

THEORETICAL ASPECTS OF COMPUTER SCIENCE

STEFAN BRUDA, BISHOP'S UNIVERSITY, WINTER 2007

Prise de notes et traduction par Philippe Giabbanelli

I. Rappels : Ensembles, Relations, Preuves, Complexité

1) Rappels sur les ensembles

Un ensemble (*set*) est une collection d'éléments distincts. Pour un ensemble S , $\mathcal{P}(S)$ est appelé power set, et contient tous les sous-ensembles que l'on peut former avec S , plus l'ensemble vide. Par exemple :

$$S = \{a, b, c\}$$

$$\mathcal{P}(S) = \{ \{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\} \}$$

Si S est fini avec $|S| = n$, alors $|\mathcal{P}(S)| = 2^n$, d'où une autre notation 2^S pour le powerset.

Une partition de S est faite avec quelques uns des sous-ensembles que l'on peut former avec S . Lorsqu'on parle des partitions de S , ce sont toutes les partitions telles que leurs unions donne couvrent S et que leur intersection est vide. Par exemple :

$$S = \{1,2,3\}$$

$$P_1 = \{ \{1\}, \{2\}, \{3\} \}$$

$$P_2 = \{ \{1,2\}, \{3\} \}$$

$$P_3 = \{ \{1,3\}, \{2\} \}$$

$$P_4 = \{ \{1\}, \{2,3\} \}$$

$$P_5 = \{ \{1,2,3\} \}$$

Le nombre de Bell B_n est le nombre de partitions différentes que l'on peut former avec un ensemble à n éléments ; les premiers nombres sont $B_0 = 1, B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15, B_5 = 52, B_6 = 203$. On a :

$$B_{n+1} = \sum_{k=0}^n \binom{n}{k} B_k$$

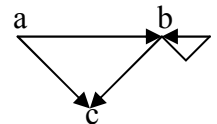
2) Relations

Une relation entre n ensembles est un ensemble de n -uplets, représentant des associations. Nous allons nous intéresser aux relations binaires. $\mathcal{R} \in Z^{A \times A}$ est une relation d'un ensemble dans lui-même. Exemple :

$$\mathcal{R} = \{ (a,b), (a,c), (b,b), (b,c) \} \in Z^{\{a,b,c\} \times \{a,b,c\}}$$

$\langle a,b,c \rangle$ est un chemin (*path*) car (a,b) et (b,c) sont dans la relation.

$\langle a,b,b,c \rangle$ est un autre chemin.



Les relations binaires ont des propriétés :

- [1] Réflexive ssi $(a,a) \in \mathcal{R}$ pour tout $a \in A$. Correspond à avoir des boucles pour tout sommet.
- [2] Transitif ssi $(a,b) \in \mathcal{R} \wedge (b,c) \in \mathcal{R} \rightarrow (a,c) \in \mathcal{R}$ pour tout $a, b, c \in A$.
- [3] Symétrique ssi $(a,b) \in \mathcal{R} \rightarrow (b,a) \in \mathcal{R}$ pour tout $a, b \in A$.
- [4] Antisymétrique ssi $(a,b) \in \mathcal{R} \wedge (b,a) \in \mathcal{R} \rightarrow a = b$ pour tout $a, b \in A$.

Réflexif et Transitif est preorder. Réflexif, Transitif, Symétrique est une relation d'équivalence (qui induit une partition sur l'ensemble). Réflexif, Antisymétrique, Transitif (RAT) est un ordre partiel.

On note par $[a] = \{b \in A \mid (a,b) \in \mathcal{R}\}$ une classe d'équivalence. On note que $[a] \cap [b] = \emptyset$ et $\bigcup_{a \in A} [a] = A$. Ainsi, les classes d'équivalences forment une partition de l'ensemble A .

3) Vocabulaire

On a des bijections naturelles $A, B, C \mapsto (A \times B) \times C$; par exemple, $f((a,b),c) = ((a,b),c)$.

On peut d'*equinumerosity* lorsque deux ensembles d'une relation sont de même cardinalité.

La preuve par récurrence (*induction proof*) s'applique uniquement aux ensembles dénombrables. Une preuve de bas niveau sur les ensembles est le principe des tiroirs et des chaussettes (*pigeonhole*).

4) Preuves par diagonalisation

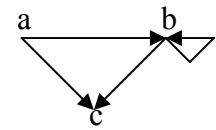
Soit $\mathcal{R} \in \mathcal{Z}^{A \times A}$. On a $D = \{a \in A : (a,a) \notin \mathcal{R}\}$, qui est l'ensemble diagonal de \mathcal{R} (*diagonal set*).

On a $\mathcal{R}_a = \{b \in A : (a,b) \in \mathcal{R}\}$, pour tout a dans l'ensemble A. On remarque que pour tout a, $D \neq \mathcal{R}_a$.

Prenons l'exemple $\mathcal{R} = \{(a,b), (a,c), (b,b), (b,c)\} \in \mathcal{Z}^{\{a,b,c\} \times \{a,b,c\}}$

Dans la représentation de la relation par matrice d'adjacence, on peut voir les \mathcal{R}_a comme étant les lignes. Le diagonal set est l'opposé de la diagonale de la matrice d'adjacence. Ici, on a 101.

	a	b	c
a	0	1	1
b	0	1	1
c	0	0	0



Application : $2^{\mathbb{N}}$ est indénombrable.

Supposons que $2^{\mathbb{N}}$ est dénombrable. Il y aurait donc une façon d'énumérer ses éléments : $2^{\mathbb{N}} = \{R_0, \dots\}$.

Rappelons les définitions : $\mathcal{R} = \{(i,j) : j \in \mathcal{R}_i\}$ et $D = \{n : (n,n) \notin \mathcal{R}\}$.

Est-ce que $D \in 2^{\mathbb{N}}$? Il devrait, puisque c'est une suite de nombres. Donc il existe un k tel que $D = \mathcal{R}_k$.

On a alors deux possibilités :

- $k \in D$ donc $k \notin \mathcal{R}_k$ en regardant les définitions. Dans ce cas là, $D \neq \mathcal{R}_k$: contradiction.
- $k \notin D$ donc $k \in \mathcal{R}_k$: autre contradiction.

On est parti du principe que $2^{\mathbb{N}}$ est dénombrable et on arrive uniquement à des contradictions.

- Remarques :
- Si A et B sont dénombrables, alors $A \times B$ et $A \cup B$ le sont également.
 - Pour prouver qu'il n'y a pas d'isomorphisme (i.e. bijection), on utilise la diagonalisation.
 - Si on a un sous-ensemble B indénombrable de A, alors A est indénombrable.

Application : $[0, 1[$ est indénombrable.

Supposons que $[0, 1[$ est dénombrable. On fait toujours une preuve par contradiction.

Il y aurait donc une façon d'énumérer ses éléments : $[0, 1[= \{R_1, R_2, \dots\}$. Tout nombre étant précédé par 0, on peut les représenter uniquement avec la partie après la virgule, sous forme binaire. D'où :

\mathcal{R}_1	b_{11}	b_{12}	b_{13}	b_{14}
\mathcal{R}_2	b_{21}	b_{22}	b_{23}	b_{24}
\mathcal{R}_3	b_{31}	b_{32}	b_{33}	b_{34}

On a alors $D = \overline{b_{11}} \cdot \overline{b_{22}} \cdot \overline{b_{33}} \cdot \overline{b_{44}}$. On peut utiliser la même notation $\overline{\quad}$ en base 10, avec le sens « tout ce qui peut-être dans l'ensemble et n'est pas le symbole barré ».

Il doit y avoir un K tel que $\mathcal{R}_K = D$. Or $D = \mathcal{R}_K \rightarrow b_{KK} = \overline{b_{KK}}$, ce qui est impossible.

5) Fermeture (closure)

Soit $D, \mathcal{R} \subseteq D^n, n > 0$. C'est une relation d'arité n.

Soit $B \subseteq D$. On dit que B est clôt (ou stable) par \mathcal{R} ssi pour tout $a_1, a_2, \dots, a_{n-1} \in B$, pour de quelconques $(a_1, a_2, \dots, a_{n-1}, a_n) \in \mathcal{R}$ alors $a_n \in B$.

