http://www.labri.fr/perso/zeitoun

I Présentation, bref historique

- 1900 : Hilbert, 10e problème : peut-on decider, de façon mécanique, si une equation Diophantienne a une solution ?
  (équation à coefficients entiers ayant un entier pour solution : $x^n + y^n = z^n$)
- 1931 : Gödel publie le théorème d'inplétude (il existe toujours un énoncé qu'on ne peut ni affirmer ni infirmer.)
- 1935 : Turing formalise une définition de machine et de calcul.
- 1936 : Church exhibe un problème non résoluble par machine.
- 1938 : Kleene prouve l'équivalence entre machines de Turing, λ-calcul (introduit par Church), fonctions récursives.
- 1947 : Post et Markov prouvent qu'un problème posé par Thue en 1914 n'est pas résoluble mécaniquement.
- 1970 : Matiyasevich répond négativement au 10e problème de Hilbert.
- 1971 : Cook et Levin formalisent la notion de problème NP complet.

<u>exemples de problèmes de décision :</u>

problème 1 : donnée : un nombre entier positif n en base 2 :
question : n est-il pair ?
⇒ problème décidable : on regarde le dernier chiffre.

problème 2 : D : un nombre entier positif n en base 10.
Q : n est-il premier ?
⇒ problème décidable : pour l'algorithme naif complexité exponentielle (en $O(n)$) → pour i de 2 à n-1
si i/n retourner faux
retourner vrai.

possibilité de résolution grace à un algorithme à complexité polynomiale.

problème 3 : D un automate $\mathcal{A}$ et un mot x

Q : $\mathcal{A}$ accepte-t-il x ?

⇒ problème décidable

problème 4 : D : un programme C.

Q : le programme est-il syntaxiquement correct ?

⇒ problème décidable grace à un compilateur.

problème 5 : D un graphe donné par une liste d'adjacence.

Q le graphe est-il 3-coloriable ?

⇒ problème décidable mais difficile de façon efficace, il n'existe pas d'algorithme à complexité polynomiale.

problème 6 : un puzzle Eternity (http://fr.eternityii.com)

le puzzle a-t-il une solution ?

⇒ problème décidable mais pas de façon efficace.

problème 7 : un programme C

le programme s'arrête-t-il sur au moins l'une de ses entrées ?

⇒ problème non décidable : les compilateurs ne peuvent que détecter les lignes qui ne servent à rien // not reached.

problème 8 : un programme C

le programme s'arrête-t-il sur toutes ses entrées ?

⇒ problème non décidable.

problème 9 : des couples de mots $(u_1, v_1), \dots, (u_m, v_m)$

existe-t-il des entiers $i_j, \dots i_k$ tels que $u_j \dots u_k = v_j \dots v_k$?

⇒ problème non décidable : problème de correspondance de Post

http://www.theory.informatik.uni-kassel.de/~staner/pcp/pcpcontest_en.html

Notion de problème de décision

Un problème de décision est une question portant sur un ensemble de données dont une description est fixée et dont la réponse est oui ou non.

Ne pas confondre problème et algorithme le résolvant.

Une instance du problème est la question posée sur une donnée particulière.

II Ensembles dénombrables. Un paradoxe

Un ensemble E est dit dénombrable s'il est en bijection avec $\mathbb{N}$.

$f : E \to F$ est

- injective si $\forall x, y \in E$ et $x \neq y$ on a $f(x) \neq f(y)$
- surjective si $\forall z \in F, \exists t \in E$ tel que $f(t) = z$
- bijective si injective et surjective, il y a le même nombre d'éléments dans E et dans F.

Un ensemble fini ou dénombrable est dit au plus dénombrable.

exemples d'ensembles dénombrables :

1. l'ensemble $\mathbb{N}^*$ des entiers strictement positifs.

$$f : \mathbb{N}^* \to \mathbb{N}$$
$$n \mapsto n - 1$$

2. pour un entier $k > 0$ donné, l'ensemble des entiers divisibles par k.

$$f : k\mathbb{N} = \{ kn \mid n \in \mathbb{N} \} \to \mathbb{N}$$
$$kn \mapsto n$$

3. l'ensemble des entiers qui sont des puissances de 2

$$f : \mathbb{N} \to \{ \text{puissances de } 2 \}$$
$$n \mapsto 2^n$$

Si $f : E \to F$ est bijective, il existe une unique fonction $g : F \to E$ telle que $f(g(x)) = x \ \forall x \in F$ et $g(f(x)) = x \ \forall x \in E$. g est une bijection.

4. l'ensemble des nombres premiers

$f : \mathbb{N} \to \text{Premiers}$

$n \mapsto (n+1)^{\text{ème}}$ nombre premier

5. l'ensemble des entiers relatifs

$f : \mathbb{Z} \to \mathbb{N}$

$$x \mapsto \begin{cases} |2x| & \text{si } x \leq 0 \\ 2x-1 & \text{si } x > 0 \end{cases}$$

Propriétés des ensembles dénombrables :

1. Toute partie d'un ensemble dénombrable est au plus dénombrable.

démonstration :

soit E un ensemble dénombrable.

montrons que $X \subset E$ est dénombrable.

comme E est dénombrable, il existe une bijection $f : E \to \mathbb{N}$

donc il existe une bijection $g : X \to f(X) = \{f(x) \mid x \in X\}$

g est définie par $\forall x \in X$, $g(x) = f(x)$

or $f(X)$ est un ensemble infini d'entiers donc il est dénombrable

$X \xrightarrow{g} f(X) \xrightarrow{h} \mathbb{N}$ (h est une bijection)

$h \circ g$ est une bijection de X dans $\mathbb{N}$.

2. Soit $f : A \to B$ une application

- si f est injective et B dénombrable alors A est au plus dénombrable.
- si f est surjective et A dénombrable alors B est au plus dénombrable.

3. Si A et B sont dénombrables, alors $A \times B$ l'est aussi.

Si $A_1, \dots, A_m$ sont dénombrables, alors $\prod_{i=1}^{m} A_i$ l'est aussi (récurrence)

4. Si J est au plus dénombrable, et si A_j est au plus dénombrable pour $j \in J$, alors $\bigcup_{j \in J} A_j$ est au plus dénombrable.

5. Si $A \neq \emptyset$ est au plus dénombrable, alors A^* est dénombrable.

6. L'ensemble des automates finis sur l'alphabet A fini est dénombrable.

7. L'ensemble des programmes C est dénombrable.

Démontrons que si A et B sont dénombrables alors $A \times B$ l'est aussi :

$A \xrightarrow{f_A} \mathbb{N}$

$B \xrightarrow{f_B} \mathbb{N}$

$A \times B$ est une bijection dans $\mathbb{N} \times \mathbb{N}$

$f : A \times B \to \mathbb{N} \times \mathbb{N}$

$(a, b) \longmapsto (f_A(a), f_B(b))$

$\mathbb{N} \times \mathbb{N}$ est en bijection avec $\mathbb{N}$ donc par composition $A \times B$ aussi.

<u>exemples d'ensembles non dénombrables :</u>

1. L'ensemble des nombres réels.

2. L'ensemble des suites infinies d'entiers.

démonstration par l'absurde : (Cantor)

$\mathcal{S} = \{ (x_m)_{m \geq 0} \mid x_m = 0 \text{ ou } x_m = 1 \ \forall m \}$

supposons qu'il existe une bijection $f : \mathbb{N} \to \mathcal{S}$

$f(0) = (u_0^0, u_1^0, \dots, u_m^0, \dots)$

$f(k) = (u_0^k, u_1^k, \dots, u_m^k, \dots)$

La suite d'entiers $(1 - u_0^0, 1 - u_1^1, \dots, u_m^m, \dots) \neq f(i) \ \forall i \in \mathbb{N}$

3. L'ensemble des parties de $\mathbb{N}$.

Soit $\mathcal{P}(\mathbb{N})$ cet ensemble, il existe une bijection naturelle de $\mathcal{P}(\mathbb{N})$ dans $\mathcal{S}$ (ensemble des suites infinies d'entiers).

$2\mathbb{N} \in \mathcal{P}(\mathbb{N})$: ensemble des entiers pairs

$2\mathbb{N} = \{0, 2, 4, 6, \dots\} \longmapsto f(2\mathbb{N}) = (1, 0, 1, 0, 1, 0, 1, \dots)$

Soit $X \in \mathcal{P}(\mathbb{N})$, $f(X) = (x_m)_m$, suite définie par $x_m = \begin{cases} 1 & \text{si } m \in X \\ 0 & \text{sinon} \end{cases}$

$f : \mathcal{P}(\mathbb{N}) \to \mathcal{S}$ est une bijection.

Un paradoxe :

- On suppose qu'on dispose d'un ordinateur à mémoire infinie.
- On considère les programmes qui écrivent une suite de 0 ou 1 ...
- On interprète une telle suite $a_1, a_2, \ldots$ comme le réel $0, a_1 a_2 \ldots$
  Un tel réel est dit calculable.
- L'ensemble de ces programmes est dénombrables : $\{P_1, P_2, \ldots\}$.
- On considère le programme qui lance P_1 jusqu'à ce que celui-ci s'apprête à écrire a_1 et écrit à la place un entier différent de a_1. Puis il lance P_2 jusqu'à ce que celui-ci s'apprête à écrire a_2 et écrit à la place un entier différent de a_2. etc.
- Ce programme écrit un réel calculable non calculable !

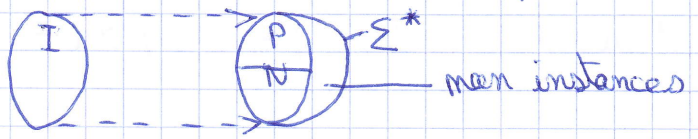
→ Si le programme P est dans la liste alors il y aura une boucle infinie.
La génération de la liste $P_1, P_2, \ldots$ est non triviale.

III Machines de Turing

Notion de problème : formalisation

Un problème de décision est la donnée d'un ensemble dénombrable I d'instances et d'un sous ensemble $P \subseteq I$ d'instances positives.

Pour Σ fini, un codage des instances est une application injective $\langle \cdot \rangle : I \to \Sigma^*$ associant à chaque élément x de I un mot $\langle x \rangle \in \Sigma^*$



Un problème de décision peut se voir comme une fonction $f_P : \Sigma^* \to \{\text{oui}, \text{non}\}$ calculant le langage L_P, au sens que $f_P(\langle x \rangle) = \text{oui} \iff x \in P$

Plus généralement, on s'intéresse au calcul de fonctions $f : \Sigma_1^* \to \Sigma_2^*$

Notion de machine

Résoudre un problème consiste à déterminer pour $x \in I$, si $\langle x \rangle \in L_p$.
On veut une machine acceptant tous les mots de L_p.
On peut reconnaître tous les langages avec un automate à nombre infini d'états.

Machines de Turing

Les machines de Turing sont des abstractions d'algorithmes, des ordinateurs.
Une machine de Turing comporte :

- une bande infinie à droite et à gauche faite de cases consécutives.
- dans chaque case se trouve un symbole (éventuellement blanc $\square$).
- une tête de lecture-écriture.
- un contrôle à nombre fini d'états.

L'accès à la mémoire (représentée par la bande) est séquentiel : la machine peut bouger sa tête à droite ou à gauche d'une case à chaque étape.

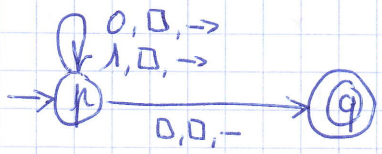
Une machine de Turing à une bande $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$ est donnée par :

- Q : ensemble fini d'états
- q_0 : état initial
- $F \subseteq Q$: ensemble d'états finaux ou acceptants
- Σ : alphabet d'entrée avec $\Sigma \subseteq \Gamma \setminus \{\square\}$.
- Γ : alphabet fini de la bande, avec $\square \in \Gamma$
- δ : ensemble des transitions

Une transition est de la forme (p, a, q, b, d), notée $p \xrightarrow{a,b,d} q$ avec $p, q \in Q$; $a, b \in \Gamma$ et $d \in \{\leftarrow, \rightarrow, -\}$.
On supposera qu'aucune transition ne part d'un état de F.

