Cours de Tronc Commun Scientifique

Recherche Opérationnelle

Introduction à la calculabilité

Frédéric Sur
École des Mines de Nancy

www.loria.fr/∼sur/enseignement/RO/

Introduction à la calculabilité

Objectifs

Témoignage : "En stage, j'ai à programmer l'optimisation d'une tournée de ramassage d'ordures ménagères. Après quatre mois de travail je n'ai (presque) plus de bugs, mais bon, ça rame. . ."

But ici : éléments de réponse à la question :

Quelles sont les limites de l'informatique ?
(des algorithmes ?)

- Réponse indépendante de la technologie ;
- Nécessite de formaliser ce qu'est un *problème*, et un *programme informatique*.

2012 centenaire de la naissance d'Alan Turing.

Qu'est-ce qu'un problème ?

Exemples :

- trier un tableau de nombres,
- résoudre un problème de programmation linéaire,
- tester la primalité d'un entier naturel,
- chercher un circuit hamiltonien dans un graphe. . .

Instance d'un problème :

- trier le tableau $\{3, 5, 1, 2, 2, 7\}$,
- est-ce que 2910431 est premier ?

Problèmes binaires (oui / non) *versus* problèmes non-binaires
$\longrightarrow$ On se restreint aujourd'hui aux problèmes binaires, on verra que cela suffit.

Qu'est-ce qu'un programme ?

⟶ Tout ce qui est programmable sur un ordinateur...

Exemple : un programme de tri admettant en entrée un tableau, écrit en Java tournant sur un Mac.

Introduction à la calculabilité

Les automates

⟶ on cherche à modéliser la notion de programme informatique, pour pouvoir établir des théorèmes sur ce qu'on peut (ou pas) faire.

Les *automates* sont **un** moyen de modéliser des programmes.

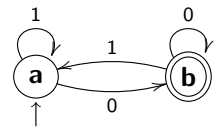
Automate = ensemble fini d'états + transitions étiquetées par les lettres d'un alphabet.
États initiaux et finals (*acceptants*).

Chaque état représente un état de la machine sur lequel est exécuté le programme, au fur et à mesure de l'exécution.

On lit une entrée (un mot) codée sur un alphabet de gauche à droite, l'automate se déplace d'états en états selon les transitions.

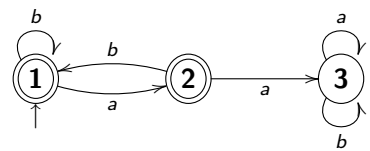
Exemples d'automate déterministe

Exemple 1 :



reconnaît les mots sur $\{0,1\}$ se terminant par 0
i.e. entiers (en binaire) divisibles par 2.

Exemple 2 :

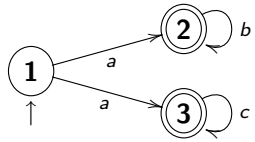


reconnaît les mots sur $\{a,b\}$ ne contenant pas deux a consécutifs.

Automates non-déterministes

Automate non-déterministe : plusieurs transitions peuvent correspondre à la même lettres

Exemple :



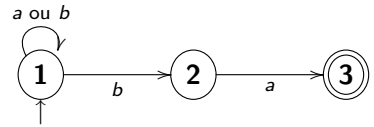
reconnaît les mots de type ab^n **ou** ac^n.

⟶ ne correspond pas à un programme réel, car il faut *choisir* une exécution.

Intérêt : Plus simple pour coder une alternative, et donc coder certains langages.

Exemple

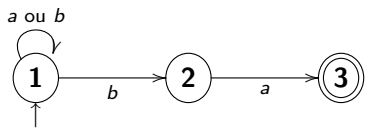
Automate non-déterministe :



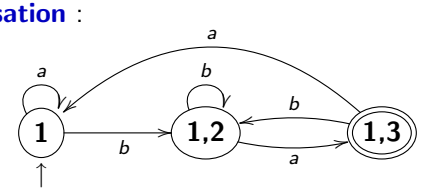
reconnaît les mots qui finissent par *ba*.

abab reconnu ?
abba reconnu ?
⟶ mots reconnus par *une* des exécutions.

Déterminisation d'un automate non-déterministe



Déterminisation :



⟶ construction d'un automate qui peut avoir 2^N états.

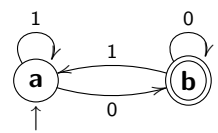
Théorème : l'ensemble des mots reconnus par un automate non-déterministe peut être reconnu par un automate déterministe (potentiellement *beaucoup plus gros*).