Par exemple, est-ce que \mathbb{N} est stable/clôt par addition ou multiplication ? Oui. Mais pas par soustraction !

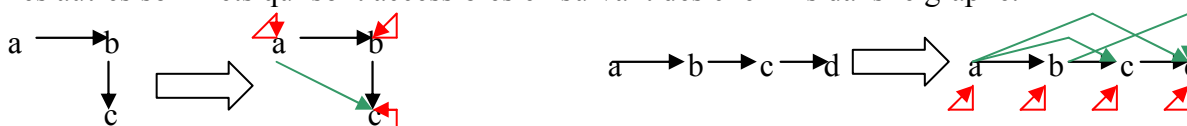
Soit A un ensemble clôt sous les relations $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n$. Soit $D \subseteq A$.

Il existe un ensemble B tel que $D \subseteq B$, B est minimal et clôt sous les relations $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n$.

'minimal' signifie que si l'on enlève un élément de l'ensemble, alors il n'est plus clôt : $B \supsetneq D$ et B est clôt. Cela revient à dire qu'il existe un ensemble minimal stable pour les opérations.

Application : déterminer la fermeture réflexive et transitive d'un ensemble.

La fermeture **réflexive** consiste à avoir des boucles sur tout sommet. La **transitive** relie à tout sommet S les autres sommets qui sont accessibles en suivant des chemins dans le graphe.



6) Analyse de complexité : Algorithme de construction d'une fermeture réflexive et transitive

1. $\mathcal{R} \subseteq 2^{A^*A}$
2. $C \leftarrow \{(a,a) : a \in A\}$
3. From $i \leftarrow 2$ to $|A|$ do
4. For each $b_1 b_2 \dots b_i$ with $b_k \in A, 0 < k \leq i$, do
5. If $b_1 b_2 \dots b_i$ is a path in \mathcal{R}
6. Then $C \leftarrow C \cup \{b_1, b_i\}$

Avec (2) on construit les fermetures réflexives en créant toutes les boucles. Ensuite, pour chaque chemin existant, on rajoute ses extrémités. Le problème est la recherche parmi toutes les séquences de i éléments de A , ce qui nous amène à une complexité en $\Omega(n^n)$, impraticable.

Illustration d'un passage du Repeat :
 $C = \{(a,b), (b,c), (a,a), (b,b), (c,c)\}$
(i.e. schéma en bas à gauche page préc.)
 $a_1 = a ; a_2 = b ; a_3 = c \rightarrow (a, c) \notin C$
 $\rightarrow C' = \{(a,c)\}$

On a n choix pour le premier élément, $n-1$ pour le 2nd et $n-2$ pour le dernier. (4) est donc en $O(n^3)$. L'algorithme est en $O(n^5)$.

1. $C \leftarrow \mathcal{R} \cup \{(a,a) : a \in A\}$
2. Repeat
3. $C' \leftarrow \emptyset$
4. For each $a_1, a_2, a_3 \in A, a_1 \neq a_2 \neq a_3$, do
5. If $(a_1, a_2) \in C \wedge (a_2, a_3) \in C \wedge (a_1, a_3) \notin C$
6. $C' \leftarrow C' \cup \{(a_1, a_3)\}$
7. $C \leftarrow C \cup C'$
8. Until $C' = \emptyset$

On peut faire encore plus rapide, en multipliant la matrice d'adjacence. Le 'for each' peut se faire en regardant dans la matrice, d'où un n^2 au lieu de n^3 . On peut aussi calculer directement la fermeture I^n , ce qui se fait en $\log n$ multiplications de matrices ; une multiplication est en n^3 d'où du $n^3 \cdot \log(n)$. Il y a encore des améliorations par des algorithmes de multiplication rapide, soit du $n^{2+\epsilon} \cdot \log(n)$.

II. Langages

1) Principes généraux

Un alphabet Σ est un quelconque ensemble fini. Une phrase sur cet alphabet est un ensemble de symboles. Un langage est un ensemble de phrases, noté Σ^* : l'ensemble de tous les mots sur l'alphabet Σ . Si A et B sont deux alphabets, $A \times B$ est toujours un ensemble fini donc c'est également un alphabet. Si $L \subseteq \Sigma^*$, alors L est un langage. La chaîne vide se note ϵ (voire λ , ou e dans le livre de Papadimitriou). Un langage peut contenir ϵ , ou même être vide. Remarquons que $\epsilon \notin \emptyset$. Les opérations qui peuvent être faites sur les chaînes sont : la longueur $||$, la concaténation $^\circ$ ou \cdot , la sous-chaîne (*substring prefix/suffix*). A partir de là, on peut définir de nouvelles opérations comme la puissance ou l'inverse (*reversal*) :

$w^n = w \cdot w \cdot \dots \cdot w, n$ times, tel que $w^0 = \epsilon$, pour tout $w \in \Sigma^*$. Définition inductive :

$$\forall w \in \Sigma^*, \begin{cases} w^0 = \epsilon \\ w^{n+1} = w \cdot w^n \end{cases}$$

L'inverse $(abc)^R$ est cba . La définition inductive est :

$$\forall w \in \Sigma^*, a \in \Sigma, \begin{cases} \epsilon^R = \epsilon \rightarrow |w|=0 \\ (aw)^R = w^R \cdot a \rightarrow |w|=n \end{cases}$$

On a $(w^R)^i = (w^i)^R$. Comme l'ensemble a été construit par induction, il est conseillé de prouver par récurrence. De même, on peut prouver $(wu)^R = u^R w^R$ par induction sur la longueur de w .

$\mathcal{P}(\Sigma^*)$ est l'ensemble de tous les langages sur un alphabet. Or, $|\mathcal{P}(\Sigma^*)| = 2^{|\Sigma^*|}$, donc indénombrable. On a $|0| = |\mathbb{N}|$ qui est dénombrable, et par l'ordre lexicographique on peut voir que certains langages sont également dénombrables : on construit la séquence $\epsilon a_1 \dots a_n a_1 a_1 a_1 a_2 \dots a_1 a_n a_2 a_1 \dots a_n a_n$.

1 2 n

Ceci est une bijection avec les entiers naturels. Il y a donc certains langages indescriptibles (*pigeonhole*).

On peut définir l'union ou l'intersection de langages, ainsi que :

- L'opposé $L_1 = \Sigma^* - L_1$ pour tout L_1 basé sur Σ
- La concaténation $L_1L_2 = \{w_1w_2 : w_1 \in L_1, w_2 \in L_2\}$
- Kleene Star, ou Kleene Closure $L^* = \{w_1...w_n : \text{pour tout } i > n, w_i \in L, n \geq 0\}$
Il s'agit de la fermeture de $L \cup \{\epsilon\}$ par la concaténation.
 $L^+ = LL^*$ ne contient pas $\epsilon \leftrightarrow L$ ne contient pas ϵ .

2) Langages réguliers

Un langage régulier REG est la fermeture de $\{a : a \in \Sigma\} \cup \{\emptyset\}$ par union, concaténation, étoile de Kleene.

La définition par induction de langages régulier est :

1. $\emptyset, \{a\} \in \text{REG}$
2. $\alpha, \beta \in \text{REG} \rightarrow \alpha\beta \in \text{REG}$
3. $\alpha, \beta \in \text{REG} \rightarrow \alpha \cup \beta \in \text{REG}$
4. $\alpha \in \text{REG} \rightarrow \alpha^* \in \text{REG}$
5. Rien d'autre n'est dans REG (implicite...)

Exemple : $\{a\}^*(\{b\}\{a\}^*)^*$, noté $a^*(ba^*)^*$. La définition d'une expression régulière est basée sur le même modèle : \emptyset et a sont des expressions régulières ; si α et β sont des expressions régulières, alors il en va de même pour $\alpha\beta, \alpha \cup \beta$, et α^* .

On note par \mathcal{L} la bijection entre une expression régulière et le langage régulier qu'elle génère.

Exemple : $\mathcal{L} = \{w \in \{0,1\}^* : |w|_{\{1\}} \equiv 0 [3]\} \leftrightarrow \mathcal{L}((0^*10^*10^*10^*)^*) \cup \mathcal{L}(0^*)$

On note par $|w|_{\{1\}}$ la longueur de la chaîne w sur l'ensemble $\{1\}$, i.e. le nombre de 1 dans la chaîne.

