n'importe quelle fonction mathématique.

(comme $S(n) \geq \Sigma(n)$, on le savait déjà !)

Démonstrateur automatique universel

Considérons la conjecture de Golbach : tout nombre pair est la somme de deux nombres premiers.
Soit le programme :

```
n = 4
while true:
    search p in 1..n so that p and n-p are prime
    if found:
        n = n+2
    else:
        halt
```

Ce programme est de taille t. Si la conjecture est fausse, il s'arrête en moins de $S(t)$ pas ; si la conjecture est vraie, il boucle à l'infini et il suffit de faire $S(t)+1$ transitions pour le savoir $\Rightarrow \exists$ une procédure de décision !
(Mais on ne peut pas calculer $S(n)$...)

Cinquième partie

Fonctions récursives

Plan

1. Machines de Turing
   - Définitions
   - Variantes
   - Machines universelles
   - Fonctions calculables
   - Machines auto-reproductrices
2. Indécidabilité, incalculabilité
   - Indécidabilité de l'arrêt
   - Réduction
   - Autres problèmes indécidables
3. Castor affairé
4. Fonctions récursives
5. Problème de correspondance de Post
   - Définition
   - Langages & grammaires

Fonctions récursives

Fonctions récursives primitives

La plus petite classe de fonctions construites par projection, composition, iteration (récursion). Ces fonctions terminent nécessairement.

$\Rightarrow$ il existe des fonctions non récursives primitives qui terminent nécessairement, et il existe des fonctions non récursives primitives qui peuvent ne pas terminer

Fonctions récursives

La plus petite classe de fonctions construites par projection, composition, itération (récursion) et minimisation.

$\Rightarrow$ équivalent aux fonctions calculables par les machines de Turing

Fonctions récursives primitives

Soit la classe des fonctions de $\mathbb{N}^k$ vers $\mathbb{N}^r$ construites à partir de :

- Identité $id : \mathbb{N}^k \to \mathbb{N}^k$ telle que $id(x_1, \ldots, x_n) = (x_1, \ldots, x_n)$
- Zéro $Z : \mathbb{N}^0 \to \mathbb{N}$ telle que $Z() = 0$
- Successeur $S : \mathbb{N} \to \mathbb{N}$ telle que $S(n) = n + 1$
- Projection $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$ telle que $\pi_i^k(x_1, \ldots, x_k) = x_i$
- Composition *Comp* telle que $Comp(f, g_1, \ldots, g_n) = h$ où $h(x_1, \ldots, x_n) = f(g(x_1), \ldots, g(x_n))$
- Récursion *Rec* telle que $Rec(f, g) = u$ où
  $\begin{cases} u(m, 0) = f(m) \\ u(m, n+1) = g(n, u(m, n), m) \end{cases}$
  (La récursion termine nécessairement par décroissance à 0)

Exemples de fonctions récursives primitives

- $Somme = Rec(\pi_1^1, Comp(S, \pi_2^3))$
  $$\begin{cases} Somme(n, 0) & = & \pi_1^1(n) \\ Somme(n, m+1) & = & S(\pi_2^3(n, Somme(n, m), m)) \end{cases}$$
- $Mult = Rec(Z, Comp(Somme, \pi_2^3, \pi_3^3))$
  $$\begin{cases} Mult(m, 0) & = & 0 \\ Mult(m, n+1) & = & Somme(Mult(m, n), m) \end{cases}$$
- $Eq0 = Rec(1, 0)$
- ...

Calculabilité des fonctions récursives primitives

Calculabilité

Les fonctions récursives primitives sont calculables.

- Les fonctions de base sont trivialement calculables
- La composition est calculable : séquence $\langle$calculer chacun des arguments ; calculer $f\rangle$
- La récursivité est calculable : $u(m, n)$ est équivalent à la boucle

  $r \leftarrow f(m)$
  for $i = 1$ **to** n **do**
  $\quad r \leftarrow g(i, r, m)$
  done
  return r

Existence de fonctions non récursives primitives

Existence de fonctions non récursives primitives

- L'ensemble des fonctions récursives primitives est dénombrable (énumérer leur texte)
- L'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ n'est pas dénombrable (théorème de Cantor : autant que de réels)
- cqfd...

Une fonction calculable non récursive primitive

Existence de fonctions calculables non récursives primitives

Il existe des fonctions calculables non récursives primitives.