On représente souvent une machine de Turing comme un automate, seules changent les étiquettes des transitions.

exemple avec $\Gamma = \{0, 1, \square\}$ et $\Sigma = \{0, 1\}$



représente la machine avec $Q = \{p, q\}$, $q_0 = p$, $F = \{q\}$ et $\delta = \{(p, 0, \square, \rightarrow, p), (p, 1, \square, \rightarrow, p), (p, \square, \square, -, q)\}$

Cette machine accepte tous les mots et efface son entrée.

Fonctionnement d'une machine de Turing

- Initialement, un mot w est écrit sur la bande entouré de $\square$.
- Un calcul d'une MT sur w est une suite de pas de calcul. Cette suite peut être finie ou infinie.
- Le calcul commence avec la tête de lecture écriture sur la première lettre de w dans l'état q_0.
- Chaque pas de calcul consiste à appliquer une transition, si possible.
- Le calcul ne s'arrête que si aucune transition n'est applicable.
- L'application d'une transition de la forme $p \xrightarrow{a,b,d} q$ consiste à
  - changer l'état de contrôle qui devient q,
  - remplacer le contenu de la case sous la tête de lecture par b
  - bouger la tête du case ou pas en fonction de d.

Configurations et calculs

Une configuration représente un instantanné du calcul.
La configuration uqv signifie que l'état de contrôle est q, le mot écrit sur la bande est uv (entouré par des $\square$) et la tête de lecture est sur la première lettre de v. La configuration initiale sur w est donc $q_0 w$.
Pour deux configurations C et C', on écrit $C \vdash C'$ lorsqu'on obtient C' par application d'une transition à partir de C.
Un calcul d'une machine de Turing est une suite de configurations : $C_0 \vdash C_1 \vdash \ldots$
Il y a trois cas possibles :
- le calcul est infini
- le calcul s'arrête sur un état final (de F)
- le calcul s'arrête sur un état non final.

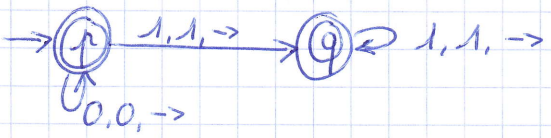
Langages acceptés

On peut utiliser une machine de Turing pour accepter des mots.
Le langage $\mathcal{L}(M) \subseteq \Sigma^*$ des mots acceptés par une machine de Turing M est l'ensemble des mots w sur lesquels il existe un calcul fini $C_0 \vdash C_1 \vdash \dots C_m$ avec $C_0 = q_0 w$ et $C_m \in \Gamma^* F \Gamma^*$.
On dit qu'une machine est déterministe si, pour tout $(p, a) \in Q \times \Gamma$, il existe au plus une transition de la forme $p \xrightarrow{a,b,d} q$.
Si M est déterministe, elle n'admet qu'un calcul par entrée.

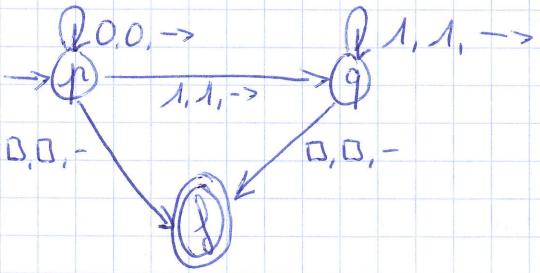
exemples de machine de Turing :

1. machine qui effectue utile (true) :

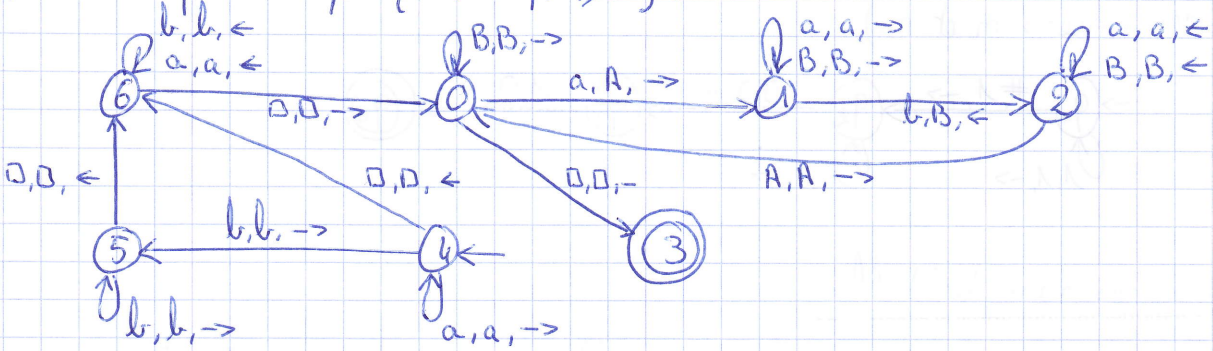
2. machine qui accepte $0^* 1^*$:



mais des transitions partent des états finaux :

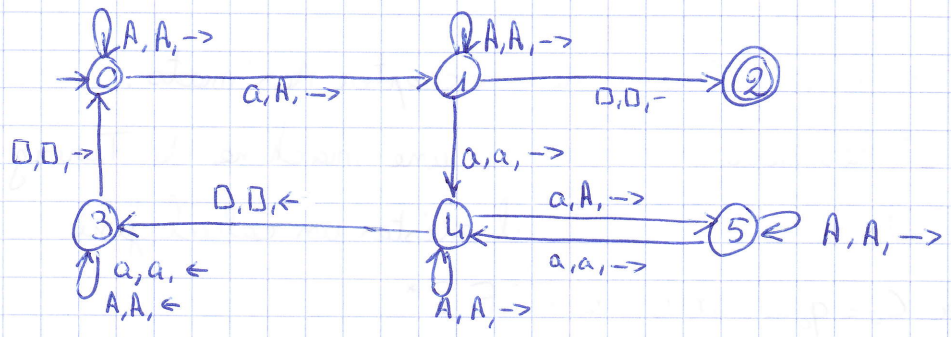


3. machine qui accepte $\{a^m b^m \mid m \geq 0\}$:

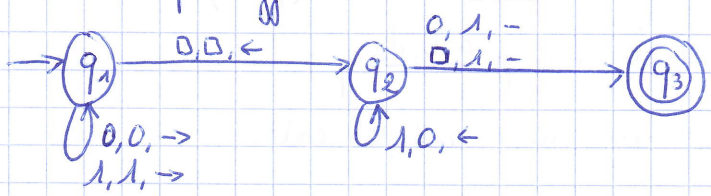


On vérifie qu'on est dans $a^* b^*$ puis on marque le premier a et le premier b et on recommence.

4. machine qui accepte $\{a^{2^m} \mid m \geq 0\}$



5. machine qui effectue l'incrément en binaire :



$q_1 10111$

$\vdash 1 q_1 0111$

$\vdash 10 q_1 111$

$\vdash 101 q_1 11$

$\vdash 1011 q_1 1$

$\vdash 10111 q_1$

$\vdash 1011 q_2 1$

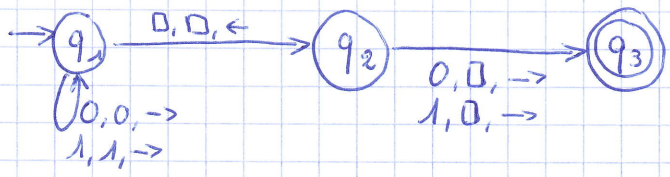
$\vdash 101 q_2 1 0$

$\vdash 10 q_2 100$

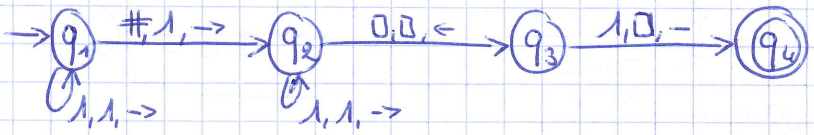
$\vdash 1 q_2 0000$

$\vdash 1 q_3 1000$

6. machine qui calcule $m \div 2$:



7. machine qui effectue l'addition en unaire :
(le mot d'entrée est de la forme $1^m \# 1^m$ interprété comme la donnée des entiers m et m).



Langages non décidables

Thèse de Church : les machines de Turing capturent tout ce qui est calculable.

Un langage L est décidable (ou récursif) s'il existe une machine de Turing qui : - s'arrête sur F partant d'un mot de L,

- s'arrête sur $Q \setminus F$ partant d'un mot de $\Sigma^* \setminus L$.

L'ensemble des langages sur $\{0,1\}^*$ est non dénombrable.

L'ensemble des machines de Turing est dénombrable.

$\Rightarrow$ Il existe donc des langages non décidables.

Variations des machines de Turing

On peut changer la définition des machines de Turing de plusieurs façons :

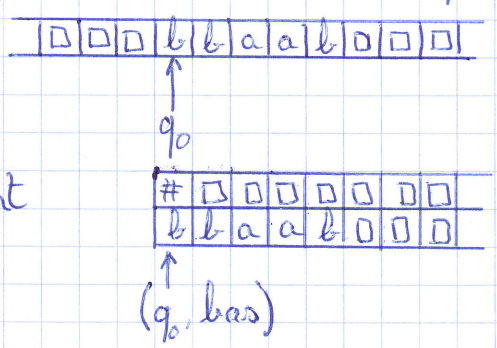
- bande finie sur la gauche et infinie sur la droite.
- un unique état acceptant et un unique état rejetant.
- calculs replacant la tête en début de bande...
  ... et terminant avec le mot d'entrée sur la bande.
- machines déterministes.
- petit alphabet de bande : $\Gamma = \Sigma \cup \{\square\}$.
- machines à plusieurs bandes et plusieurs têtes.

Ces variations n'affectent pas ce que l'on peut accepter ou calculer.

Intuitivement, une machine $M2$ simule une machine $M1$ lorsque $M2$ peut effectuer les mêmes calculs que $M1$. $M2$ simule $M1$ si elle peut effectuer les mêmes passages entre configurations (éventuellement par plusieurs transitions).

Bandes finies à gauche :

On replie la bande et on représente la configuration initiale.



On a $\Gamma_2 = (\{\#\} \cup \Gamma_1) \times \Gamma_1$ et $Q_2 = Q_1 \times \{haut, bas\}$

est un nouveau symbole servant à repérer le début de bande.

reste à écrire les transitions :

si $p \xrightarrow{a,b,\rightarrow} q$ est une transition de la MT à bande bi-infinie, on ajoute les transitions suivantes à la MT à bande infinie à droite :

$(p, bas) \xrightarrow{(x,a),(x,b),\rightarrow} (q, bas) \quad \forall x \in \Gamma \cup \{\#\}$

$(p, haut) \xrightarrow{(a,x),(b,x),\leftarrow} (q, haut) \quad \forall x \in \Gamma$.

Inversement, toute machine travaillant sur une bande finie peut être simulée par une machine à bande bi-infinie.

A partir de $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$, on construit $M' = (Q', q_0', F', \Sigma', \Gamma', \delta')$.

On ajoute un état OK, <u>seul état acceptant</u> de la nouvelle machine M' et des transitions de F vers OK.

$Q' = Q \cup \{OK\}$, $q_0' = q_0$, $F' = \{OK\}$, $\Sigma' = \Sigma$, $\Gamma' = \Gamma$, $\delta' = \delta \cup (F \times \Gamma \times \{OK\} \times \Gamma \times \{-\})$.

On peut de même transformer M pour qu'il y ait un <u>unique état rejetant</u> KO.

Composition des machines de Turing :

En connectant l'état final (supposé unique) d'une machine M_1 à l'état initial d'une machine M_2, on compose les fonctions calculées par les deux machines.