$(0^*10^*10^*10^*)^*$ est le générateur du langage. Les automates finis sont les accepteurs pour ces générateurs.

Soit $\mathcal{L}((a \cup b)^*) = \{a,b\}^*$ et $\mathcal{L}(a^*(ba^*)^*) = \{a,b\}^*$. On veut montrer l'égalité, c'est à dire l'inclusion dans les deux sens. On va ici démontrer que $\mathcal{L}(a^*(ba^*)^*) \subseteq \{a,b\}^*$. Soit $w \in \{a,b\}^* : w_n = a^{D_0}b^{E_1}a^{D_1}b^{E_2}a^{D_2}...b^{E_n}a^{D_n}$ avec $d_i, e_i \in \mathbb{N}, 0 < i \leq n$. On prouve qu'il s'agit du même ensemble par une induction sur n :

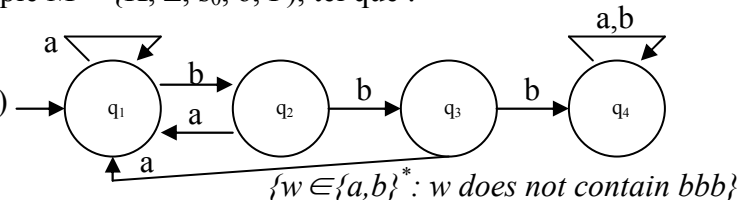
- ♦ $n = 0 : w = a^{D_0} \in \mathcal{L}(a^*) \subseteq \mathcal{L}(a^*(ba^*)^*)$
- ♦ $w_n \in \mathcal{L}(a^*(ba^*)^*)$. On veut prouver que $w_{n+1} = w_n b^{E_{n+1}} a^{D_{n+1}} \in \mathcal{L}(a^*(ba^*)^*)$.
- ♦ Considérons $E_{n+1} = 1$. On a $w_{n+1} = w_n b a^{D_{n+1}} \in \mathcal{L}(a^*(ba^*)^*) \mathcal{L}(ba^*) = \mathcal{L}(a^*(ba^*)^*(ba^*)) \subseteq \mathcal{L}(a^*(ba^*)^*)$

Pour $E_{n+1} > 1$, on a $w_n b^{E_{n+1}-1} a^{D_{n+1}}$. On a $w_n \in \mathcal{L}(a^*(ba^*)^*)$, $b^{E_{n+1}-1} \in \mathcal{L}(b^*) \subseteq \mathcal{L}((ba^*)^*)$, et $ba^{D_{n+1}} \in \mathcal{L}(ba^*) \subseteq \mathcal{L}((ba^*)^*)$. D'où $w_{n+1} \in \mathcal{L}(a^*(ba^*)^*) \mathcal{L}((ba^*)^*) \mathcal{L}((ba^*)^*) = \mathcal{L}(a^*(ba^*)^*(ba^*)^*(ba^*)) \subseteq \mathcal{L}(a^*(ba^*)^*)$.

3) Automates Déterministes Finis (DFA : Deterministic Finite Automaton)

Un automate déterministe fini est représenté par le tuple $M = \{K, \Sigma, s_0, \delta, F\}$, tel que :

- K est l'ensemble fini d'états
- Σ est l'alphabet d'entrée
- s_0 est l'état unique de démarrage (*initial state*)
- δ est une fonction de transition
- F est l'ensemble des états finals



Une configuration $K \times \Sigma^*$ est l'état courant avec la chaîne restant à traiter. Par exemple, $(q_1, 01101)$ quand on démarre avec la chaîne 01101 sur l'automate précédent. Les transitions effectuées par l'automate M sont représentées par $(q_1, 01101) \vdash_M (q_2, 1101)$, etc. On note par \vdash_M^* la fermeture réflexive et transitive de \vdash_M , dite *yields*.

M accepte une entrée $w \leftrightarrow$ il existe $f \in F$ tel que $(s_0, w) \vdash_M^* (f, \epsilon)$
 \leftrightarrow l'automate se trouve dans un état final en ayant consommé toute la chaîne
 $\mathcal{L}(M) = \{w \in \Sigma^* : M \text{ accepte l'entrée } w\} = \{w \in \Sigma^* : \text{il existe } f \in F : (s_0, w) \vdash_M^* (f, \epsilon)\}$.

4) Automates Non-Déterministes Finis (NFA)

Dans un automate déterministe, δ est une fonction de transition. Ici, on utilise une relation de transition Δ telle que $\Delta \subseteq K \times (\Sigma \cup \{\epsilon\}) \times K$, et on a $(q, aw) \vdash_M (q', w)$ ssi $(q, a, q') \in \Delta$. Conséquences immédiates :

- Il peut y avoir des ϵ -transitions, c'est-à-dire un changement d'état qui ne consomme pas d'entrée.
- Comme c'est une relation, on peut se projeter sur plusieurs éléments à la fois (\neq fonction !).

Il n'y a plus nécessairement qu'un chemin de l'état initial à l'état final. Cependant, tout automate non-déterministe peut-être transformé en un automate déterministe (voir 71 preuve de déterminisation). On utilise les deux notations suivantes :

- $E(q)$, fermeture réflexive et transitive de l'état q par la relation $\{(q, \epsilon, q') : (q, \epsilon, q') \in \Delta\}$.
(fermeture par ϵ -transition)
- $\delta(Q, a) = \{E(p) : q \in Q \wedge (q, a, p) \in \Delta\}$. (fermeture par l'entrée a puis par ϵ -transition)

Les états finals sont les $F' = \{Q \in K' : Q \cap F \neq \emptyset\}$. L'algorithme de construction est en $O(2^n)$ avec n états dans l'automate non-déterministe. Attention : il n'est pas nécessairement mieux d'avoir un automate complètement déterministe. D'une part, le nombre de ses états peut potentiellement augmenter jusqu'à 2^n , et d'autre part on réduit considérablement les possibilités de traitement parallèle ; on peut donner à traiter plusieurs morceaux d'un automate non-déterministe lors d'intersection sur des ϵ -transitions.

5) Fermetures

Le langage accepté par les automates finis est-il clôt sous l'union ? Oui, il suffit de créer une ϵ -transition à partir d'un nouvel état initial vers tous les états de l'union. Formellement :

$$K = K_1 \cup K_2 \cup \{\epsilon\} \text{ tel que } s \notin K_1 \cup K_2 \quad F = F_1 \cup F_2 \quad \Delta = \Delta_1 \cup \Delta_2 \cup \{(s, \epsilon, s_1), (s, \epsilon, s_2)\}$$

Est-il clôt sous le complément ? Oui, il faut intervertir les états finals et non finals, et on complète les états en les voyant vers un nouvel état commun (« *sink state* »).

La clôture par intersection découle des deux clôtures précédents, si l'on applique les règles de De Morgan:

$$\overline{\mathcal{L}(M_1) \cap \mathcal{L}(M_2)} = \overline{\mathcal{L}(M_1)} \cup \overline{\mathcal{L}(M_2)}$$

On a aussi une clôture par concaténation : pour deux automates M_1, M_2 , il suffit de faire une ϵ -transition de tout état final de M_1 vers l'état initial de M_2 . Enfin, on a la clôture sous la fermeture de Kleene. Quel est l'intérêt de toutes ces propriétés ? La construction automatique d'automates (c.f. algo de Thompson).

6) Un langage est régulier ssi il est accepté par un automate fini

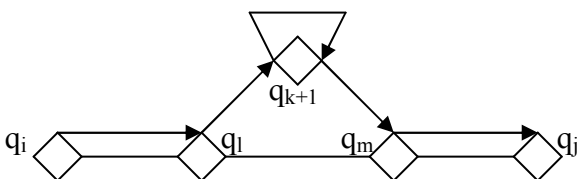
Voir preuve page 79. On traitera ici que le langage reconnu par un automate fini est régulier.