Langages et problèmes

Définition : l'ensemble des mots (sur un alphabet fixé) reconnus par un automate est son *langage*.

⟶ un automate *résout un problème* s'il accepte un langage dont chaque mot encode les instances positives du problème.

Exemple : automate résolvant le problème "n est-il divisible par 2 ?", avec encodage binaire (sur l'alphabet $\{0,1\}$).



Limites des automates

Théorème (Kleene) : les langages reconnus par les automates sont les *langages réguliers*.

Mais : tous les langages ne sont pas réguliers.
. . .et en particulier, les automates ne modélisent pas *tous* les programmes possibles.

Exemple : il n'existe pas d'automate reconnaissant :

$$\{a^n b^n,\ n \in \mathbb{Z}\}$$

Preuve : raisonnement par l'absurde (*pumping lemma* ou *lemme de gonflement*).

Conclusion : les automates ne forment pas un modèle assez "général" (pour notre problème).

Remarque : Les automates (langages réguliers) sont effectivement utilisés e.g. pour le *parsing* dans un compilateur.

Les machines de Turing (1936 !)

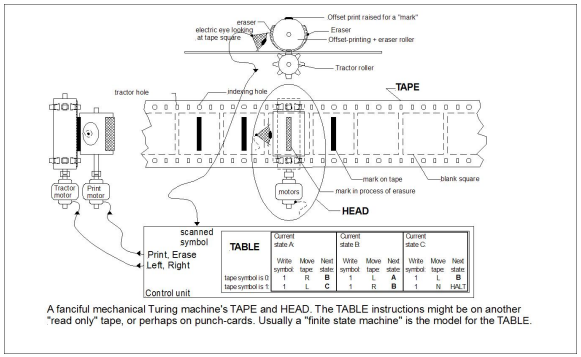


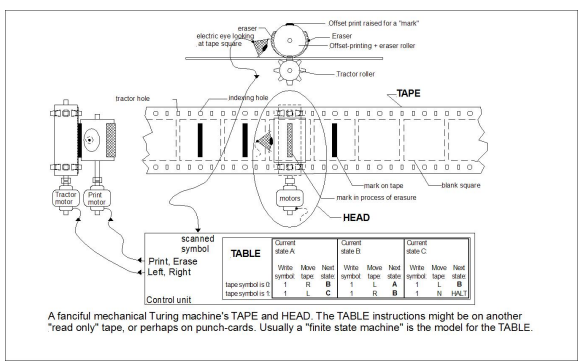
Illustration : wikipedia.org

"Définition" : à chaque étape, la machine lit le symbole sur le ruban se trouvant sous la tête de lecture, remplace ce symbole selon la table de transition de son état courant, se déplace à gauche ou à droite selon la table, change d'état.

Un mot (en entrée) est accepté si la machine arrive dans un état accepteur.

Illustration

Exemple de transition : Si machine dans état 4 et lecture de c, alors remplacer c par a, passer dans état 18, et déplacer ruban vers la droite.



(notion de *résultat du calcul* : ce qui reste sur le ruban)

Voir aussi "The LEGO Turing Machine" :
`http://www.youtube.com/watch?v=cYw2ewo06c4`

Exemple

Problème des circuits hamiltoniens dans les graphes.

Étape 1 : trouver un encodage (=une description) raisonnable de tout graphe à l'aide de l'alphabet.
Par exemple : *matrice de transition* codée sur le ruban.

Étape 2 : "Construire" une machine de Turing, qui étant donné un graphe encodé sur la bande, termine dans un état accepteur ssi le graphe a un circuit hamiltonien.

La thèse de (Turing-) Church ($\sim$ 1930-1940)

Les encodage des entrées d'un problème binaire pour lesquels il existe un programme informatique répondant correctement oui ou non sont ceux *décidés* par une machine de Turing.

$\longrightarrow$ Les instances positives sont reconnus par la machine : terminaison sur un état accepteur (langage de la machine).
$\longrightarrow$ Celle-ci termine toujours (et donc on est capable de "rejeter" les mots non acceptés)

Remarque : c'est une "thèse", discutable, pas un théorème puisqu'on cherche justement à formaliser ce qu'est un programme.

Machines de Turing non-déterministes

Comme pour les automates, on définit les *machines de Turing non-déterministes* : dans un état donné, la lecture d'un symbole peut mener à différentes actions (écriture, gauche ou droite, état suivant).

Propriété : les machines de Turing non-déterministes acceptent les mêmes langages que les machines de Turing déterministes.

$\longrightarrow$ comme pour les automates, au prix d'une augmentation exponentielle de la taille de la machine.