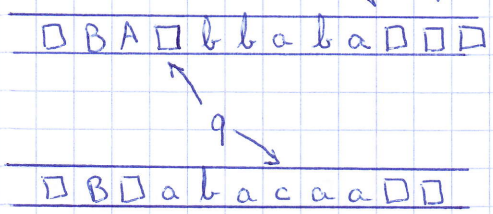
- composition séquentielle : $M_1 ; M_2$
- test : si M_1 alors M_2 sinon M_3
- boucle : tant que M_1 faire M_2.

Plusieurs bandes :

On peut utiliser plusieurs bandes.

Initialement, le mot d'entrée est écrit sur une bande. Chaque transition lit un symbole sur chaque bande, et en fonction des symboles lus et de l'état, la machine change d'état, écrit un nouveau symbole sous chacune des têtes, déplace chaque tête indépendamment.

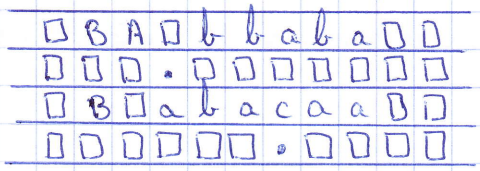
simulation possible, regrouper les bandes :



temps de calcul de la machine du haut : $f(n)$

temps de calcul de la machine du bas : $(f(n))^2$.

est représenté par :



idée de la simulation : position des têtes de lecture stockée dans la bande.

Chaque transition de la machine originale est simulée par :

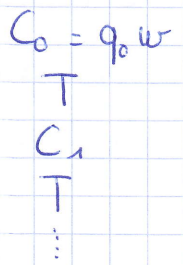
- un aller sur la bande pour récupérer les symboles sous les têtes
- un retour pour simuler la transition.

Contrairement à la simulation de bande infinie par bande finie, plusieurs mouvements sont nécessaires pour simuler un seul mouvement. La nouvelle machine est intuitivement plus lente.

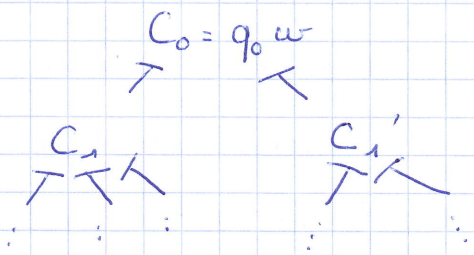
Machines déterministes

Comme pour les automates finis, pour toute machine M, on peut construire une machine déterministe qui simule M.

calcul d'une machine déterministe :

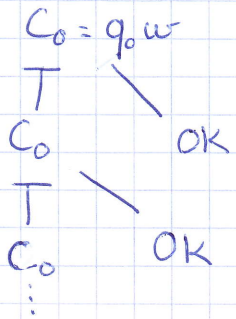


calcul d'une machine non déterministe :



idée : on parcours l'arbre des calculs en largeur.

(pas de parcours en profondeur pour ne pas boucler :



On note r le nombre maximal de choix dans M.
On utilise trois bandes :

- bande 1 : sauvegarde la valeur du mot d'entrée w.
- bande 2 : génère dans l'ordre hiérarchique les suites finies d'entiers sur $\{1, \dots, r\}$.
- bande 3 : effectue le calcul de M sur w correspond à la suite de choix écrite sur la bande 2.

IV Machines RAM

Une machine RAM est une machine possédant :

- un programme, qui est une suite d'instructions finie $I_0, I_1, \dots$
- une suite infinie de registres $R_0, R_1, \dots$
- deux registres spéciaux :
  - l'accumulateur A,
  - le compteur de programmes PC (program counter)
- une bande d'entrée sur laquelle la machine lit ses données.
- une bande de sortie sur laquelle la machine écrit ses résultats.

Chaque registre peut mémoriser n'importe quel entier positif ou nul. Initialement, tous les registres ont comme valeur 0. Le registre PC contient le numéro de la prochaine instruction à exécuter. Les autres registres, y compris l'accumulateur, contiennent des valeurs manipulées par la machine au cours du calcul.

Les instructions sont de quatre types :

- manipulation de registres
- opérations arithmétiques
- sauts et arrêt
- entrées / sorties.

instructions de manipulation de registres :

- load : charge une valeur dans l'accumulateur A :
  - load # <m> : met l'entier m dans l'accumulateur (adressage immédiat)
  - load <m> : met l'entier contenu dans R_m dans A (adressage direct)
  - load (<m>) : met l'entier contenu dans R_{R_m} dans A (adressage indirect).
- store : range la valeur contenue dans A dans un registre :
  - store <m> : met la valeur contenue dans A dans R_m
  - store (<m>) : met la valeur contenue dans A dans R_{R_m}.

opérations arithmétiques :

- incr : incrémente de 1 la valeur contenue dans l'accumulateur
- decr : décrémente de 1 la valeur contenue dans A si elle est strictement positive, ne fait rien sinon.

sauts et arrêt :

- jump <m> : saut inconditionnel à l'instruction I_m.
- jz <m> : saut à l'instruction I_m si l'accumulateur est nul.
- stop : fin du programme.

En pratique, on se permet de mettre des étiquettes au niveau des instructions vers lesquelles on veut aller.

instructions d'entrées / sorties :

Sur la bande d'entrée se trouve une suite d'entiers (les entrées du programme).

- read : lit la valeur courante de la bande d'entrée et la met dans l'accumulateur. La prochaine valeur lue sera la suivante de la bande d'entrée.
- write : écrit sur la bande de sortie la valeur contenue dans A.

La machine exécute les instructions en commençant par I_0. Une fois

l'instruction I_m exécutée, c'est l'instruction I_{m+1} qui est exécutée, sauf après une instruction de saut ou stop.

Jeux d'instructions des machines RAM VS. processeurs

Les machines RAM

+ peuvent manipuler des entiers arbitraires,
+ peuvent utiliser un nombre arbitraire de registres
− ont un jeu d'instructions plus restreint que celui des processeurs habituels.

Mais on peut reprogrammer les instructions manquantes

→ opérations arithmétiques (addition, multiplication, soustraction, ...)
→ appel de sous programmes : call / return (nécessite un décalage de registres).

exemples:

On a sur la bande d'entrée les entiers x et y.
On veut sur la bande de sortie $x+y$.

0. read	8. load 1
1. store 0	9. incr
2. read	10. store 1
3. store 1	11. jump 4
4. load 0	12. load 1
5. jz 12	13. write
6. decr	14. stop
7. store 0	

pour la multiplication:

$R_2 = x$

$R_3 = y$

$R_4 = 0$

on fait R_2 -- et $R_4 += R_3$, si $x == 0$, write R_4.

0. read	10. load 3	19. incr
1. store 2	11. store 0	20. store 1
2. read	12. load 4	21. jump 14
3. store 3	13. store 1	22. load 1
4. load #0	14. load 0	23. store 4
5. store 4	15. jz 22	24. jump 6
6. load 2	16. decr	25. load 4
7. jz 25	17. store 0	26. write
8. decr	18. load 1	27. stop
9. store 2		

Décaler les registres utilisés par une RAM pour laisser les registres 0 à k-1 libres :

load <m> ⇝ load <m+k+3> (on se réserve 3 registres de travail).

store <m> ⇝ store <m+k+3>

load (<m>) ⇝ load <m+k+3>

```
    store <k>
    load #<k+3>
    store <k+1>
L:  load <k>
    jz K
    decr
    store <k>
    load <k+1>
    incr
    store <k+1>
    jump L
K:  load (<k+1>)
```

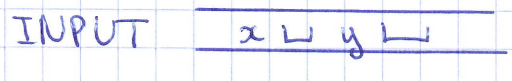
Simulation par machines de Turing

Une machine de Turing peut simuler une machine RAM : tout programme RAM peut être réaliser par une machine de Turing à plusieurs bandes :

- une bande d'entrée sur laquelle on met les données séparées par des □
- une bande de sortie
- une bande pour l'accumulateur (en binaire par exemple)
- une bande pour tous les autres les registres, séparés par □
- une bande de travail servant à retrouver le contenu d'un registre.

exemple :

machine RAM :



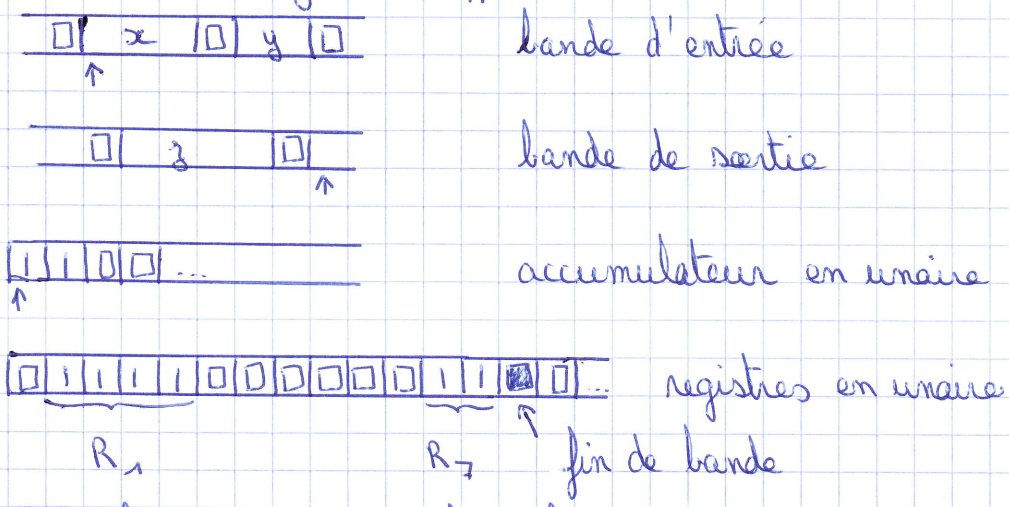
A $\boxed{2}$

R_1 $\boxed{4}$

R_7 $\boxed{2}$

(tous les autres registres sont nuls)

machine de Turing : (on suppose des bandes unidirectionnelles)



avant le 1e □, on met la valeur de R_0.

avant le 2e □, on met la valeur de R_1.

⋮

Chaque instruction est codée par une partie de la machine.

Les instructions sont reliées entre elles grâce aux états.

- incr / decr : on ajoute ou on supprime un 1 à l'accumulateur.
- load<n> : recherche du contenu de R_n = on initialise la bande de travail

à m et on place la tête de la bande des registres derrière le m-ième □.
Puis, on recopie dans A.

- jump q : correspond aux transitions $p \xrightarrow{x,x,-} q$ pour $x \in \Gamma$.
- jz q : idem si l'accumulateur est nul. Sinon, aller en $p+1$.
- stop : aller dans l'état OK.

On peut compiler un programme C (sans entrée-sortie) vers un programme RAM.
La simulation précédente montre qu'on peut compiler tout programme RAM en machine de Turing.
Tout programme sans entrée-sortie dans n'importe quel langage de programmation actuel se compile donc en machine de Turing.

Inversement, on peut simuler toute machine de Turing par une RAM.

V Problèmes indécidables

Définitions :

Un langage L est récursivement énumérable s'il est le langage accepté par une machine de Turing.
RE = classe des langages récursivement énumérables.
Note : sur un mot $\Sigma^* \setminus L$, la machine de Turing peut ne pas s'arrêter.
→ Il existe une machine de Turing M qui accepte tous les mots de $L \in RE$, il faut que M finisse sur OK. M finit sur KO ou boucle si $m \notin L$.
Un langage L est décidable ou récursif s'il est le langage accepté par une machine de Turing qui s'arrête sur toute entrée.
R = classe des langages récursifs.
→ Il existe une machine de Turing M' qui accepte tous les mots de $L \in R$. Elle termine sur OK si $m \in L$, sur KO sinon.

Rationnel $\subset$ Algébrique $\subset$ R $\subseteq$ RE

La classe des langages RE est fermée par union.
($\Leftrightarrow$ si $L_1 \in RE$ et $L_2 \in RE$ alors $L_1 \cup L_2 \in RE$)
La classe des langages R est fermée par union et complément.
($\Leftrightarrow$ si $L \in R$ alors $\overline{L} \in R$)
Si L et $\Sigma^* \setminus L$ sont dans RE alors ils sont dans R :

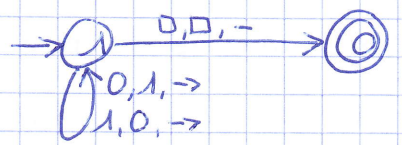
- on construit une machine qui simule en parallèle les machines M et N acceptant L et $\Sigma^* \setminus L$ (elle alterne un pas de calcul de M et un pas de calcul de N)
- cette machine s'arrête si l'une des machine s'arrête.
- le mot appartient soit à L, soit à $\Sigma^* \setminus L$, donc elle s'arrête toujours.