Un langage régulier est la fermeture de $\{w : w \text{ sur } \Sigma \cup \{\emptyset\}\}$ par concaténation, fermeture de Kleene, union. Soit $\mathcal{R}(i, j, k)$ toutes les chaînes acceptées par les chemins de l'état q_i à q_j de rang k (cela veut dire qu'on ne dépasse pas l'état q_k). $\mathcal{L}(M) = \cup \{\mathcal{R}(i, j, n) : q_j \in F\}$. Si on prouve $\mathcal{R}(i, j, k)$ régulier alors $\mathcal{L}(M)$ régulier.

Preuve par induction. Base : $k = 0$. $\mathcal{R}(i, j, 0)$ est régulier.

Hypothèse (assumption). $\mathcal{R}(i, j, k)$ sont réguliers.

Preuve. $\mathcal{R}(i, j, k+1) = \mathcal{R}(i, j, k) \cup \mathcal{R}(i, l, k) \mathcal{R}(l, k+1, 1) \mathcal{R}(k+1, k+1, k)^* \mathcal{R}(k+1, m, 0) \mathcal{R}(m, j, k)$



Au lieu de faire le chemin normal de q_i à q_j , on va en sortie à un point q_l pour joindre le nouvel état q_{k+1} , et on y re-rentre en q_m . On va donc de q_i à q_l au pire en k étapes, on va directement à q_{k+1} (où on reste aussi longtemps qu'on veut), on va directement à q_m et on peut aller en k étapes à q_j . C'est une preuve plutôt par construction.

7) Pumping Theorem

Si $\mathcal{L} \in \text{REG}$, il existe un unique $n > 0$ tel que pour tout $w \in \mathcal{L}$, $|w| > n$, $w = xyz$, $y \neq \varepsilon$ et $xy^iz \in \mathcal{L} \quad \forall i \geq 0$. Autrement dit, il y a une partie du mot que l'on peut dupliquer autant de fois que l'on veut « pour tout mot assez grand ». Cela marche pour les langages qui contiennent au moins une chaîne infinie ; dans ce cas, cela nous donne une infinité de mots infinis puisqu'il suffit que l'on en prenne un, on applique le pumping theorem et on obtient une chaîne infinie plus longue.

Ce théorème est utilisé en preuves par contradiction pour montrer qu'un langage n'est pas régulier.

$\mathcal{L} = \{a^n b^n : n \geq 0\} \subseteq \mathcal{L}_1(a^* b^*) \subseteq \mathcal{L}_2(w \in \{a,b\}^* : |w|_a = |w|_b)$. On affirme que \mathcal{L} est régulier (preuve par contradiction). Donc, pour un certain n on a $a^n b^n = xyz$, $y \neq \varepsilon$ et $xy^iz \in \mathcal{L}$. On a 3 possibilités :

- Le y contient juste des a , donc $xy^iz = a^{n-k} a^{k*i} b^n$ or le nombre de a augmente mais pas les b , $\notin \mathcal{L}_2$
- Le y contient juste des b , donc xy^iz donc le nombre de b augmente mais pas les a , $\notin \mathcal{L}_2$
- Le y contient un peu des deux, donc $y = a^k b^l$ et $xy^2z = a^{n-k} a^{k*2} a^{k*1} b^{n-l} b^l \notin \mathcal{L}_1$

Donc \mathcal{L} n'est pas un langage régulier.

Comment montrer que \mathcal{L}_2 n'est pas régulier ? Ce n'est pas parce qu'il inclut un langage non régulier qu'il est nécessairement non régulier : un ensemble plus petit n'est pas nécessairement plus faible.

$\mathcal{L}_2 \cap \mathcal{L}_1 = \mathcal{L}$ car avoir des a avant les b et autant de a que de b correspond au langage \mathcal{L} . Or, \mathcal{L}_2 est régulier par assumption, \mathcal{L}_1 est régulier par définition mais \mathcal{L} n'est pas régulier alors qu'il devrait. Comme les langages réguliers sont clôt par intersection, alors c'est que \mathcal{L}_2 n'est pas un langage régulier.

Montrons que $\{a^n : n \text{ est premier}\}$ n'est pas non plus régulier. $xy^iz = a^k a^{l*i} a^m$ tel que $k+l+m$ était premier. On devrait avoir $k+m+li$ premier pour tout i dans \mathbb{N} . Or si $i = k+m$ on a $(k+m)(l+1)$, qui n'est pas premier.

8) Simulation d'un automate

Entrée : $w \in \Sigma^*$. Sortie : oui ou non (appartenance au langage représenté). Pour un automate déterministe :

- | | |
|------------------------------------|--|
| 1. $S \leftarrow s_0$ | <i>On commence par l'état initial</i> |
| 2. while $w \neq \varepsilon$ | <i>Tant qu'on a un symbole à traiter</i> |
| 3. let $w = aw'$ | |
| 4. if $\delta(S, a) = s'$ then | <i>S'il y a une transition alors on l'applique</i> |
| 5. $S \leftarrow s'$ | |
| 6. $w \leftarrow w'$ | |
| 7. else return false | <i>Sinon rejet immédiat</i> |
| 8 return $(S \in F)$ | <i>Il faut être sur un état final pour accepter le langage</i> |

La simulation se fait en $O(|w|)$. Maintenant, dans le cas d'un automate non-déterministe, il faut considérer en même temps tous les autres chemins possibles. On arrive à l'algorithme suivant :

- | | |
|---|--|
| 1. $S \leftarrow E(s_0)$ | <i>On prend tous les états atteints par ε-transition sur l'état init.</i> |
| 2. while $w \neq \varepsilon$ | <i>Tant qu'on a un symbole à traiter</i> |
| 3. let $w = aw'$ | |
| 4. $S' \leftarrow \emptyset$ | |
| 4. for each $s_i \in S$, if $\Delta(s_i, a, s')$ | |
| 5. $S' \leftarrow S' \cup \{E(s')\}$ | |
| 6. $w \leftarrow w'$ | |
| 7. if $S' = \emptyset$ return false | |
| 8 return $(S \cap F \neq \emptyset)$ | |

La simulation se fait en $O(|k|^2|w|)$. C'est extrêmement intéressant, surtout comparé à l'algorithme de détermination qui est exponentiel. Il est donc mieux de simuler un automate non déterministe que de vouloir le déterminer d'abord (surtout si on additionne des coûts de minimisation par la suite...).

III. Grammaires formelles : Context Free Grammar

1) Les principes

Il y a typiquement des choses que l'on peut vouloir exprimer qui sortent du contexte des langages réguliers. Typiquement, une langue naturelle ou un programme informatique. Même certaines expressions simples ne sont pas régulières, telles $\{d^* a^n d^* b^* d^* : n \geq 0\}$; cette expression est pourtant bien pratique car elle représente un programme avec une fonction déclarée d'arité n et utilisée d'arité n , par exemple.

Une grammaire $G = (V, \Sigma, R, S)$ est composée des éléments suivants :

- Σ : terminaux
- $V \supseteq \Sigma$: alphabet
- $V - \Sigma$: non terminaux (utilisés dans les productions)
- $S \in (V - \Sigma)$: axiome, ou symbole de départ
- $R \subseteq (V - \Sigma) \times V^*$: les règles, de la forme $\{V - \Sigma \rightarrow V^*\}$, soit {non terminaux expansés en un mix}.

Une règle s'écrit avec \rightarrow et une réécriture avec \Rightarrow . Ainsi :

$$v \Rightarrow w \text{ ssi } \exists x, y \in V^*, a \in (V - \Sigma), (a \rightarrow z) \in R, \text{ tel que } v = xay \text{ et } w = xyz.$$

\Rightarrow_G indique qu'on réécrit avec la grammaire G . \Rightarrow_G^* est la fermeture réflexive et transitive de \Rightarrow et on la nomme dérivation. Un langage peut se définir par succession de règles :

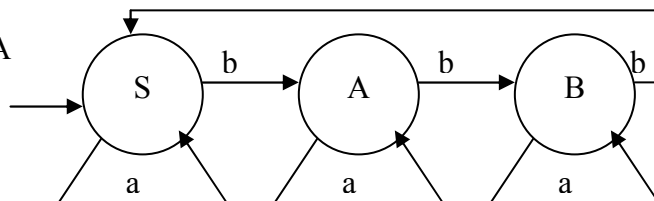
$\mathcal{L}(G) = \{w \in \Sigma^* : S \Rightarrow_G^* w\}$ (ensemble des mots auxquels on arrive en partant de l'axiome S et par règles)

Les grammaires context-sensitive ont les règles $R \subseteq V^*(V - \Sigma)VV^*$ soit $\alpha A \beta \rightarrow \alpha w \beta$ où α et β sont le contexte autour de A .