Diagonalisation de Cantor : énumérer les fonctions récursives primitives (par ordre lexicographique de leurs composants de base) et considérer la fonction ayant une valeur différente de la diagonale.

	0	1	2	$\cdots$
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$\cdots$
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$\cdots$
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$\cdots$
f_3	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Considérer $g(n) = f_n(n) + 1$. g est calculable par construction.

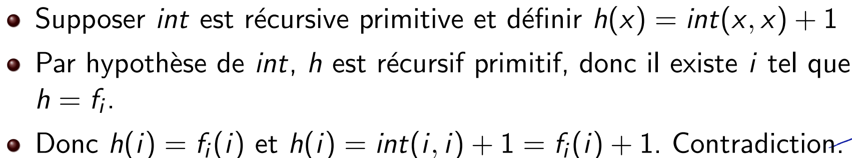
Si g est récursive primitive, elle a un numéro m, i.e. $g = f_m$, alors $g(m) = f_m(m)$ et $g(m) = f_m(m) + 1$. Contradiction. $\square$

Interpréteur universel

La fonction qui évalue n'importe quel terme récursif primitif n'est pas récursive primitive (mais elle est calculable).

Preuve par diagonalisation :

- Considérer un codage des fonctions récursives primitives, p.e. codage en ascii de la chaîne la définissant
- Définir l'interpréteur universel *int* :
  $int(i, x) =$
  $\begin{cases} g(x) \text{ si } i \text{ est le code d'une fct récursive primitive } g = f_i \\ 0 \text{ sinon} \end{cases}$



- Supposer *int* est récursive primitive et définir $h(x) = int(x, x) + 1$
- Par hypothèse de *int*, h est récursif primitif, donc il existe i tel que $h = f_i$.
- Donc $h(i) = f_i(i)$ et $h(i) = int(i, i) + 1 = f_i(i) + 1$. Contradiction. □

Ackermann : une fonction récursive, non récursive primitive

Fonction d'Ackermann (simplifiée) :

$$\begin{aligned} A(0, n) &= n + 1 \\ A(k+1, 0) &= A(k, 1) \\ A(k+1, n+1) &= A(k, A(k+1, n)) \end{aligned}$$

La fonction d'Ackermann croît plus vite que toute fonction récursive primitive.

(preuve compliquée)

La fonction d'Ackermann n'est pas récursive primitive
(mais elle est calculable et termine nécessairement).

Fonctions récursives

Fonctions définies à partir de :

- Primitif récursif
- Minimisation non bornée : pour une fonction $f(n,i)$, la fonction $\mu i\ f$ est telle que : $\mu i\ f = \begin{cases} \text{le plus petit } i \text{ tel que } f(n,i) = 1 \\ \text{non définie sinon} \end{cases}$

(c'est un peu plus subtil : f peut être une fonction partielle)

μi est calculable : $(\mu i\ f)(n) =$

$i \leftarrow 0$
while $f(n,i) \neq 1$ **do** $i \leftarrow i+1$; **done**
return i

Expressivité

Les fonctions récursives sont équivalentes aux machines de Turing.

(le gain par rapport à récursif primitif est la boucle non bornée)

Fonctions récursives $\rightarrow$ MT

On a vu que :

1. Les fonctions de bases sont calculables par une MT
2. La composition est calculable par une MT
3. La récursivité est calculable par une MT
4. La minimisation non bornée est calculable par une MT

MT $\rightarrow$ fonctions récursives

Soit une MT calculant une fonction f de $\mathbb{N}$ dans $\mathbb{N}$
(si autre alphabet, utiliser un codage)
Soit les fonctions :

- $init(x)$ qui donne la configuration initiale pour l'entrée x
- $next(c)$ qui donne la configuration qui suit c
- $config(c, n) = Rec(id, Comp(config, \pi_2^3))$ qui donne la n-ième configuration : $\begin{cases} config(c, 0) & = c \\ config(c, n+1) & = next(config(c, n)) \end{cases}$
- $stop(c)$ qui vaut 1 si c est finale, 0 sinon
- $steps(x) = \mu i \ stop(config(init(x), i))$ est le nombre de pas pour que la MT s'arrête sur l'entrée x.
- $out(c)$ qui donne la valeur calculée dans la configuration c

Alors $f(x) = out(config(init(x), steps(x)))$
□

Indécidabilité de la récursion primitive

Indécidabilité de la récursion primitive

Savoir si une fonction récursive est récursive primitive est un problème indécidable.