Codage des machines de Turing

On travaille sur $\Sigma = \{0, 1\}$ et on peut supposer $\Gamma = \{0, 1, \square\}$.
On ne considère que les MT à un unique état OK.
Toute MT de ce type peut se coder sur l'alphabet $\{0, 1\}$.
On code une transition $p_i \xrightarrow{a_j, b_k, d_l} p_m$ par $0^i 1 0^j 1 0^k 1 0^l 1 0^m$.
On code la machine M par la suite de ses transitions, séparées entre elles par 11 avec deux blocs 111 en début et fin.
On note ce code $\langle M \rangle$.
Si w est un mot, on note $\langle M, w \rangle$ le code M suivi par w.

exemple :

M :



$0 = a_0$ $\quad$ $\leftarrow : 0$

$1 = a_1$ $\quad$ $- : 1$

$\square = a_2$ $\quad$ $\rightarrow : 2$

$\langle M \rangle = 111\ 0110100 10$

$11\ 0101100 10$

$11\ 0100100 101$

111

Le langage diagonal

On numérote les mots de $\Sigma = \{0,1\}^*$ dans l'ordre hiérarchique (longueur d'abord, ordre lexicographique pour une longueur donnée).

$w_0 = \varepsilon$, $w_1 = 0$, $w_2 = 1$, $w_3 = 00$, $w_4 = 01$, ...

Une machine de Turing peut calculer le numéro d'un mot.

On note M_i la machine telle que $\langle M_i \rangle = w_i$ (par convention $\mathcal{L}(M_i) = \emptyset$ si w_i n'est pas le code d'une machine de Turing).

Le langage $L_d = \{w_i \mid w_i \notin \mathcal{L}(M_i)\}$ n'est pas RE.

	w_1	w_2	...	w_k
M_1				
M_2				
$\vdots$				
M_k	0	1	...	0

le mot w_1 n'est pas reconnu par M_k.

le mot w_2 est reconnu par M_k.

$w_k \in L_d$, L_d est indécidable.

Démontrons que L_d n'est pas RE par l'absurde :

supposons qu'il existe M telle que $\mathcal{L}(M) = L_d$

donc il existe i tel que $M = M_i$

1e cas : M_i accepte w_i, $w_i \in L_d$

donc $\underbrace{w_i \notin \mathcal{L}(M_i)}_{\text{par définition de } L_d} \underbrace{= \mathcal{L}(M) = L_d}_{\text{par définition de } M}$ $\longrightarrow$ absurde

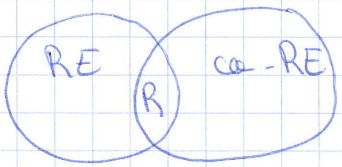
2e cas : $w_i \notin L_d$

donc $w_i \in \mathcal{L}(M_i) = \mathcal{L}(M) = L_d$ $\longrightarrow$ absurde.

Donc L_d est non RE.

$\Sigma^* \setminus L_d = \{w_i \mid w_i \in \mathcal{L}(M_i)\}$ non R mais RE

donc $L_d \in$ co-RE ($\Leftrightarrow$ le complémentaire est dans RE)



Réductions

Soient P_A et P_B deux problèmes.

On note L_A l'ensemble des instances positives de P_A et L_B l'ensemble des instances positives de P_B.

Une réduction de P_A vers P_B est une fonction calculable par MT $f : \Sigma^* \to \Sigma^*$ telle que $x \in L_A \Leftrightarrow f(x) \in L_B$.

On note $P_A \leqslant P_B$ (P_A se réduit à P_B).

L'existance d'une réduction de P_A vers P_B assure que

- si P_B est décidable alors P_A l'est aussi
- si P_A est indécidable alors P_B l'est aussi.

Langage et machine universelle

Le langage $L_u = \{ \langle M, w \rangle \mid M \text{ accepte } w \}$ est RE mais pas R.

S'il était dans R, alors $\Sigma^* \setminus L_d$ le serait aussi.

A partir d'une machine de Turing hypothétique M qui s'arrête sur toute entrée et telle que $\mathcal{L}(M) = L_u$, on construit une machine de Turing M' qui s'arrête sur toute entrée et telle que $\mathcal{L}(M') = \Sigma^* \setminus L_d$.

$\Sigma^* \setminus L_d \leqslant L_u$: on construit la reduction de $\Sigma^* \setminus L_d$ à L_u.

$w_i \in \Sigma^* \setminus L_d \Leftrightarrow f(w_i) \in L_u$.

Soit $f : \Sigma^* \to \Sigma^*$ définie par $f(w_i) = \langle M_i, w_i \rangle$

$$\begin{aligned} w_i \in \Sigma^* \setminus L_d &\Leftrightarrow w_i \in \mathcal{L}(M_i) \\ &\Leftrightarrow \langle M_i, w_i \rangle \in L_u \\ &\Leftrightarrow f(w_i) \in L_u \qquad (\mathcal{E}) \end{aligned}$$

De $(\mathcal{E})$, on deduit une machine de Turing M' pour décider $\Sigma^* \setminus L_d$ (ce qui donne une contradiction car nous savons $\Sigma^* \setminus L_d$ n'est pas décidable).

M' fonctionne ainsi :

- à partir de w_i, elle construit $\langle M_i, w_i \rangle$
- elle lance M sur $\langle M_i, w_i \rangle$.

Un autre langage indécidable

Le langage $L_\emptyset = \{ \langle M \rangle \mid \mathcal{L}(M) = \emptyset \}$ n'est pas RE.

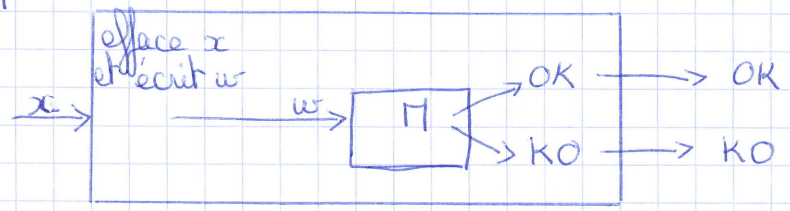
S'il était dans R, alors L_u le serait aussi.

On montre que $\Sigma^* \setminus L_\emptyset$ est RE. On construit une réduction $L_u \leq \Sigma^* \setminus L_\emptyset$.

A partir d'un algorithme pour décider $L_\emptyset$, on décrit un algorithme pour décider L_u. On doit construire, à partir de $\langle M, w \rangle$, une machine $M' = f(\langle M, w \rangle)$ telle que $\langle M, w \rangle \in L_u \Leftrightarrow M' \in L_\emptyset$.

C'est à dire, $w \in \mathcal{L}(M) \Leftrightarrow \mathcal{L}(M') = \emptyset$.

M'




```
M'(x) {
    if ( M(w) == OK)
        return OK;
    return KO;
}
```

donc $\Sigma^* \setminus L_\emptyset$ est non décidable puisque L_u est non décidable.

Donc $L_\emptyset$ n'est pas dans R.

Théorème de Rice

Soit $\mathcal{E}$ l'ensemble des langages RE de $\{0,1\}^*$.

Une propriété des langages RE est un sous ensemble $\mathcal{P}$ de $\mathcal{E}$.

Une propriété $\mathcal{P} \subseteq \mathcal{E}$ est triviale si $\mathcal{P} = \emptyset$ ($\mathcal{P}$ ne contient aucun langage ; à ne pas confondre avec $\mathcal{P} = \{\emptyset\}$: $\mathcal{P}$ ne contient que le langage vide) ou $\mathcal{P} = \mathcal{E}$.

$\mathcal{P} = \emptyset$ et $\mathcal{P} = \mathcal{E}$ sont deux propriétés décidables.

Toute propriété non triviale $\mathcal{P}$ des langages RE est non décidable.

Attention : il s'agit d'une propriété des langages, non des MT.

Preuve : à nouveau une réduction à partir de L_u.

Quitte à changer $\mathcal{P}$ et $\mathcal{E} \setminus \mathcal{P}$, on peut supposer $\emptyset \notin \mathcal{P}$.
Comme $\mathcal{P} \neq \emptyset$, il existe $L \in \mathcal{P}$. Soit M_L telle que $\mathcal{L}(M_L) = L$. A partir de M, w donné, on construit la machine de Turing suivante :

Cette machine accepte $\emptyset \notin \mathcal{P}$ si $\langle M, w \rangle \notin L_u$ et $L \in \mathcal{P}$ sinon.
Donc si $\mathcal{P}$ était dans R, L_u le serait aussi.

<u>exemple de problèmes indécidables :</u>

- étant données une machine M et un mot w, M s'arrête-t-elle sur w ?
- étant donnée une machine M, s'arrête-t-elle sur ε ?
- étant donnée une machine M, s'arrête-t-elle sur au moins une entrée ?
- étant donnée une machine M, s'arrête-t-elle sur toute entrée ?
- étant données une machine M et un état q de M, M utilise-t-elle l'état q sur l'entrée ε ?

```
bool termine (int (*f)(int), int x);
  //retourne vrai si f se termine, faux sinon.
int f_test (int x) {
    if (termine (f_test, x))
        while (true);
    return 0;
}
```

<u>Indécidabilité hors du monde des MT : le PCP</u>

Problème de correspondance de Post (1946)
Donnée : n paires de mots $(u_1, v_1), \dots, (u_n, v_n)$.
Question : existe-t-il une suite finie $i_1, \dots, i_k$ telle que $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$
(les suites d'indices sont les mêmes).

On peut voir les couples de mots comme des dominos :

a	aa	ba	bab
ab	a	aa	abba

une solution : a.bab.ba.aa.aa = ab.abba.aa.a.a

la suite d'indices : 1, 4, 3, 2, 2

Le PCP modifié (PCPM)

donnée : m paires de mots $(u_1, v_1), \dots, (u_m, v_m)$

question : existe-t-il une suite finie $i_1, \dots, i_k$ telle que $i_1 = 1$

et $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$.

On montre que $L_u \leq$ PCPM $\leq$ PCP. (L_u = langage universel)

Comme L_u est indécidable, il en est de même pour PCPM et PCP.

On peut montrer PCP $\leq$ PCPM :

- si on a un algorithme pour résoudre PCPM, on a un algorithme pour résoudre PCP.
- il suffit de résoudre m PCPM différents, selon le mot avec lequel on commence.

Montrons PCPM $\leq$ PCP :

Supposons donné un algorithme pour résoudre PCP.

On introduit une nouvelle lettre \$, et pour $a_1 \dots a_k \in A^+$, soient

$p(a_1, \dots a_k) = \$a_1 \dots \$a_k$ et $s(a_1 \dots a_k) = a_1\$ \dots a_k\$$.

Soit $(u_1, v_1), \dots, (u_m, v_m)$ une instance de PCPM.

Soient les $2m+1$ mots suivants :

$$x_i = p(u_i) \qquad y_i = s(v_i)$$

$$x_{m+i} = p(u_i)\$ \qquad y_{m+i} = s(v_i)$$

$$x_{2m+1} = p(u_1) \qquad y_{2m+1} = \$s(v_1)$$

le PCPM sur l'instance $((u_l, v_l))_{1 \leq l \leq m}$ a une solution si et seulement si le PCP sur l'instance $((x_l, y_l))_{1 \leq l \leq 2m+1}$ en a une.

On rappelle que $L_u = \{\langle M, w\rangle \mid M \text{ accepte } w\}$ est indécidable.

On montre $L_u \leq$ PCPM.