Exemple : $G = (\{S, a, b\}, \{a, b\}, R, S)$ tel que $R = \{S \rightarrow aSb, S \rightarrow \epsilon\}$. Cela engendre $\mathcal{L} = \{a^n b^n : n \geq 0\}$.

Exemple de règles plus complexes :

$S \rightarrow \epsilon$	$S \rightarrow aS$	$S \rightarrow bA$
$A \rightarrow aA$	$A \rightarrow bB$	
$B \rightarrow aS$	$B \rightarrow bB$	



Tout langage accepté par un automate fini peut-être accepté par une grammaire (puisque c'est la classe supérieure de langage). Soit un automate $M = (K, \Sigma, s_0, \Delta, F)$ et on veut $G(V, \Sigma, R, S)$ la grammaire telle que $\mathcal{L}(M) = \mathcal{L}(G)$. On la construit de façon suivante :

$V = K \cup \Sigma$
 $S = s_0$
 $R = \{q \rightarrow ap : (q, a, p) \in \Delta\} \cup \{q \rightarrow \epsilon : q \in F\}$.

On recopie toutes les transitions et on arrête la grammaire sur un état final.

Ces règles sont spéciales : $R \subseteq (V - \Sigma) \times (\Sigma(V - \Sigma) \cup \{\epsilon\})$, caractéristiques de grammaires régulières (et donc cas particuliers de grammaires context free). Comme on a un exemple de grammaire context free qui n'est pas régulier, alors la classe des langages réguliers est strictement incluse dans celle context free.

Le langage des parenthèses balancées (*balanced brackets*) est défini par les règles :

$S \rightarrow (S)$
 $S \rightarrow \epsilon$
 $S \rightarrow SS$

Un exemple : $(())()$ donne $S \rightarrow SS \rightarrow (S)S \rightarrow (S)(S) \rightarrow ((S))(S) \rightarrow (())(S) \rightarrow (())()$

Cependant, il y a plusieurs façons de produire $(())()$ à partir de nos règles. On dit qu'il y a plusieurs parcours ou dérivations possibles. Deux dérivations sont similaires si elles diffèrent uniquement dans l'ordre d'application de règles consécutives.

Chaque langage Context Free est accepté par un automate à pile, et chaque automate à pile représente un langage Context Free (i.e. il y a équivalence). Montrons que pour toute grammaire Context Free, on peut créer un automate à pile qui l'accepte. On a la grammaire $G = (V, \Sigma, R, S)$ et $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$:

$$\Delta = \{((p, \varepsilon, \varepsilon), (q, S))\} \cup \{((q, a, a), (q, \varepsilon)) : a \in \Sigma\} \cup \{((q, \varepsilon, A), (q, w)) : A \rightarrow w \in R\}$$

Pour le langage défini par les règles $S \rightarrow SS, S \rightarrow (S), S \rightarrow \varepsilon$, cette définition automatique donne :

$$\{((p, \varepsilon, \varepsilon), (q, S)), ((q, '(', '('), (q, \varepsilon)), ((q, ')', ')'), (q, \varepsilon)) ((q, \varepsilon, S), (q, SS)) \\ ((q, \varepsilon, S), (q, (S))), ((q, \varepsilon, S), (q, \varepsilon))\}$$



MIDTERM Jeudi au retour des vacances à New York
Open Book

Maintenant on a un automate et on veut en extraire la grammaire (prouve l'équivalence dans l'autre sens). On définit un automate à pile simple (PDA) sur lequel on veut opérer, de la façon suivante :

Soit $M = (K, \Sigma, \Gamma, S, \Delta, F)$. Pour tout $((q, a, \alpha), (q, \beta)) \in \Delta$, avec $q \neq S$, on a :

- $\alpha \in \Gamma$
- $|\beta| \leq 2$ (limite de ce qu'on peut pousser sur la pile à chaque fois : 2 symboles)

Cet automate est 'simple' car on établit des restrictions : on ne peut regarder qu'un seul symbole et en pousser au maximum deux. Maintenant, on cherche à rendre tout automate simple par une transformation. On met un symbole z de fond de pile, pour s'assurer qu'il y ait toujours quelque chose : $\Gamma = \Gamma' \cup \{z\}$. On ajoute donc la transition $((s', \varepsilon, \varepsilon), (s, z))$ où s' est le nouvel état initial qui nous permet de mettre z . On invente un nouvel état final : $F' = \{f'\}$, et on ajoute les transitions $((f, \varepsilon, z), (f', \varepsilon)) : f \in F$.

Il faut s'assurer que l'on conserve les restrictions sur toute transition. On ne veut pas les $((q, a, \alpha), (q', \beta))$:

- où $\alpha = \alpha_1 \dots \alpha_n, n > 1$, car on regarde plus d'un symbole
- où $\beta = \beta_1 \dots \beta_n, n > 2$, car on empile plus de deux symboles

La solution est de créer des états intermédiaires : utiliser plusieurs états qui regardent un symbole si on veut simuler un état qui regarde plusieurs symboles. Ainsi, $((q, a, \alpha_1 \dots \alpha_n), (q', \beta))$ est remplacé par :

$$\{((q, a, \alpha_1), (q_1, \varepsilon)) ((q_1, \varepsilon, \alpha_2), (q_2, \varepsilon)) \dots ((q_{n-1}, \varepsilon, \alpha_n), (q', \beta))\}$$

où $q_i, 1 \leq i < n$, sont les nouveaux états

De même, on remplace $((q, a, \alpha), (q', \beta_1 \dots \beta_n))$ par :

$$\{((q, a, \alpha), (q_n, \beta_n)) ((q_n, \varepsilon, \beta_{n-1}), (q_{n-1}, \beta_{n-1} \beta_n)) \dots ((q_2, \varepsilon, \beta_2), (q', \beta_2 \beta_1))\}$$

il faut empiler à partir de la fin n . A chaque fois on pose le symbole vu aussi, et c'est pour cela que notre automate simple a besoin d'avoir jusqu'à 2 symboles empilés en même temps.

A présent, on peut utiliser l'automate simplifié pour en extraire une grammaire $G = (V, \Sigma, R, S)$. On a :

- ◆ $V = \Sigma \cup \{S\} \cup \{<p, A, q> : p, q \in K', A \in \Gamma'\}$ où a <start state, top of the stack, end state>
- ◆ $\{S \rightarrow <s, z, f'\>\} \cup$ shortcut : je démarre de s avec z sur la pile
- ◆ $\{<q, A, r> \rightarrow a<q', \beta, r> : ((q, a, A), (q', \beta)), \text{ avec } A, B \in \Gamma \cup \{\varepsilon\}, r \in K'\}$
 $\cup \{<q, A, r> \rightarrow a<q', \beta_1, r><r', \beta_2, r> : ((q, a, A), (q', \beta, \beta_2)) \in \Delta, \text{ avec } A, \beta_1, \beta_2 \in \Gamma, r, r' \in K'\}$
 $\cup \{<q, \varepsilon, q> \rightarrow \varepsilon : q \in K'\}$

4) Opérations sur les grammaires

Soient deux grammaires $G_1 = (V_1, \Sigma, R_1, S_1)$ et $G_2 = (V_2, \Sigma, R_2, S_2)$, $V_1 \cap V_2 \neq \emptyset$, $S \notin V_1$, $S \notin V_2$. On veut construire la grammaire $G = (V, \Sigma, R, S)$ pour les opérations suivantes :

- Union : $\mathcal{L}(G) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$. On met simplement un nouvel état initial S et l'on réunit, d'où
 $V = V_1 \cup V_2 \cup S \quad R = R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}$
- Concaténation : $V = V_1 \cup V_2 \cup S \quad R = R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}$

- La fermeture de Kleene n'est qu'une succession de concaténation : $\mathcal{L}(G) = \mathcal{L}(G_1)^*$.
 $V = V_1 \cup S \quad R = R_1 \cup \{S \rightarrow \varepsilon, S \rightarrow S_1 S\}$
- Les langages Context Free ne sont pas clôt par complément, donc pas par intersection.