Il n'existe pas d'algorithme qui détermine si une fonction récursive quelconque peut être définie comme fonction récursive primitive.
Démonstration similaire à l'indécidabilité de l'arrêt des machines de Turing (en plus acrobatique) car, par réduction, cela revient à reconnaître que la fonction ne peut jamais boucler.

Conclusion

- Les fonctions récursives primitives sont
  - un sous-ensemble des fonctions totales (qui terminent pour toute entrée)
  - assez expressives pour de nombreuses fonctions usuelles : addition, multiplication, factorielle, minimum/maximum, conditionnelle, quantification bornée...
- Il existe d'autres calculs, expressifs et nécessairement terminants (calcul des constructions de Coq, système de réécriture avec contraintes sur les règles...)
- Ils deviennent vite équivalents aux machines de Turing (et donc indécidables) quand on veut lever leurs contraintes

Sixième partie

Problème de correspondance de Post

Plan

1. Machines de Turing
   - Définitions
   - Variantes
   - Machines universelles
   - Fonctions calculables
   - Machines auto-reproductrices
2. Indécidabilité, incalculabilité
   - Indécidabilité de l'arrêt
   - Réduction
   - Autres problèmes indécidables
3. Castor affairé
4. Fonctions récursives
5. Problème de correspondance de Post
   - Définition
   - Langages & grammaires

Problème de correspondance de Post

Problème de correspondance de Post (PCP)

Soit un alphabet Σ et deux suites de n mots $u_1, \ldots, u_n$ et $v_1, \ldots, v_n$.
Existe-t-il une suite finie $i_1, \ldots, i_k$ telle que $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$?

(noter : les suites d'indices sont les mêmes des deux côtés)

Visualisation : voir chaque paire de mots (u_i, v_i) comme un domino (en autant d'exemplaires que nécessaire). Peut-on concaténer des dominos tel que les deux mots soient identiques ?

Exemple : $u = \langle ab, c, ba, abc, ab \rangle$ et $v = \langle a, bcab, bb, bc, ba \rangle$.
Solution : $\langle 1, 2, 1, 4 \rangle$.

$$\boxed{\begin{matrix} ab \\ a \end{matrix}} \boxed{\begin{matrix} c \\ bcab \end{matrix}} \boxed{\begin{matrix} ab \\ a \end{matrix}} \boxed{\begin{matrix} abc \\ bc \end{matrix}} = \boxed{\begin{matrix} abcababc \\ abcababc \end{matrix}}$$

Indécidabilité

Indécidabilité

PCP est indécidable si l'alphabet Σ est de taille ≥ 2.

Preuve par réduction du problème de l'arrêt : on simule un calcul d'une MT par un PCP, la résolution du PCP est un calcul acceptant de la MT.

Décidabilité

- PCP est décidable si l'alphabet Σ est de taille $= 1$.
- PCP est décidable si le nombre de mots $n \leq 2$, indécidable si $n \geq 5$ (prouvé en 2015), inconnu pour $3 \leq n \leq 4$.
- PCP est décidable (en temps exponentiel) si chaque mot u_i commence par une lettre différente, ainsi que chaque mot v_i.
- PCP borné où l'on cherche une solution de moins de k mots est décidable (en temps exponentiel).

Langages rationnels

Décidabilité des langages rationnels

Globalement, tous les problèmes concernant les langages rationnels et les automates à états finis sont décidables :

- $m \in L(A)$
- $L(A) = \emptyset$
- $L(A) = L(B)$
- $L(A) \subseteq L(B)$
- ...

Preuve : unicité et finitude de l'automate déterministe minimal.

Grammaires algébriques

Décidabilité des grammaires algébriques

Soit une grammaire algébrique G, les problèmes suivants sont décidables :

- $m \in L(G)$
- $L(G) = \emptyset$

Preuve : analyseur d'Earley ou LR généralisé

Indécidabilité des grammaires algébriques

Soit deux grammaires algébriques G et G' sur un alphabet Σ, les problèmes suivants sont indécidables :

1. $L(G) \cap L(G') = \emptyset$
2. $L(G) = \Sigma^*$
3. $L(G) = L(G')$
4. $L(G) \subseteq L(G')$

Preuve par réduction de PCP.