Etant donnée une machine de Turing M et un mot w, on construit une instance $((u_\ell, v_\ell))_{1\leq \ell\leq m}$ de PCPM telle que $\langle M, w\rangle \in L_u \iff$ PCP sur l'instance $((u_\ell, v_\ell))_{1\leq \ell\leq m}$ a une solution.

On peut supposer que

- le seul état d'arrêt de M est q_{OK}
- M déplace sa tête à chaque transition.

La réduction $L \leq$ PCPM :

idée : la seule solution sera la suite des configurations sur w.

La première partie est en retard d'une configuration

domino 1 : $(\#, \# q_0 w \#)$

dominos de copie : (a, a), $(\#, \#)$

dominos de transitions :

si $p \xrightarrow{a,b,\rightarrow} q \in \delta$, on met un domino (pa, bq).

si $p \xrightarrow{a,b,\leftarrow} q \in \delta$, on met un domino (xpa, qxb) pour tout $x \in \Gamma$.

si $p \xrightarrow{\square,b,\rightarrow} q \in \delta$, on met un domino $(p\#, bq\#)$

si $p \xrightarrow{\square,b,\leftarrow} q \in \delta$, on met un domino $(xp\#, qxb\#)$ pour tout $x \in \Gamma$

dominos de synchronisation en fin de calcul :

pour chaque $a, b \in \Gamma$: (aq_{OK}, q_{OK}), $(q_{OK}b, q_{OK})$, $(q_{OK}\#\#, \#)$

<u>Quelques autres problèmes indécidables :</u>

- Etant donné un jeu fini de tuiles carrées, avec condition de compatibilité entre côtés, déterminer si on peut paver le 1/4 de plan.
- Etant donnée une grammaire, déterminer si elle ambiguë.
- Etant donné un nombre fini de matrices 3×3 à coefficient entiers, déterminer si un produit permet d'annuler la composante $(3,2)$.
- Etant donnée une suite calculable d'entiers, déterminer si elle converge.

Révisions : vrai / faux

a) L'ensemble des graphes orientés finis est dénombrable.

→ Vrai

union dénombrable des graphes orientés à 1 sommets

union dénombrable des graphes orientés à 2 sommets

C'est l'union dénombrable d'ensembles finis.

b) Tout langage fini est décidable.

→ Vrai

Tout langage fini est un langage rationnel, il est donc reconnu par un automate fini.

c) Pour tous $K, L \subseteq \Sigma^*$ avec $\Sigma = \{0, 1\}$, si K se réduit à L alors $\Sigma^* \setminus K$ se réduit à $\Sigma^* \setminus L$.

→ Vrai

f est une réduction de K vers L ($f : \Sigma^* \to \Sigma^*$) si $x \in K \Leftrightarrow f(x) \in L$

$f : \Sigma^* \to \Sigma^*$ est une réduction de $\Sigma^* \setminus K$ vers $\Sigma^* \setminus L$ si $x \in \Sigma^* \setminus K \Leftrightarrow f(x) \in \Sigma^* \setminus L$

d) Tout langage sur un alphabet à une lettre est décidable.

→ Faux

L'ensemble de ces langages n'est pas dénombrable.

e) Etant donnés une machine de Turing M, un mot w et un entier k, on peut décider si M accepte w en au plus k pas de calcul.

→ Vrai

On arrête le calcul au bout de k pas et on regarde l'état de la machine. (travail plus long si M n'est pas déterministe)

f) Etant données deux machines de Turing M_1 et M_2, on peut décider si $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$

→ Faux

Si M_2 rejette tous les mots, $\mathcal{L}(M_2) = \emptyset$ et on ne peut pas décider si $\mathcal{L}(M_1) \subseteq \emptyset$.

g) Le langage des mots sur l'alphabet ASCII représentant un programme C syntaxiquement correct est décidable.

→ Vrai

C'est un analyseur syntaxique (ex: gcc)

h) Le complément de tout ensemble récursivement énumérable est aussi récursivement énumérable.

→ Faux

RE n'est pas fermé par complément.

Révision : problèmes indécidables

Les problèmes suivants sont indécidables :

P1 : Etant donnée une machine de Turing M, décider si elle accepte au moins sept mots.

→ preuve avec le théorème de Rice

P2 : Etant donnée une machine de Turing M, décider si elle s'arrête toujours avec écrit sur sa bande 2008.

→ le problème d'acceptation du mot vide est indécidable grâce au théorème de Rice.

donnée : M

question : $\varepsilon \in \mathcal{L}(M)$?

Accept-$\varepsilon \leq$ Arret-2008

réduction : $M \rightsquigarrow M_1 = f(M)$ tq $\varepsilon \in \mathcal{L}(M) \Leftrightarrow M_1$ s'arrête avec 2008 sur sa bande.

M_1 efface sa bande et lance M sur ε.

si M s'arrête en KO sur ε, on fait boucler M1

sinon si M s'arrête sur OK, M1 écrit □2008□ et s'arrête.

P3 : Etant données une MT M et un état p de M, décider si M utilise l'état p sur au moins une entrée.

→ même démonstration que P2.

P4 : Etant donnés une MT M et un état p de M, décider si M utilise l'état p sur toutes ses entrées.

→ même démonstration que P2.

PS: Etant donnée une MT M, décider si elle accepte n'importe quel mot représentant (en binaire) un entier premier.

→ preuve grâce au théorème de Rice.

VI Réductions : logique, graphes, et problèmes sur entiers.

Littéraux, clauses et formule CNF (Conjonctive Normal Forme)

Etant données des variables $x_1, x_2, \ldots$:

- un littéral est soit une variable x_i soit la négation d'une variable $\neg x_i$.
- une clause est une disjonction de littéraux.

  exemple : $x_1 \vee x_2 \vee \neg x_3 \vee x_5$.
- une 3-clause est une clause avec 3 littéraux différents

  exemple : $x_1 \vee \neg x_2 \vee x_3$
- une formule CNF est une conjonction de clauses.
- une formule 3-CNF est une conjonction de 3-clauses.

  exemple : $(x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_3)$

Le problème SAT

donnée : une formule CNF sur des variables $\{x_1, x_2, \ldots\}$

question : existe-t-il une assignation de chaque variable x_i par vrai ou faux qui rend la formule vraie ?

C'est un problème décidable car il y a un nombre fini de cas qu'on teste.

S'il y a n littéraux, il y a 2^n cas.

Pas de solution polynomiale.

Le problème 3-SAT

donnée : une formule 3-CNF sur les variables $\{x_1, x_2, \ldots\}$

question : existe-t-il une assignation de chaque variable x_i par vrai ou faux qui rend la formule vraie ?

Le problème 3-SAT est donc moins général que SAT.

Réduction de SAT vers 3-SAT

A toute instance φ de SAT, on associe une instance $\tilde{\varphi}$ de 3-SAT telle que φ est satisfaisable $\Leftrightarrow$ $\tilde{\varphi}$ est satisfaisable.

Remarque : on va construire $\tilde{\varphi}$ en temps polynomial par rapport à $|\varphi|$.

Si $\varphi = c_1 \wedge c_2 \wedge \dots \wedge c_k$ où chaque c_i est une clause, on construit $\tilde{\varphi} = \varphi_1 \wedge \dots \wedge \varphi_k$ où chaque φ_i est une conjonction de 3-clauses,

- φ_i utilise les variables c_i + éventuellement de nouvelles variables
- si une affectation des variables rend c_i vraie, on peut la compléter pour rendre φ_i vraie
- inversement, si une affectation des variables rend φ_i vraie, sa restriction aux variables c_i rend c_i vraie.

construction de φ_i :

- si $c_i = l_1$ (un littéral), on ajoute deux variables y_i, z_i
  $\varphi_i = (l_1 \vee y_i \vee z_i) \wedge (l_1 \vee \neg y_i \vee z_i) \wedge (l_1 \vee y_i \vee \neg z_i) \wedge (l_1 \vee \neg y_i \vee \neg z_i)$
- si $c_i = l_1 \vee l_2$, on ajoute une variable y_i
  $\varphi_i = (l_1 \vee l_2 \vee y_i) \wedge (l_1 \vee l_2 \vee \neg y_i)$
- si c_i est une 3-clauses : $\varphi_i = c_i$
- si $c_i = l_1 \vee \dots \vee l_k$ avec $k \geq 4$, on ajoute $k-3$ variables $t_{i,1}, \dots, t_{i,k-3}$
  $\varphi_i = (t_{i,1} \vee l_1 \vee l_2) \wedge (\neg t_{i,1} \vee l_3 \vee t_{i,2}) \wedge (\neg t_{i,2} \vee l_4 \vee t_{i,3}) \wedge \dots \wedge (\neg t_{i,k-3} \vee l_{k-1} \vee l_k)$

exemple :

$\varphi = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_4)$

alors la construction donne :

$\tilde{\varphi} = (t_{1,1} \vee x_1 \vee \neg x_2) \wedge (\neg t_{1,1} \vee x_3 \vee t_{1,2}) \wedge (\neg t_{1,2} \vee \neg x_4 \vee x_5)$
$\wedge (y_1 \vee x_1 \vee \neg x_2) \wedge (\neg y_1 \vee x_1 \vee \neg x_2)$
$\wedge (\neg x_1 \vee x_2 \vee x_4)$

On vérifie qu'avec la construction précédente :

- si une une affectation des variables rend chaque c_i vraie, on la

complète facilement pour rendre chaque φ_i vraie.

- inversement, si une affectation des variables rend chaque φ_i vraie, la restriction de cette affectation aux variables de c_i rend c_i vraie.

Donc c_i est satisfaisable $\Leftrightarrow$ φ_i est satisfaisable avec les mêmes valeurs pour les variables de c_i.

- Comme les variables ajoutées dans φ_i n'apparaissent que dans φ_i :

  φ est satisfaisable $\Leftrightarrow$ $\tilde{\varphi}$ est satisfaisable.

On a donc SAT $\leq$ 3-SAT

Inversement, comme 3-SAT est un problème moins général que SAT :

3-SAT $\leq$ SAT

Le problème 3-coloration

Donnée : un graphe non orienté G

question : existe-t-il une 3-coloration de G ?

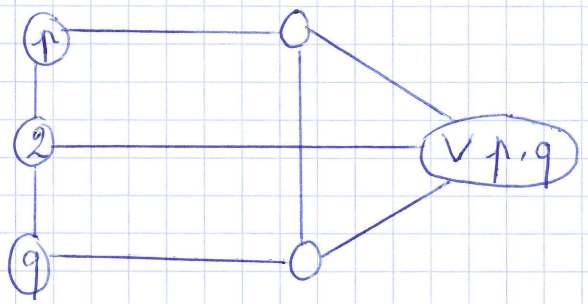
Réduction de 3-SAT vers 3-coloration

A toute instance φ de 3-SAT, on associe une instance G_φ de 3-coloration telle que φ est satisfaisable $\Leftrightarrow$ G_φ admet une 3-coloration.

On utilise des sous graphes (appelés gadgets) pour coder

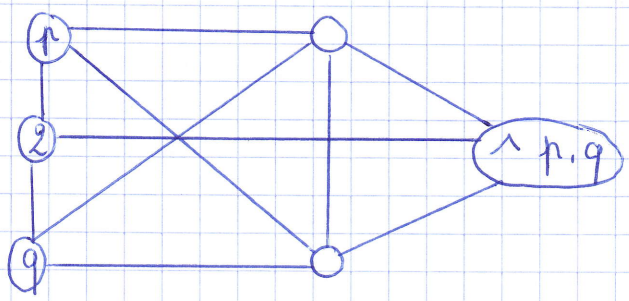
- les littéraux vrais dans une évaluation qui satisfait φ,
- les opérateurs logiques $\wedge$ et $\vee$.

- On utilise trois sommets particuliers 0, 1 et 2 reliés entre eux, qu'on peut supposer, quitte à renommer les couleurs, coloriés par 0, 1, 2.
- Pour chaque variable x_i, 2 sommets x_i et $\neg x_i$ reliés entre eux et à 2.
- Opérateurs : OU $(p \vee q)$ codé par



$\vee p,q$ coloriable par 1 si et seulement si p ou q sont coloriés 1.