Soit un automate fini (reg. lang.) $M_1 = (K_1, \Sigma, S_1, \delta_1, F_1)$ et un automate à pile $M_2 = (K_2, \Sigma, \Gamma_2, S_2, \Delta_2, F_2)$. On veut construire un automate $M = (K, \Sigma, \Gamma, S, \Delta, F)$ tel que $\mathcal{L}(M) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. Comme on ne peut pas faire l'intersection entre deux langages Context Free, on le fait entre un langage Context Free et un langage régulier. Sur la même idée que le transducer, on veut construire un automate en parallèle :

$$K = K_1 \times K_2 \quad S = (S_1, S_2) \quad F = F_1 \times F_2$$

$$\Delta = \{ ((q, p), a, \alpha), ((q', p'), \beta) : q' = \delta(q, a) \wedge ((p, a, \alpha), (p', \beta)) \in \Delta_2 \}$$

Consuming one input : (pre-requisite, where we end)

$$\cup \{ (((q, p), a, \alpha), ((q', p'), \beta)) : ((p, \varepsilon, \alpha), (p', \beta)) \in \Delta_2 \}$$

Si l'input n'est pas considérée alors seul M_2 peut faire quelque chose car M_1 déterministe

Si les deux automates sont à pile et qu'on veut faire un device fonctionnant en parallèle, alors il lui faudrait 2 piles, donc nous sortons du cadre des automates à pile.

5) Pumping theorem for Context Free Languages

Soit une grammaire $G = (V, \Sigma, R, S)$. Le nombre maximal d'enfants que peut avoir le nœud du parsing tree est dit branching factor, ou fan-out, noté $\Phi(G)$, et donné par la plus longue règle de la grammaire. Soit un parse tree de hauteur h et largeur y . On a $|y| \leq \Phi(G)^h$. Au 1^{er} niveau, on a 1 nœud et il peut y avoir jusqu'à $\Phi(G)$ enfants ; de même au niveau 2 on va jusqu'à $\Phi(G)^2$ enfants, etc. etc.

Soit un chemin de longueur maximale dans l'arbre et $h > |V - \Sigma|$. Par le pigeon-hole principle, on en conclut qu'il y a au moins un non-terminal répété. Il existe donc une portion où l'on peut dupliquer autant de fois que l'on veut.

For any $w \in \mathcal{L}(G) \wedge |w| > \Phi(G)^{|V-\Sigma|}$, we have $w = xyzuv$ with $yu \neq \varepsilon$ and $xy^n zu^n v \in \mathcal{L}(G)$ for all $n \geq 0$.

Prouvons que $\mathcal{L} = \{a^n b^n c^n : n \geq 0\}$ n'est pas context-free. On fait la preuve classique par contradiction.

On affirme que \mathcal{L} est context free. Il y a donc $w = xyzuv$ tel que $xy^k zu^k v \in \mathcal{L}$. Pour y et u , 2 variantes :

- yu contiens a, b et c alors $xy^k zu^k v \in \mathcal{L}(a^* b^* c^*)$ car on a le résultat dans le désordre
- yu ne contient pas a , alors $xy^k zu^k v$ ne contient pas un nombre égal de a, b , et c . De même b et c .

Soient $\mathcal{L}_1 = \{a^m b^m c^n : m, n \geq 0\}$, $\mathcal{L}_2 = \{a^m b^n c^n : m, n \geq 0\}$. On remarque $\mathcal{L}_1 = \{a^m b^m : m \geq 0\} \cdot \{c^n : n \geq 0\}$.

La première partie de \mathcal{L}_1 est context-free et la seconde régulière, donc \mathcal{L}_1 est context free. En revanche, $\mathcal{L}_2 = \{a\}^* \cdot \{b^n c^n : n \geq 0\}$ et $\mathcal{L}_1 \cap \mathcal{L}_2 = \mathcal{L}$ qui n'est pas context-free. Donc, les langages context-free ne sont pas clôt par intersection, ce qui veut dire qu'ils ne le sont pas non plus par complément.

6) Automate à pile déterministe

Deux séquences α, β sur 2 alphabets sont dites consistants si α est un préfixe de β ou β est un préfixe de α . Les transitions $((p, a, \alpha), (q, \beta))$ et $((p, a', \alpha'), (q, \beta'))$ sont dites compatibles si a et a' sont consistants ainsi que α et α' . Un automate à pile déterministe est un automate à pile dans lequel il n'y a pas deux transitions distinctes qui sont compatibles.

\mathcal{L} est un langage context-free déterministe si $\mathcal{L}\$$ est accepté par un automate à pile déterministe. Les automates à pile déterministes sont clôt par complément et intersection ; corollaire : les langages context-free déterministes sont un sous-ensemble strict des langages context-free.

7) Complément d'un automate fini à pile

Pour établir le complément, le problème est la gestion de la pile. Soit M un automate simple et une configuration $(q, aw, bw) \vdash_M (q', w, b'u)$ pouvant résulter de :

- $((q, \epsilon, b), (q', b')) \rightarrow a = \epsilon$ et $b' = \epsilon$ ou $b' \neq \epsilon$. On a un espoir d'accepter l'entrée si ça fait descendre la pile.
- $((q, a, b), (q', b')) \rightarrow a \neq \epsilon$.

On veut identifier les transitions de Δ qui ne peuvent pas conduire à accepter l'entrée. Ainsi (p, α, β) est sans issue (*dead end*) s'il ne conduit pas en une étape à (p', ϵ, β) ou (p', α, ϵ) . On remplace toute transition considérée comme morte/inutile en utilisant le sink state (l'état puit sur lequel on envoie tout) :

$$((p, \alpha, \beta), (q, \gamma)) \in \mathcal{D} \rightarrow ((p, \alpha, \beta), (r, \gamma)) \text{ où } r \text{ est le nouvel état}$$

On ajoute des transitions afin de rejeter de façon civilisée : on regarde quand même la chaîne jusqu'à la fin et on vide la pile. On introduit les $((r, a, \epsilon), (r, \epsilon))$ pour tout $a \in \Sigma$, $((r, \$, \epsilon), (r', \epsilon))$ où r' est un nouvel état, et $((r', \epsilon, \alpha), (r', \epsilon))$ pour tout $\alpha \in \Gamma$.

On a $\mathcal{L}(M') = \mathcal{L}(M)$ car on accepte toujours le même langage mais on a juste modifié les rejets. On fait de r' le seul état final (on intervertit les rôles). Par conclusion, cette preuve constructiviste montre que les langages context-free déterministes sont clôt par complément.

8) Simulation

On veut un automate acceptant le langage généré par une grammaire $G = (V, \Sigma, R, S)$. On fait du bottom-up parsing sur des grammaires dites *weak precedence* (c'est le cas de toutes les grammaires pour langages de programmation récents) ; SLR et LR(1) sont des restrictions de ce type de grammaire. On simule par :

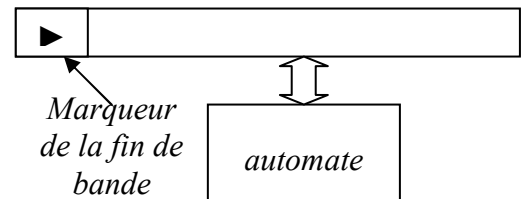
$$\Delta = \{ ((p, a, \epsilon), (p, a)) : a \in \Sigma \} \cup \{ ((p, \epsilon, \alpha^R), (p, A)) : A \rightarrow \alpha \in R \} \cup \{ ((p, \epsilon, s), (q, \epsilon)) \}$$

Shift *Reduce*

IV. Machines de Turing et Calculabilité

1) Définition

Dans une machine de Turing, on peut se déplacer comme on veut de façon linéaire, lire et écrire dans la bande. Toute la bande est initialisée par le symbole \sqcup ou #. Formel : $M = (K, \Sigma, \delta, s, H)$ où les états finis H sont les Halting states, Σ est l'alphabet de la bande avec deux symboles spéciaux (end, marker, blank) et deux symboles L (bouger la tête à gauche), R (bouger la tête à droite). Une fois dans un Halting State, la règle est qu'on ne peut en sortir.