Grammaires algébriques

Soit un PCP sur Σ avec n mots $u_1, \ldots, u_n$ et $v_1, \ldots, v_n$.

Ajouter un alphabet $A = \{a_1, \ldots, a_n\}$ formé de lettres $\notin \Sigma$.

Soit la grammaire algébrique G_u avec les productions $S \to \sum_{i=1}^{m} a_i S u_i + \Lambda$.

Alors $L(G_u) = \{a_{i_1} \cdots a_{i_m} u_{i_m} \cdots u_{i_1} \mid m \geq 0 \wedge 1 \leq i_k \leq n\}$.

1. Réduction du PCP à l'égalité des langages : G_u et G_v sont constructibles par une MT à partir du PCP.
   Le PCP à une solution ssi $L(G_u) \cap L(G_v) \neq \emptyset$.
2. Réduction du problème de l'arrêt.
3. Réduction du 2 au 3 en prenant $L(G') = \Sigma^*$
4. Réduction du 3 au 4 par double inclusion

Septième partie

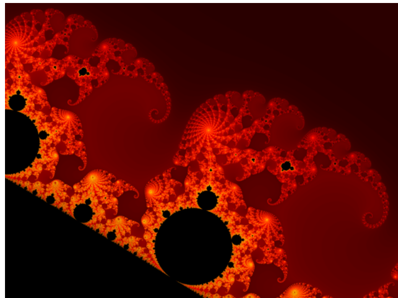
Complexité de Kolmogorov

Complexité / simplicité

Quelle séquence est la plus simple ?

- 1212121212121212121212121212
- 1415926535897932384626433832
- 7452698612751472365892164565

Quelle image est la plus simple ?



(source : wikipedia)

Complexité algorithmique de Kolmogorov

Définition (à base de machine de Turing)

La complexité de Kolmogorov d'une séquence de bits s est la taille de la plus petite machine de Turing qui produit s :

$$K(s) = \min_{\mathcal{M} \in MT} \{|\langle \mathcal{M} \rangle| \text{ tel que } \mathcal{M}() = s\}$$

(où $\langle \mathcal{M} \rangle$ est le codage de Gödel de $\mathcal{M}$ et $|.|$ sa taille en bit)

Redéfinition (à base de processeur)

Soit une machine universelle $\mathcal{U}$, la complexité de Kolmogorov pour $\mathcal{U}$ d'une séquence s est la taille du plus petit programme exécutable par $\mathcal{U}$ qui produit s :

$$K_{\mathcal{U}}(s) = \min_{p \in P_{\mathcal{U}}} \{|p| \text{ tel que } \mathcal{U}(p) = s\}$$

(où $P_{\mathcal{U}}$ est l'ensemble des programmes exécutables par $\mathcal{U}$ et $\mathcal{U}(p)$ l'exécution de p sur $\mathcal{U}$)

Exemples

- La complexité de $12121212\ldots$ est très faible : **print**$(''12'' * 14)$
- La complexité de $14159265\ldots$ est faible : il existe des petits programmes (200 caractères en Haskell) qui calculent les décimales de π. La complexité des 20 premières décimales ou des 20000 premières est quasiment la même.
- La complexité $745269861\ldots$ est de l'ordre de sa longueur (`print("...")`), la chaîne est aléatoire (a priori).
- La complexité de l'image de droite est bien inférieure à sa taille : ensemble de Mandelbrot.

Théorème d'invariance

Borne supérieure

$\exists c \in \mathbb{N}, \forall s \in \{0,1\}^*, K_{\mathcal{U}}(s) \leq |s| + c$

Considérer le programme "print(s)".

Indépendance du langage

Soient $\mathcal{U}$ et $\mathcal{V}$ deux machines universelles, alors
$\exists c, \forall s \in \{0,1\}^*, |K_{\mathcal{U}}(s) - K_{\mathcal{V}}(s)| \leq c$.