ET $(p \vee q)$ codé par

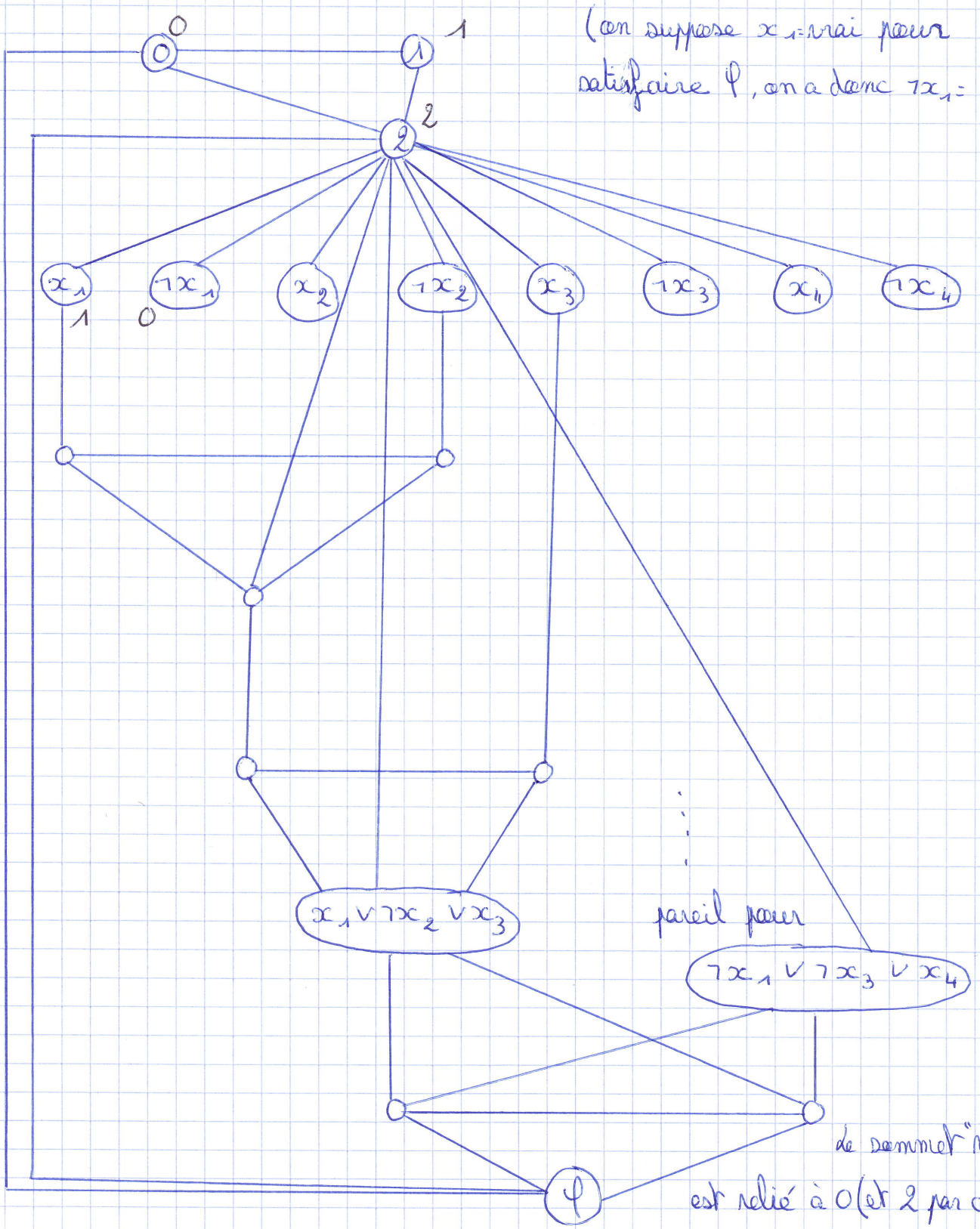


1 p,q colorable par 1 si et seulement si p ET q sont colorés 1.

exemple :

$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

(on suppose x_1 = vrai pour satisfaire φ, on a donc $\neg x_1$ = faux)



le sommet "résultat" est relié à 0 (et 2 par construction)

f est étiqueté par 1 car il est relié à 0 et à 2.
Le graphe est de taille linéaire par rapport à la formule.
On a donc, par transitivité SAT $\leq$ 3-coloration

Réduction de 3-coloration vers SAT

A chaque sommet s on associe 3 variables s_0, s_1, s_2.
idée : s_i est vrai code le fait que s a la couleur i.

$$f_G = \underbrace{\bigwedge_{s \in S} (s_0 \vee s_1 \vee s_2)}_{\text{conjonction de toutes les couleurs}} \wedge \underbrace{\bigwedge_{s \in S} (\neg s_0 \vee \neg s_1) \wedge (\neg s_1 \vee \neg s_2) \wedge (\neg s_0 \vee \neg s_2)}_{\text{on ne peut pas avoir 2 couleurs sur 1 sommet}}$$

$$\wedge \underbrace{\bigwedge_{\{s,t\} \in A} (\neg s_0 \vee \neg t_0) \wedge (\neg s_1 \vee \neg t_1) \wedge (\neg s_2 \vee \neg t_2)}_{\text{2 voisins n'ont pas la même couleur.}}$$

Clique et ensemble indépendant

Dans un graphe G non orienté, une clique pour G est un ensemble de sommets tous reliés 2 à 2.
Le problème Clique est le suivant :

donnée : un graphe G non orienté et un entier $K > 0$
question : existe-t-il une clique de G de taille K ?

Réduction de 3-SAT vers Clique

A toute instance φ de 3-SAT, on associe une instance G_φ, K_φ de Clique telle que φ est satisfaisable $\Leftrightarrow$ G_φ a une clique de taille K_φ et telle que on peut construire G_φ et K_φ en temps polynomial par rapport à $|\varphi|$.
Soit $\varphi = (l_0 \vee l_1 \vee l_2) \wedge \ldots \wedge (l_{3k-3} \vee l_{3k-2} \vee l_{3k-1})$.
Le graphe G_φ a $3k$ sommets : $l_1, l_2, \ldots, l_{3k}$.
Deux sommets l_i, l_k sont reliés si

- ils ne proviennent pas d'une même clause ($i/3 \neq k/3$)
- ils ne sont pas de la forme $l, \neg l$.

On choisit l'entier K_φ égal à k.
On vérifie que G_φ a une clique de taille K_φ ssi φ est satisfaisable.

VII Classes de complexité : P, NP, NP-complétude

Complexité en temps

Soit M une machine de Turing (s'arrêtant sur toute entrée).
Soit $step_M(w)$ le nombre maximal d'instructions exécutées sur l'entrée w jusqu'à l'arrêt de M.
Soit $f_M(n) = \max\{step_M(w) \mid |w| = n\}$
$f_M(n)$ mesure le temps passé par M sur une entrée de taille n, dans le pire des cas.

Les classes P et NP

$NP = \{\mathcal{L}(M) \mid \exists\, p$ polynôme tel que $f_M(n) \leq p(n)\}$
$P = \{\mathcal{L}(M) \mid \exists\, p$ polynôme tel que $f_M(n) \leq p(n)$ et M déterministe$\}$
On a bien sûr $P \subseteq NP$.
Pour montrer qu'un problème est NP, on utilise souvent
- une phase non déterministe, où la machine devine le résultat
- une phase déterministe, où elle vérifie si ce résultat est correct en temps polynomial.

exemple :
machine qui teste si un nombre n'est pas premier.
test de primalité de p :
pour tout $i \in [2, p-1]$
si $i \mid p$ retourner NON-PREMIER
retourner PREMIER
$\Rightarrow$ algorithme exponentiel (p divisions en temps polynomial)
($2^{O(k)}$ si k est la taille de p) $k = \lfloor \log_2(p+1) \rfloor$

A priori, ce n'est pas parce qu'un problème est dans NP que son complémentaire l'est aussi.

Les problèmes SAT, 3-SAT, 3-COLORATION et CLIQUE sont dans la classe NP.

→ L'appartenance à P ne dépend pas du nombre de bandes utilisées.
Théorème : Si L est décidé par une MT à k bandes déterministe en temps $f(n)$, alors L est décidé par une MT usuelle en $O(f^2(n))$.

→ On peut passer d'une MT non déterministe à une MT déterministe au prix d'une exponentielle en complexité en temps.
théorème : Si L est décidé par une MT non déterministe en temps $f(n)$, alors L est décidé par une MT usuelle en temps $2^{O(f(n))}$.

Réductions polynomiales :

Soient L_A et L_B deux langages de Σ^*.
Une réduction polynomiale de L_A vers L_B est une fonction $f : \Sigma^* \to \Sigma^*$ tq
- f est calculable par une MT déterministe en temps polynomial
- on a l'équivalence $x \in L_A \Leftrightarrow f(x) \in L_B$.

On note $L_A \leq_p L_B$ (L_A se réduit à L_B de façon polynomiale).
L'existe d'une réduction de L_A vers L_B assure que
- si $L_B \in P$ alors $L_A \in P$
- si $L_B \in NP$ alors $L_A \in NP$.

Les réductions SAT $\leq$ 3-SAT $\leq$ 3-COLORATION et 3-SAT $\leq$ CLIQUE sont bien polynomiales.

Le problème A est NP-complet si tout problème NP B se réduit à A en temps polynomial.

Problèmes NP-complets.

Le langage L (ou problème) est NP-complet si
- $L \in NP$
- pour tout langage $K \in NP$, on a $K \leq_p L$

Théorème de Cook-Levin

SAT est NP-complet.

Donc, d'après les réductions polynomiales vues précédemment, les problèmes 3-SAT, 3-COLORATION et CLIQUE sont NP-complets.

Principe de la preuve.

On part d'un langage quelconque de NP, décidé par une MT M.

On construit à partir de M et d'un mot d'entrée w une formule de taille polynomiale $f_{M,w}$ et telle que M accepte $w \Leftrightarrow f_{M,w}$ est satisfaisable.

Tout calcul de M sur w prend $p(n)$ étapes, où p est un polynôme et $n = |w|$.

Donc, tout calcul de M sur w utilise au plus $p(n)+1$ cases.

On introduit des variables avec leur signification intuitive

- $Q(i,k)$: vrai ssi l'état à l'instant i est q_k.
- $P(i,j)$: vrai ssi la position à l'instant i est j.
- $L(i,j,a)$: vrai ssi la lettre à l'instant i dans la case j est a.

Note : seulement un nombre polynomial de variables.

Grâce aux variables Q, P et L, on encode le calcul : à tout instant

- on se trouve dans un et un seul état,
- une et une seule case est lue,
- il y a une et une seule lettre dans chaque case.

On code également que

- au temps 0, la configuration est $q_0 w$
- au temps $p(n)$, la machine est dans l'état q_{OK}.
- entre le temps i et le temps $i+1$, on applique une transition.

Autre problème NP-complet : Somme d'entiers

donnée : des entiers $x_1, \dots, x_k > 0$ et un entier s

question : existe-t-il $1 \leq i_1 < i_2 < \dots < i_p \leq k$ tels que $x_{i_1} + x_{i_2} + \dots + x_{i_p} = s$?

C'est clairement dans NP : on devine $i_1, \dots, i_p$ et on teste.

On montre que c'est NP-complet par une réduction 3-SAT $\leq$ SOMME.

A partir de φ on construit $x_1, \dots, x_k, s$ tels que φ est satisfaisable $\Leftrightarrow$

$\exists\, 1 \leq i_1 < i_2 < \dots < i_p \leq k$ tels que $x_{i_1} + x_{i_2} + \dots + x_{i_p} = s$.