Les transitions sont les $\delta : K \times \Sigma \rightarrow K \times (\Sigma \cup \{L, R\})$.
 On définit certaines restrictions :
 ← Réécrire le contenu de la cellule courante
 → ou faire bouger la tête

- pour tout $h \in H$, $f(h, a)$ est indéfini pour tout $a \in \Sigma$. On ne sort pas d'un halting state.
- Si $\delta(q, \blacktriangleright) = (p, b)$ alors $b = r$. Si on vient sur le marqueur de fin de bande alors la seule action possible est d'aller à droite, puisqu'on est à la fin.
- Si $\delta(q, a) = (p, b)$ alors $b \neq \blacktriangleright$.

Une configuration c est telle que $c \in K \times \Sigma^* \times \Sigma^* ((\Sigma - \{\#\}) \cup \{\epsilon\})$. On la note par $(q, \underline{u}aw)$ en soulignant le symbole où la tête se trouve actuellement. Par exemple $\blacktriangleright abbaa\#a###$ donne $(q, abbaa\#a)$.

Le *yield in one step* (mouvements de la machine) est $((q, uaw) \vdash_M (q', u'a', w'))$ si $\delta(q, a) = (q', b)$ pour $b \in \Sigma \cup \{L, R\}$ et :

- si $b \in \Sigma$ alors $u = u'$, $w = w'$, $a' = b$
- si $b = L$ (bouger la tête à gauche) alors $u'a' = u$, $w' = aw$ si $a \neq \#$ ou $w \neq \epsilon$ sinon $w' = \epsilon$
- si $b = R$ (bouger la tête à droite) alors $u' = ua$, $w = a'w'$ si $a' \neq \#$ sinon $w = w' = \epsilon$

Construisons une machine de Turing pour s'entraîner. Soit $\Sigma = \{i, \#, \blacktriangleright\}$, $s = q_1$, $H = \{k\}$, $K = \{q_1, q_2, q_3, q_4\}$

$\delta(q_1, \#) = (q_2, L)$ $\delta(q_2, i) = (q_2, \#)$ $\delta(q_2, \#) = (q_3, L)$ $\delta(q_3, i) = (q_3, L)$
 $\delta(q_3, \#) = (q_4, i)$ $\delta(q_4, i) = (q_4, R)$ $\delta(q_4, \#) = (h, \#)$

On fait marcher la machine sur un exemple : $(q_1, \#iii\#i\#) \vdash (q_2, \#iii\#i) \vdash (q_2, \#iii\#i\#) \vdash (q_3, \#iii\#i) \vdash (q_3, \#iii\#i) \vdash (q_4, \#iii\#i) \vdash (q_4, \#iii\#i) \vdash (q_4, \#iii\#i\#) \vdash (h, \#iii\#i\#)$.

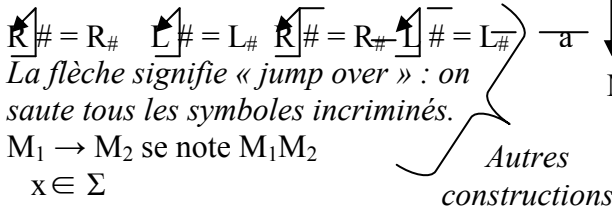
Cette machine concatène 2 chaînes de contenu identique. Si on compte en nombre de barres on peut même dire que ça réalise une addition...

2) Comment écrire une machine de Turing

On définit une machine de base (basic machine) M_a :

pour tout $x \in \Sigma$: $\delta(s, x) = (h, a)$.
 pour tout $a \in \Sigma$: $a = M_a, L = M_L, R = M_R$

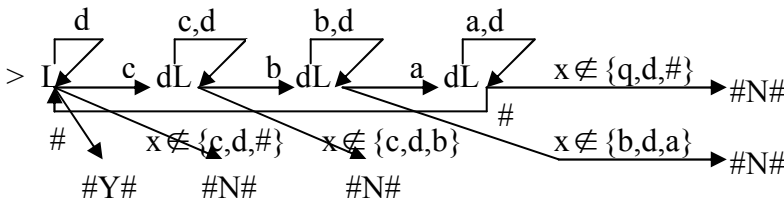
On peut combiner ces machines de base :



Ca donne une machine de turing M :
 $\triangle \delta = \delta_1 \cup \delta_2 \cup \delta_3$
 \triangle remplacer toutes les occurrences de h dans δ_1 par q nouveaux états
 \triangle on ajoute $\delta(q, a) = q_2$ et pour tout $x \in \Sigma \setminus \{a\}$, $\delta(q, x) = q_3$ où q_2 est l'état de départ de M_2 et q_3 celui de M_3

En utilisant nos nouvelles constructions, la machine précédente s'écrit $>L \rightarrow \#L\# i R\#$

Soit la machine $L = \{a^n b^n c^n : n \geq 0\}$. La machine accepte avec $\#Y\#$ et refuse avec $\#N\#$. On l'écrit par :



On réécrit sur a, b et c jusqu'à ce qu'il n'y ait plus rien à réécrire.

3) Langages rékursifs et rékursivement énumérables

On dit que M décide le langage L si en recevant $\#w\#$ alors M s'arrête toujours et donne $\#Y\#$ si $w \in L$ $\#N\#$ sinon

Si M décide L alors L est un langage rékursif, c'est-à-dire défini en utilisant des fonctions rékursives primaires qui sont des outils aussi puissantes que les machines de Turing (et dont le mot a été emprunté).

Soit M une machine de Turing, $M = (K, \Sigma, \delta, s, \{h\})$ et $M(w) = y$ où $(s, \#w\#) \vdash_M^* (h, \#y\#)$.

$f : \Sigma^* \rightarrow \Sigma^*$ est rékursif ssi il existe une machine de turing M tel que pour tout $w \in \Sigma^*$, $M(w) = f(w)$.

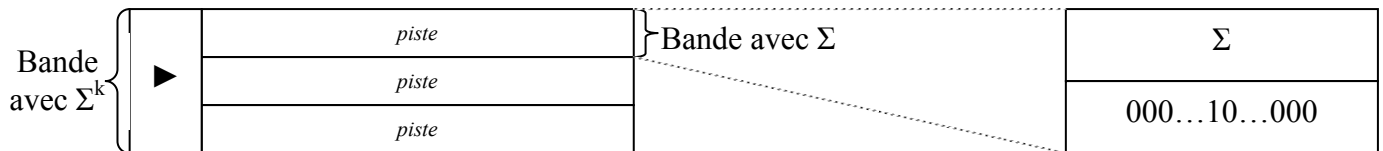
Une machine de turing M semi-décide un langage $L \in \Sigma^*$ ssi M s'arrête sur w ssi $w \in L$. Alors L est un langage rékursivement énumérable.

Si on décide, alors on semi-décide aussi ; il suffit de faire une boucle infinie pour ne pas s'arrêter au lieu de mettre $\#N\#$. En revanche, il y a des langages rékursivement énumérables qui ne sont pas rékursifs.

4) Machine de turing multi-tape

On a k tapes pour un k donné. La fonction de transition de la machine était $\delta : K \times \Sigma \rightarrow K \times (\Sigma \cup \{L, R\})$ et maintenant qu'on a k tapes, cela donne $\delta : K \times \Sigma^k \rightarrow K \times (\Sigma \cup \{L, R\})^k$.

Pour toute machine de Turing à k tapes, il existe une machine de turing à 1 seule bande faisant la même chose. C'est simplement le côté pratique qui nous intéresse, pour isoler des parties en les mettant sur des bandes différentes. Pour prouver que cette machine n'est pas plus puissante que celle de base, on prend toutes les bandes et on les considère comme les pistes d'une grosse bande, en sauvegardant l'état courant.



Chaque bande est divisée en 2 parties. La partie inférieure marque (avec un 1) la position de la tête sur la bande d'origine : il ne faut pas négliger le fait que chaque bande était censée avoir sa propre tête. L'alphabet de la machine est maintenant $\{\blacktriangleright, \#\} \cup (\{0,1\} \times \Sigma)^k$. On scanne la bande une fois pour trouver, et une seconde fois pour modifier ; chaque transition se fait en $k_n + k_n = O(n)$; autrement dit on est $O(n)$ fois plus lents que la machine d'origine, mais cela ne modifie pas la classe de complexité puisqu'on reste polynomial ou exponentiel selon ce qu'on était au départ.