Introduction à la calculabilité

Problèmes indécidables

Remarque : l'ensemble des langages est non dénombrable.

car un langage est une partie de l'ensemble des mots que l'on peut former sur un alphabet,
et $\mathcal{P}(\mathbb{N})$ n'est pas dénombrable. (théorème de Cantor 1891)

Mais les machines de Turing et l'ensemble des mots sont dénombrables, donc l'ensemble des langages décidés ($\cup_{M_i}\{w_j$ reconnu par $M_i\}$) aussi.
(union dénombrable d'ensembles dénombrables)

Conclusion : il y a des langages qui ne sont pas reconnus par une machine de Turing.

Corollaire (avec Church) : il y a des problèmes qui ne peuvent pas être résolus par un programme informatique !

On parle de langages / problèmes *indécidables*.

Exemples de problèmes indécidables

Exemple 1 :
Entrée : un programme en Java, et une ligne du code
Problème : la ligne sera-t-elle exécutée ?

$\longrightarrow$ Indécidable. ($\sim$ théorème de Rice 1953)

Exemple 2 :
Dixième problème de Hilbert (1900) :
déterminer si $p(x_1, \ldots, x_n) = 0$ où $p \in \mathbb{Z}[X_1, \ldots, X_n]$ a une solution entière.

$\longrightarrow$ Indécidable. (Yuri Matiyasevich $\sim$ 1970)

Introduction à la calculabilité

Mesure de la complexité algorithmique

Soit $(\forall n)$, $T(n)$ le nombre *maximal* de transitions opérées par une machine de Turing résolvant un problème lorsque l'entrée est codée sur le ruban par n cases.
$\longrightarrow$ codage "raisonnable"
$\longrightarrow$ indépendant de la programmation de l'algorithme, donc on cherche seulement un *ordre de grandeur*.

Exemples :

- Recherche du plus grand entier dans un tableau : $T(n) = \mathcal{O}(n)$
- Résolution du voyageur de commerce par la programmation dynamique : $T(n) = \mathcal{O}(n^2 2^n)$ où n est le nombre de villes.

$\longrightarrow$ on distingue complexité *polynomiale* et *sur-polynomiale* (e.g. *exponentielle*) (raisonnable vs non-raisonnable)
Cf cours programmation dynamique pour des exemples.

La classe P

Définition : un problème est dans la classe P si le langage associé est décidé par une machine de Turing *déterministe* et $T(n) = \mathcal{O}(P(n))$ où P est un polynôme.

Exemple :
Problème : 5 est-il le plus grand élément d'un tableau ?
Résolution :
Donnée : un tableau `tab` à n éléments.
Algorithme : pour `k` variant entre 1 et n, si `tab[k]` > 5 alors répondre "non" sinon `k++`.

$\longrightarrow$ complexité : $\mathcal{O}(n)$.

La classe NP

Définition : un problème est dans la classe NP si le langage associé est décidé par une machine de Turing *non-déterministe* et $T(n) = \mathcal{O}(P(n))$ où P est un polynôme.

Remarque : $P \subset NP$.

Exemple : la recherche d'un cycle hamiltonien est dans NP.

On génère de façon non-déterministe des circuits de longueur n (=nbre de sommets d'une instance)
"i.e." (modulo la lecture de la taille du graphe...)
1) dans l'état q_1, je ne lis rien, et j'écris 1 **ou** 2 **ou** 3...**ou** n et je passe dans l'état q_2
2) dans l'état q_2, je ne lis rien, et j'écris 1 **ou** 2 **ou**...**ou** n et je passe dans l'état q_3
etc. jusque q_n état accepteur.
(complexité en $\mathcal{O}(n)$)

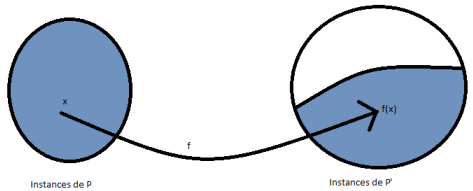
Ensuite, on vérifie que le parcours généré est une permutation des sommets et correspond à un circuit hamiltonien (encore en $\mathcal{O}(n)$).
Si oui, on accepte, sinon on n'accepte pas.

La classe NP-complet

"Définition" : Un problème NP est dit NP-complet s'il est au moins aussi difficile que tout problème de NP.

Théorème : il existe au moins un problème NP-complet (théorème de Cook 1971, SAT).