On écrit les entiers en base 10 :

$$\varphi = \underbrace{(y_1 \vee \neg y_2 \vee y_3)}_{c_1} \wedge \underbrace{(\neg y_1 \vee y_2 \vee y_4)}_{c_2}$$

positif

négatif

	y_1	y_2	y_3	y_4	c_1	c_2	
$x_1 = p(y_1)$	1	0	0	0	1	0	
$x_2 = m(y_1)$	1	0	0	0	0	1	
$x_3 = p(y_2)$	0	1	0	0	0	1	
$x_4 = m(y_2)$	0	1	0	0	1	0	
$x_5 = p(y_3)$	0	0	1	0	1	0	
$x_6 = m(y_3)$	0	0	1	0	0	0	(y_3 n'apparait négatif ni dans c_1 ni dans c_2)
$x_7 = p(y_4)$	0	0	0	1	0	1	
$x_8 = m(y_4)$	0	0	0	1	0	0	
s	1	1	1	1	3	3	(3 est le nombre de variables dans une clause)

on ajoute deux entiers identiques :

$x_9 = x(c_1)$	0	0	0	0	1	0
$x_{10} = x'(c_1)$	0	0	0	0	1	0
$x_{11} = x(c_2)$	0	0	0	0	0	1
$x_{12} = x'(c_2)$	0	0	0	0	0	1

Pour reconstituer les 1 de s, on choisit soit $p(y_i)$ soit $m(y_i)$ mais on ne peut pas avoir les deux.

⚠ Jamais de retenue dans les sommes car on est en base 10.

Le problème Partition

donnée : des entiers $x_1, \dots, x_k > 0$

question : existe-t-il $X \subseteq \{1, \dots, k\}$ tel que $\sum_{i \in X} x_i = \sum_{i \notin X} x_i$

C'est clairement dans NP : on devine X et on teste.

On cherche une partie des entiers telle que sa somme est égale à la somme des entiers qui restent.

Réduction Somme d'entiers vers Partition

Soit $x_1, \dots, x_k, s$ une instance de Somme d'entiers.

Soit $x = \sum x_i$.

On construit (en temps polynomial) l'instance $x_1, \dots, x_k, x-2s$ de Partition.

→ Si Somme d'entiers a une solution sur $x_1, \dots, x_k, s$,

alors Partition a une solution sur $x_1, \dots, x_k, x-2s$.

$\exists X \subseteq \{1, 2, \dots, k\}$ tel que $\sum_{i \in X} x_i = s$

donc $\sum_{i \in X} x_i + (x-2s) = x-s$ et $\sum_{i \notin X} x_i = x-s$.

→ Inversement, si Partition a une solution sur $x_1, \dots, x_k, x-2s$,

alors Somme d'entiers a une solution sur $x_1, \dots, x_k, s$.

⚠ la donnée de Partition doit être positive donc si $x-2s=0$, on ne le met pas.

VIII Fonctions récursives

Fonctions calculables

Une machine de Turing peut être utilisée pour calculer.

exemple : pour calculer une fonction $f : \mathbb{N}^k \to \mathbb{N}$, on peut utiliser $k+2$ bande

- k bandes d'entrée pour les arguments
- une bande de calcul
- une bande de sortie sur laquelle le résultat sera écrit.

Lorsqu'une machine est lancée avec l'argument w sur sa bande d'entrée, elle doit s'arrêter avec $f(w)$ sur sa bande de sortie.

On s'intéresse aux fonctions qu'il est possible de calculer par MT.

La classe des fonctions primitives récursives

Plus petite classe de fonctions $\mathbb{N}^k \to \mathbb{N}$

contenant (briques de base de Kleene)

- la fonction nulle de $\mathbb{N}^k \to \mathbb{N}$ pour tout $k \geq 0$

- la fonction Succ : $\mathbb{N} \to \mathbb{N}$ telle que $Succ(x) = x+1$
- la projection $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$ telle que $\pi_i^k(x_1, \dots, x_k) = x_i$

fermée par

- composition : si $f_1, \dots, f_k : \mathbb{N}^m \to \mathbb{N}$ et $g : \mathbb{N}^k \to \mathbb{N}$ sont primitives récursives, alors $h : \mathbb{N}^m \to \mathbb{N}$ l'est aussi, où h est définie par
  $h(x_1, \dots, x_k) = g(f_1(x_1, \dots, x_k), \dots, f_k(x_1, \dots, x_k))$.
- récursion : si $f : \mathbb{N}^m \to \mathbb{N}$ et $g : \mathbb{N}^{m+2} \to \mathbb{N}$ sont primitives récursives, alors $h : \mathbb{N}^{m+1} \to \mathbb{N}$ l'est aussi, où h est définie par
  $h(0, x_1, \dots, x_m) = f(x_1, \dots, x_m)$
  $h(x+1, x_1, \dots, x_m) = g(x, h(x, x_1, \dots, x_m), x_1, \dots, x_m)$.

<u>Comment programmer h quand on connait f et g ?</u>

```
int h (int x, int x_1, ..., int x_m)
{
    int res = f(x_1, ..., x_m);
    for (int i = 0; i < x; i++)
        res = g (i, res, x_1, ..., x_m);
    return res;
}
```

<u>Remarque</u> :

Les fonctions primitives récursives se calculent en C sans boucle while () donc on est sûr qu'on s'arrête.

<u>Exemples de fonctions primitives récursives</u> :

- la fonction prédécesseur : $h(0) = 0$ // fonction nulle
  $h(x) = x \dot{-} 1$ (max de 0 et $x-1$)
  par récursion : $h(x+1) = x = g(x, h(x))$ avec $g = \pi_1^2$
  Donc prédécesseur est primitive récursive.
- l'addition de deux entiers : $Add(0, y) = y$
  $Add(x+1, y) = Add(x, y) + 1$

on a $f = \pi_1^2$ et $g(x, Add(x,y), y) = Succ(Add(x,y)) = Succ(\pi_2^3(x, Add(x,y), y))$

Donc Add est primitive récursive.

- la multiplication de deux entiers : $Mult(0,y) = \pi_1^2(0,y) = 0$

$$Mult(x+1, y) = Add(Mult(x,y), y)$$

Donc Mult est primitive récursive.

- la différence tronquée : $x \dot{-} y \equiv \max\{0, x-y\}$

$Soustraire(y,x) = x \dot{-} y$

$Soustraire(0,x) = x$

$Soustraire(y+1, x) = Soustraire(y,x) - 1$

qui est la définition d'une fonction primitive récursive.

On peut permuter les arguments :

$$moins(x,y) = x \dot{-} y = Soustraire(\pi_2^2(x,y), \pi_1^2(x,y))$$

⚠ On ne fait la récurrence que sur un paramètre qui doit être le premier argument.

- le maximum de deux entiers : $\max(x,y) = x + (y \dot{-} x)$
- le minimum de deux entiers : $\min(x,y) = x + y \dot{-} \max(x,y)$

Propriétés des fonctions primitives récursives

La somme, le produit, la différence tronquée de fonctions primitives récursives est primitive récursive.

Prouvons que $\sum_{i=0}^{m} f(i, x_1, \dots, x_k)$ est primitive récursive :

$$g(0, x_1, \dots, x_k) = f(0, x_1, \dots, x_k)$$

$$g(m+1, x_1, \dots, x_k) = g(m, x_1, \dots, x_k) + f(m+1, x_1, \dots, x_k)$$

Si $f : \mathbb{N}^{p+1} \to \mathbb{N}$ est primitive récursive alors

$$g(m, x_1, \dots, x_p) = \sum_{i=0}^{m} f(i, x_1, \dots, x_p)$$

et $h(m, x_1, \dots, x_p) = \prod_{i=0}^{m} f(i, x_1, \dots, x_p)$ sont primitives récursives.

Fonctions définies par cas :

Une fonction dont les valeurs sont 0 ou 1 est appelée prédicat.
On interprète 0 comme faux et 1 comme vrai.
des opérateurs booléens classiques, appliqués à des prédicats primitifs recursifs fournissent des prédicats primitifs récursifs :

$P \wedge Q = \text{mult}(P, Q)$

$P \vee Q = \min(1, \text{somme}(P, Q))$

$\neg P(x) = 1 - x$.

Si $P(x, x_1, \dots, x_p)$ est un prédicat primitif récursif, il en est de même de

- $\forall x \leq m \; P(m, x_1, \dots, x_p) \equiv A_p(m, x_1, \dots, x_p) = \prod_{i=0}^{m} P(i, x_1, \dots, x_p)$
- $\exists x \leq m \; P(m, x_1, \dots, x_p) \equiv E_p(m, x_1, \dots, x_p) = 1 \dot{-} A_{1-P}(m, x_1, \dots, x_p)$

Si $P_1, \dots, P_k$ sont des prédicats primitifs récursifs et $f_1, \dots, f_{k+1}$ des fonctions primitives récursives, il en est de même de

$$g(x_1, \dots, x_m) = \begin{cases} f_1(x_1, \dots, x_m) & \text{si } P_1(x_1, \dots, x_m) \\ f_2(x_1, \dots, x_m) & \text{sinon et si } P_2(x_1, \dots, x_m) \\ \dots & \\ f_k(x_1, \dots, x_m) & \text{sinon et si } P_k(x_1, \dots, x_m) \\ f_{k+1}(x_1, \dots, x_m) & \text{sinon.} \end{cases}$$

Minimisation bornée :

Si $P(x, x_1, \dots, x_p)$ est un prédicat primitif récursif, la fonction

$$\mu_B P(m, x_1, \dots, x_p) \equiv \begin{cases} \min\{x \leq m \mid P(x, x_1, \dots, x_p)\} & \text{si un tel } x \text{ existe} \\ m + 1 & \text{sinon} \end{cases}$$

est primitive récursive.

On a $\mu_B P(m, x_1, \dots, x_p) = \sum_{k=0}^{m} \prod_{i=0}^{k} (1 \dot{-} P(i, x_1, \dots, x_p))$

On ne peut pas se passer de borner cette minimisation (sinon, la fonction n'est plus nécessairement primitive récursive).

Fonctions calculables, non primitives récursive :

Ackermann et Sudan ont trouvé en 1927-28 des fonctions

- non primitives récursives
- calculable mécaniquement

$$A(m,x) = \begin{cases} x+1 & \text{si } m=0 \\ A(m-1,1) & \text{si } m>0 \text{ et } x=0 \\ A(m-1, A(m,x-1)) & \text{si } m,x>0 \end{cases}$$

Une variante (D. Kozen) : $B(m,x) = B_m(x)$ où

$$B_0(x) = x+1$$

$$B_{m+1}(x) = \underbrace{B_m \circ \dots \circ B_m}_{x \text{ fois}}(x)$$

On vérifie que les récurrences terminent.

B croît trop vite pour être primitive récursive (idem pour A)

$B_0(x) = x+1$, $B_1(x) = 2x$, $B_2(x) \geqslant 2^x$ $(B_2(x) = 2^x . x)$

$B_3(x) \geqslant \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}_{x} = 2 \uparrow x$, $B_4(x) \geqslant 2 \uparrow\uparrow x$ $(B_4(2) = 2^{2048} \dots)$

où $2 \uparrow (x+1) = 2^{2 \uparrow x}$ et $2 \uparrow\uparrow (x+1) = 2 \uparrow (2 \uparrow\uparrow x)$ et, plus généralement,

$B_{m+2}(x) \geqslant 2 \uparrow \dots \uparrow_{m \text{ fois}} (x)$.

On retrouve la définition d'Ackermann.

<u>La fonction d'Ackermann n'est pas primitive récursive.</u>

On démontre que si $f : \mathbb{N}^p \to \mathbb{N}$ est primitive recursive alors il existe k tel que $f(x_1, x_2, \dots, x_p) < B_k(\max(x_1, \dots, x_p))$ (1)

Or, la fonction B ne vérifie pas 1, elle n'est donc pas primitive récursive (idem pour A).

En revanche, chaque B_k est primitive récursive.

<u>Bijection $\mathbb{N}^2 \to \mathbb{N}$ primitive recursive</u>

La bijection $c : \mathbb{N}^2 \to \mathbb{N}$ définie par $c(i,j) = \frac{(i+j)(i+j+1)}{2} + i$ est primitive récursive.

(preuve : montrer que $m \mapsto m \div 2$ est primitive recursive.