$\rightarrow$ la complexité de Kolmogorov ne dépend pas du langage de programmation (à une constante près).
Preuve : considérer un interpréteur $I_{\mathcal{U}}^{\mathcal{V}}$ qui soit un interpréteur de $\mathcal{V}$ écrit pour $\mathcal{U}$. Soit $p_{\mathcal{V}}$ le plus petit programme pour $\mathcal{V}$ qui engendre s. Alors $I_{\mathcal{U}}^{\mathcal{V}}(p_{\mathcal{V}})$ est un programme pour $\mathcal{U}$ qui engendre s et de taille $|p_{\mathcal{V}}| + |I_{\mathcal{U}}^{\mathcal{V}}|$. $\square$

Incalculabilité de K

La complexité de Kolmogorov n'est pas bornée

$\forall n \in \mathbb{N}, \exists s \in \{0,1\}^*, K(s) > n$

Preuve : il n'y a que $\Sigma_{i=1}^{n} 2^i = 2^{n+1} - 1$ programmes de taille $\leq n$. Ils ne peuvent engendrer qu'au plus $2^{n+1} - 1$ séquences distinctes, alors qu'il y en a une infinité. □

Incalculabilité de K

K n'est pas calculable.

Supposons K calculable, et soit k la taille du code qui la calcule. Soit le programme qui cherche la plus petite séquence de complexité $\geq k + 51$:

```
n = 1
while K(n) < k + 51:
    n = n + 1
print(n)
```

Ce programme de complexité $k + 50$ affiche une séquence de complexité $\geq k + 51$. Absurde. □

Incompressibilité

Incompressibilité

Une séquence s est incompressible si $K(s) \geq |s|$.

(il en existe, même argument que l'absence de borne)

En fait, la proportion de séquences de taille n et de complexité $n - k$ est inférieure à $\frac{1}{2^k}$: la majorité des séquences sont pas ou peu compressibles.

La complexité algorithmique de Kolmogorov est une bonne mesure de l'aléatoire, bien meilleure que l'entropie de Shannon ou toute mesure statistique (qui attribue la même complexité à 0 1 10 11 100 101 110 . . . (séquence de Champernowne) qu'aux décimales de π ou qu'une séquence obtenue par tirage aléatoire équiprobable).

Approximer Kolmogorov

Utiliser des outils de compression auto-extractifs (zip etc), garder le minimum ou à défaut la longueur de la séquence.

Huitième partie

Conclusion

Théorème de Rice

Théorème de Rice

Toute propriété sémantique non triviale d'un programme est indécidable.

Réduction du problème de l'arrêt.

- Soit P propriété non triviale ($\exists M$ qui vérifie P et $\exists M'$ qui vérifie $\neg P$). On suppose que $\emptyset \notin P$ quitte à échanger P et $\neg P$ ($\emptyset =$ machine qui n'accepte aucun mot).
- Soit $\mathcal{M}_0$ qui vérifie P : $L(\mathcal{M}_0) \in P$.
- Pour toute paire $(\mathcal{M}, m)$, construire la machine $\mathcal{M}_m$ d'entrée u : **si** $\mathcal{M}$ accepte m **alors** simuler $\mathcal{M}_0$ avec u **sinon** rejeter
- Si $\mathcal{M}$ accepte m, $L(\mathcal{M}_m) = L(\mathcal{M}_0)$; sinon $L(\mathcal{M}_m) = \emptyset$
- Donc $m \in L(\mathcal{M}) \Leftrightarrow L(\mathcal{M}_m) \in P$
- Tester si $\mathcal{M}_m$ vérifie P répond à si $\mathcal{M}$ accepte m = problème de l'arrêt. Contradiction. $\square$

Il n'y a pas de méthode universelle pour décider si toute boucle s'arrête, si une fonction quelconque est croissante, si une variable est bornée, etc

Au-delà de Church-Turing

Peut-on aller au-delà des machines de Turing ?

Oracle

Oracle : supposer qu'un problème indécidable possède un oracle qui répond correctement et instantanément.

- Il existe une hiérarchie dans l'indécidabilité : si P_1 est réductible à P_2 ($P_1 \leq P_2$) mais pas l'inverse, un oracle pour P_1 ne suffit pas à décider P_2.
- Quelque soit la puissance de l'oracle, il existe des problèmes indécidables avec.

Saut dans l'infini

Nombre d'états infini ?

Alphabet infini ? Alphabet non dénombrable (les réels) ?

(le codage de Gödel n'est plus faisable)

Conclusion

Bilan

- Définition de la notion de calcul.
  La notion de fonction calculable ne dépend pas du modèle de calcul (s'il est assez expressif)
- Limites : décidabilité, calculabilité.
  Toute question sérieuse est indécidable.
- Raisonnement par réduction.

Le fait qu'un problème soit indécidable / qu'une fonction soit non calculable ne signifie pas que des instances particulières ne sont pas décidables / calculables.