Question : comment démontre-t-on la NP-complétude ?
$\longrightarrow$ Étant donné un problème NP-complet $\mathcal{P}$ décidé par M et un problème NP $\mathcal{P}'$ décidé par M', si on peut coder toute instance x de $\mathcal{P}$ de taille n en une instance $f(x)$ de $\mathcal{P}'$ de taille au plus $p(n)$ (p polynôme) telle que $f(x)$ est décidé par M' ssi x est décidé par M, alors $\mathcal{P}'$ est NP-complet.



Exemples de problèmes NP-complets

- Existe-t-il un circuit Hamiltonien dans un graphe ?
- Dans un graphe complet, existe-t-il un circuit passant une seule fois par chaque sommet, de longueur $< L$? (voyageur de commerce)
- Problème du sac à dos (*Knapsack problem*)
- Programmation linéaire en nombres entiers
  car les problèmes précédents peuvent être vus comme des pb de PLNE (cf exercices) : réduction polynomiale !
- etc.

P ≠ NP ?

P ⊂ NP : facile.

Question à 1,000,000 de dollars : NP ⊂ P ?
(Millenium prize, Clay Mathematics Institute 2000)

Remarque 1 : par déterminisation des machines de Turing non-déterministes, on passe à complexité exponentielle...

Remarque 2 : si on trouve **un** algorithme polynomial pour **un** problème NP-complet, alors NP ⊂ P.

Remarque 3 : si NP ≠ P, alors **tout** problème NP-complet a une complexité sur-polynomiale.

Introduction à la calculabilité

Des problèmes de décision aux problèmes d'optimisation

Dans le cours de recherche opérationnelle, on s'intéresse à des problèmes d'*optimisation*.

Exemple : Sac à dos.

$\longrightarrow$ on veut remplir un sac à dos avec n objets de poids p_i et de valeur v_i. Le sac à dos à une capacité limitée, et on veut maximiser sa valeur totale.

Problème de décision associé : peut-on remplir le sac à dos de manière à ce que sa valeur soit $\geqslant V$.

Théorème : le problème de décision "sac à dos" est NP-complet.

Question : impact sur problème d'optimisation ?
$\longrightarrow$ si le pb d'optimisation est résoluble par un algorithme polynomial, alors le pb de décision également.

Remarques

Attention à ne pas sur-interpréter la NP-complétude !

- Le problème NP-complet est peut-être plus général que le problème à traiter.
  Exemple (bête) : circuit hamiltonien dans un graphe *complet*...
- C'est de la complexité dans le pire cas : peut-être que l'algo a un comportement polynomial dans 99% des cas.
  Exemple : l'algorithme du simplexe est exponentiel dans le pire cas, mais fonctionne souvent très bien.
  Remarque : PL $\in$ P (Khachiyan 1979)

Introduction à la calculabilité

Conclusion

Il y a des limites à ce qu'on peut faire avec un ordinateur. . .

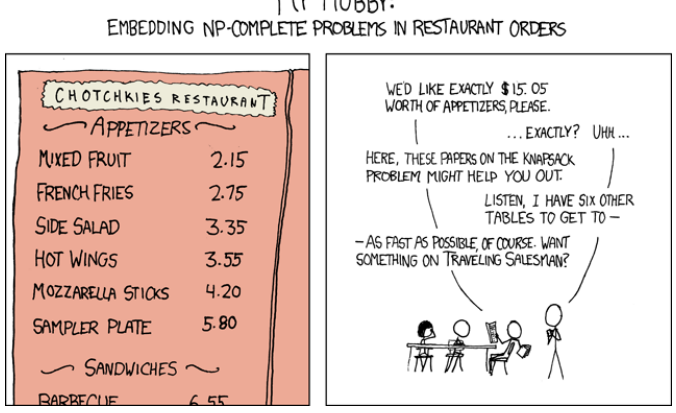
1. Vous ne pouvez pas écrire un programme pour chaque problème.
   (et *personne* ne le peut)
   Ex : résoudre les équations diophantiennes générales.

2. Vous ne pouvez pas tout programmer de manière efficace : attention à la complexité.
   Ex : tournée de ramassage des poubelles.

D'où l'intérêt des *heuristiques* : guider la recherche en évitant de parcourir le nombre sur-polynomial de cas.
Elles donnent des solutions "approchées".

À suivre.

Référence bibliographique



Source : xkcd.com

⟶ Pierre Wolper, *Introduction à la calculabilité*, InterEditions1991. (par exemple)

Calculabilité
F. Sur - ENSMN
Motivation
Que signifie "calculer" ?
Les automates
Les machines de Turing
Indécidabilité
Classes de complexité
P et NP
P ≠ NP ?
Problèmes d'optimisation et complexité
Conclusion