Les fonctions $d_1, d_2 : \mathbb{N} \to \mathbb{N}$ telles que $c^{-1} = (d_1, d_2)$ sont aussi primitives récursives. Par exemple, pour d_1, utiliser

$$d_1(x+1) = \begin{cases} 0 & \text{si } \exists y \leq x, x = \frac{y(y+1)}{2} \\ d_1(x)+1 & \text{sinon} \end{cases}$$

Récurrence simultanée

On note $\vec{x} = x_1, \dots, x_p$.

Propriété :

Si $f_1, \dots, f_k : \mathbb{N}^p \to \mathbb{N}$ et $g_1, \dots, g_k : \mathbb{N}^{k+p+1} \to \mathbb{N}$ sont des fonctions primitives récursives, alors les fonctions $h_1, \dots, h_k : \mathbb{N}^{p+1} \to \mathbb{N}$ définies comme suit le sont aussi

$$h_i(0, \vec{x}) = f_i(\vec{x})$$
$$h_i(m+1, \vec{x}) = g_i(m, h_1(m, \vec{x}), \dots, h_k(m, \vec{x}), \vec{x}).$$

Preuve :

Pour $k = 2$, on utilise le codage $c : \mathbb{N}^2 \to \mathbb{N}$ et ses décodages d_1 et d_2 pour exprimer $h = c(h_1, h_2)$ à l'aide du schéma de récurrence.

$$h(m, \vec{x}) = c(h_1(m, \vec{x}), h_2(m, \vec{x}))$$
$$h_1(m, \vec{x}) = d_1(h(m, \vec{x}))$$
$$h_2(m, \vec{x}) = d_2(h(m, \vec{x}))$$
$$h(0, \vec{x}) = c(h_1(\vec{x}), h_2(\vec{x}))$$
$$\begin{aligned} h(m+1, \vec{x}) &= c(h_1(m+1, \vec{x}), h_2(m+1, \vec{x})) \\ &= c(g_1(m, d_1(h(m, \vec{x})), d_2(h(m, \vec{x})), \vec{x})) \\ &= G(m, h(m, \vec{x}), \vec{x}) \end{aligned}$$
$$(i, j, k) \mapsto c(c(i, j), k)$$

Fonctions récursives

Soit $P(z, \vec{x}) : \mathbb{N}^{p+1} \to \{0, 1\}$ un prédicat non nécessairement total, (c'est-à-dire non nécessairement défini pour tout $\mathbb{N}^{p+1}$)

On écrit $P(z, \vec{x}) = a$ si P est défini sur $(z, \vec{x})$ et vaut a.

$\mathcal{E}_P(\vec{x}) = \{ y \in \mathbb{N} \mid P(y, \vec{x}) = 1 \wedge \bigwedge_{z<y} P(z, \vec{x}) = 0 \}$

$\mathcal{E}_P(\vec{x})$ est soit un singleton (si $P(k, \vec{x}) = 1$ et $P(i, \vec{x}) = 0 \ \forall i < k$, $\mathcal{E}_P(\vec{x}) = \{k\}$), soit l'ensemble vide (si P n'est pas défini pour un y).

Soit $\mu P : \mathbb{N}^n \to \mathbb{N}$,
$$\mu P(\vec{x}) = \begin{cases} \min \mathcal{E}_P(\vec{x}) & \text{si } \mathcal{E}_P(\vec{x}) \neq \emptyset \\ \text{indéfini} & \text{sinon.} \end{cases}$$

La fonction μP peut ne pas être totale, même si P l'était.

⚠ Malgré les définitions similaires, la minimisation (générale) permet de construire des fonctions non primitives récursives, contrairement à la minimisation bornée.

Calcul des fonctions récursives

Les fonctions primitives récursives sont calculables par programme.

Si $P(y, x)$ est un prédicat récursif, $\mu P(x)$ se "calcule" par

```
int muP (int x)
{
    y=0;
    while (P(y,x)==0)
        y++;
    return y;
}
```

Le calcul peut ne pas s'arrêter
- soit parce que l'un des appels $P(y, x)$ ne termine pas,
- soit parce que $P(y, x)$ vaut toujours 0.

La classe des fonctions récursives

Plus petite classe de fonction $\mathbb{N}^k \to \mathbb{N}$
contenant la fonction nulle, la fonction successeur et les projections
fermée par composition, recursion, minimisation : si P est un prédicat recursif, μP l'est aussi.

Fonctions récursives et fonctions MT-calculables

Théorèmes :

- Les fonctions récursives sont exactement les fonctions de $\mathbb{N}^k$ dans $\mathbb{N}$ pour $k > 0$, qui sont calculables par machine de Turing.
- Conséquence : il existe une fonction récursive, définie sur $\mathbb{N}$ tout entier, qui n'est pas primitive récursive.
- Il existe des fonctions non récursives.

Montrons qu'un prédicat est primitif récursif :

$$P(x,y) = \begin{cases} 1 & \text{si } x < y \\ 0 & \text{sinon} \end{cases}$$

$x < y \Leftrightarrow y \dot{-} x \neq 0$

vaut_zero $(0) = 1$

vaut_zero $(z+1) = 0$

$P(x,y) = \neg$ vaut_zero $(y \dot{-} x)$

Soient f_1, f_2, P_1 primitifs récursifs

$$g(x) = \begin{cases} f_1(x) & \text{si } P_1(x) \\ f_2(x) & \text{sinon} \end{cases}$$

$g(x) = f_1(x)\, P_1(x) + (1 - P_1(x))\, f_2(x)$

de façon générale :

$g = f_1 P_1 + (1 - P_1)[f_2 P_2 + (1 - P_2)[f_3 P_3 + \dots]]$

minimisation bornée :

$x_1, \dots, x_p$ fixés

$f(x) = P(x, x_1, \dots, x_p)$

$f(0)$	$f(1)$	$f(2)$	$f(3)$	...	$f(m)$
0	0	1	0		1

$g(m) = \mu_B P(m, x_1, \dots, x_p)$

$g(0)$	$g(1)$	$g(2)$	$g(3)$	...	$g(m)$
1	2	2	2		2

$$h(k) = \prod_{i=0}^{k} 1 - P(i, x_1, \dots, x_p)$$

$h(0)$	$h(1)$	$h(2)$	$h(3)$	...	$h(m)$
1	1	0	0		0

$$s(m) = \sum_{k=0}^{m} h(k) = g(m)$$

$s(0)$	$s(1)$	$s(2)$	$s(3)$	...	$s(m)$
1	2	2	2		2

Feuille supplémentaire

NOM .....

Prénom Modèles de calcul

N° 1

Bijections calculables :

Une fonction est calculable s'il existe une machine de Turing qui calcul g, $g(p) = T(m, p)$ $\forall p \in \mathbb{N}$ (les valeurs de g seront la m-ième ligne).

Un ensemble est dénombrable s'il est en bijection avec $\mathbb{N}$.

Un ensemble fini ou dénombrable est dit au plus dénombrable.

Machines de Turing

$E \subseteq \mathbb{N}$ est décidable = récursif

$\Leftrightarrow$ il existe une machine de Turing s'arrêtant sur toute entrée.

$\Leftrightarrow$ il existe un algorithme implémentant $d(m) = \begin{cases} 1 \text{ si } m \in E \\ 0 \text{ sinon.} \end{cases}$

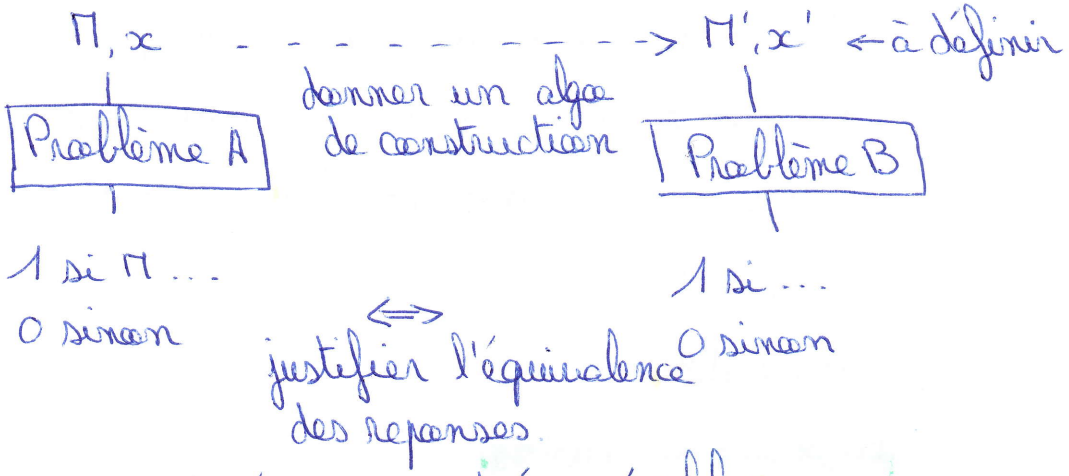
Modèles alternatifs aux machines de Turing et indécidabilité

Une machine RAM a des registres qui contiennent des entiers non limités $\Leftrightarrow$ infinités de bandes infinies.

- load # <m> : met l'entier m dans l'accumulateur
- load <m> : met l'entier contenu dans R_m dans l'accumulateur
- load(<m>) : met l'entier contenu dans R_{R_m} dans l'accumulateur
- store <m> : met la valeur contenue dans A dans R_m
- store(<m>) : met la valeur contenue dans A dans R_{R_m}
- incr : incrémente de 1 la valeur dans l'accumulateur
- decr : décrémente de 1 la valeur dans l'accumulateur
- jump <m> : saut inconditionnel à l'instruction I_m.

- jz<m> : saut à I_m si l'accumulateur est nul.
- stop : fin du programme.
- read : lit la valeur courante de la bande d'entrée et la met dans l'accumulateur.
- write : écrit sur la bande de sortie la valeur de A.

Réduction du problème A au problème B : $A \leq B$



$E \subseteq \mathbb{N}$ est récursivement énumérable
⟺ il existe une machine de Turing M telle que $\mathcal{L}(M)$ est l'ensemble des écritures de E
⟺ il existe un algorithme de semi-décision pour $d(m) = \begin{cases} 1 & \text{si } m \in E \\ \uparrow & \text{sinon} \end{cases}$

$R \subseteq RE$

Diagonalisations - Théorème de Rice

```
int test (int x)
{   if (termine (test, x)) // teste une propriété sur la fonction
        while (true); // code ne vérifiant pas la propriété
    return 0;
}
```

Si la propriété sur le langage n'est pas triviale alors la question est indécidable.

machine reconnaissant un langage
⟺ fonction retournant 0 ou 1.
machine qui calcule un résultat
⟺ fonction affichant le résultat à l'écran.

Feuille supplémentaire

NOM

Prénom Modèles de calcul

N° 2

Classes P et NP

	Problème P (polynomial)	Problème NP (polynomial non déterministe)	Problème NP-complet NP-difficile
définition	peut être résolu par une MT déterministe travaillant en temps polynomial par rapport à la taille du mot d'entrée	peut être résolu par une MT non déterministe travaillant en temps polynomial par rapport à la taille du mot d'entrée	tout problème NP peut s'y réduire NP-complet = NP-difficile + NP
caractérisation informelle	on peut le résoudre en temps polynomial par rapport à la taille des données	il existe un système de certificats + vérificateur polynomial	NP ≤ NP-difficile
techniques de preuves	- donner l'algo - réduire polynomialement à un autre problème P.	- donner un système de certificat (y) + vérificateur V dans P tel que $Q(x) \Leftrightarrow V(x,y)$ - réduire à un autre problème P	réduire polynomialement à ce problème, un problème déjà connu comme NP-complet

$P \subseteq NP \subseteq$ NP-difficile $\subseteq$ Décidable.

Fonctions récursives

classe des fonctions $\mathbb{N}^k \to \mathbb{N}$ primitives récursives

contenant la fonction nulle

la fonction successeur

la projection

fermée par composition

récursion.

classe des fonctions $\mathbb{N}^k \to \mathbb{N}$ récursives
contenant la fonction nulle,
la fonction successeur,
les projections
fermée par composition,
récursion,
minimisation.