5) Variante : Random Access Turing Machine

Dans cette machine, on a un accès en temps constant à n'importe quel endroit de la bande, comme la mémoire d'un ordinateur. Rappelons que la machine de Turing de base on doit faire passer la tête par toutes les cases intermédiaires. Une fois encore, cela n'augmente pas le pouvoir de calculabilité et ne modifie pas la classe de complexité.

6) Machines de Turing non déterministes

Il s'agit d'une machine de Turing classique $M = (K, \Sigma, \Delta, s, H)$ où Δ est une relation de transition non déterministe telle que $\Delta \subseteq K \times \Sigma \times K \times (\Sigma \cup \{L, R\})$; on autorise plusieurs futurs.

On veut savoir si un nombre n est composite (*i.e.* il admet une décomposition en nombre premiers). La machine devine p et q , $p \leq n$, $q \leq n$, et on regarde si $pq = n$. La multiplication pq se fait assez facilement puisqu'on a déjà une machine permettant d'additionner. Pour comparer, il suffit de faire un « et » bit à bit entre les deux bandes. La partie intéressante est donc le « *non-deterministic guess* », ou la possibilité que la machine puisse deviner. Pour cela, elle essaye en parallèle toutes les combinaisons possibles de p et q . Pour générer des p et q , on les fait chacun sur leur bande en mettant des 0 et des 1 tant qu'on est pas sorti de n (dès qu'on lit un # sur la 1^{ère} ligne alors on arrête la machine) ; si on a pas de blanc, on peut s'arrêter ou continuer : c'est important car on doit aussi pouvoir générer les nombres inférieurs à n .

Accepter une chaîne dans la machine de Turing de base consiste à s'arrêter dans un halting state. Ici, la machine accepte ou semi-décide un langage $\text{ssi } (s, \#w\#) \vdash_M^* (h, x)$ où x est n'importe quel output ; on conserve la même définition mais la sémantique est un peu différente dans le cas non-déterministe.

M décide un langage \mathcal{L} ssi $(s, \#w\#) \vdash_M^* (h, \#y\#)$ si $x \in \mathcal{L}$ et $(s, \#w\#) \vdash_M^* (h, \#n\#)$ si $x \notin \mathcal{L}$
 et il existe $n \geq 0$ tel qu'il n'y a pas de c tel que $(s, \#w\#) \vdash_M^N c$
 (autrement dit toutes les branches doivent s'arrêter un jour... on l'expande :
 $c_1 \vdash_M^N c$ ssi il existe c_1', c_2', \dots, c_n t.q. $c_1' = c_1, c_n' = c_2, c_1' \vdash_M c_2', \vdash_M \dots \vdash_M c_n'$)

Une machine de Turing M calcule la fonction $f : \Sigma^* \rightarrow \Sigma^*$ si $M(w) = f(w)$ pour tout $w \in \Sigma^*$ et que toutes les branches possibles s'arrêtent. Une alternative à la 1^{ère} condition est $(s, \#w\#) \vdash_M^* (h, \#f(w)\#)$.

Pour toute machine de Turing non déterministe qui décide ou semi-décide un langage, ou calcule une fonction, il existe une machine de Turing déterministe qui fait la même chose.

$$c \vdash_M \begin{matrix} c_1 \\ \dots \\ c_m \end{matrix} \text{ avec } m \leq \underbrace{|\mathbf{K}| (|\Sigma| + 2)}_{\text{, car } \Delta \subseteq K \times \Sigma \times K \times (\Sigma \cup \{L, R\})}$$

On a un degré maximal de non-déterministe. Soit M_d la variante déterministe pour M , elle a 2 branches :

- working tape (contient le mot w)
- guiding tape (contient une chaîne de nombres, ou une séquence, tel que $1 \leq i_j \leq |\mathbf{K}|(|\Sigma| + 2)$)

On utilise une machine de Turing déterministe M' avec 3 bandes :

- la 1^{ère} contient w en read only (aucune altération possible)
 - la 2nd est la working tape
 - la 3^{ème} contient une chaîne de nombres
- } Tous les possibles M_d

Et on la fait fonctionner de la façon suivante :

1. Copier w sur la working tape
2. Lancer M_d
3. Si on obtient l'output désiré (en accord avec ce qu'on cherche) sur la 2nd tape, on s'arrête.
Formellement : si M_d atteint (h, u) pour un $u \in \Sigma^*$
4. Sinon, on génère le contenu suivant de la 3^{ème} tape en suivant l'ordre lexicographique (i.e. on génère une autre machine M_d de façon ordonnée)
5. Recommencer à l'étape 1

Si l'output est produit en n étapes, comme on teste toutes les configurations avec l'ordre lexicographique, on obtient une complexité $1 + m + m^2 + \dots + m^n = \sum_{i=1}^n m^i = O(m^n)$.

7) Machine de Turing Universelle

Considérons une machine de Turing universelle U (en gros, un ordinateur) qui prend une chaîne et une machine de Turing à simuler ; U renvoie une certaine sortie. On cherche une façon d'écrire une machine de Turing en binaire pour qu'elle puisse être donnée comme argument à U , et ainsi simulée. On note par $\langle n \rangle$ les états, avec n un codage binaire à taille fixée selon la machine de Turing ; on a $q_0 \dots 0$ l'état initial et $q_1 \dots 1$ le halting state. Une fois les états encodés, on s'occupe du reste : l'alphabet Σ et les transitions Δ . On note l'alphabet par $\langle n \rangle$ avec n un code binaire à taille fixée selon la machine de Turing ; on a quelques symboles spéciaux : $a_0 \dots 0$ pour #, $a_0 \dots 1$ pour ►, $a_0 \dots 10$ pour L et $a_0 \dots 11$ pour R. Les transitions sont des (q, a, q', a') et comme chacun peut s'encoder, il suffit de concaténer. Par exemple :

$q : q_001 \quad a : a_111 \quad q' : q_010 \quad a' : a_000 \quad \text{concaténation } q_001a_111q_010a_000\$$

$M \rightarrow \langle M \rangle$ dénote l'encodage de M , ou encore $M \rightarrow \text{enc}(M)$. On voit que $\text{enc}(M) = \text{enc}(\Delta)$ car cela suffit pour spécifier complètement la machine (bijection).

On a $U(\text{enc}(M)\#\text{enc}(w)) = \text{enc}(M(w))$. Ainsi $(s, \#\text{enc}(M)\#\text{enc}(w)\#) \vdash_U^* (h, \#M(w)\#)$. U a trois bandes :

- input, soit $\#\text{enc}(M)\#\text{enc}(w)\#$
- storage tape, initialement vide
- simulation, initialement $q_0 \dots 0$

On commence par copier $\#\text{enc}(M)\#$ sur la 2nd bande et on simule M .

8) Le Halting Problem

Soit $\text{halt}(P, x) = \text{stop}$ ssi P stop sur l'entrée X . On définit $\text{diagonal}(X)$ par :

si $\text{halt}(X, X)$ s'arrête alors $\text{diagonal}(X)$ sinon stop

On donne à un programme le programme lui-même, et on lui demande de voir s'il va s'arrêter. On définit $H = \{\text{enc}(M)\#\text{enc}(w) : M \text{ s'arrête sur } w\}$ le langage correspondant, semi-décidé par U donc récursivement énumérable. Nous allons prouver qu'il n'est pas récursif, ou que M_H ne décide pas H .

On fait une preuve par contradiction : supposons que H est récursif et regardons ce que ça engendre. On devrait avoir $H_1 = \{\text{enc}(M) : M \text{ s'arrête sur } \text{enc}(M)\}$ récursif, et comme les langages récursifs sont clôtés par compléments alors $\neg H_1 = \{x : x \text{ n'est pas l'encodage d'une machine de Turing ou } x = \text{enc}(M) \text{ et } M \text{ ne s'arrête pas sur } x\}$ devrait être récursif. Comme $\neg H_1$ doit être récursif, il est aussi récursivement énumérable et il existe une machine M^* qui le semi-décide. Mais est-ce que $\text{enc}(M^*) \in \neg H_1$? Si oui :

- M^* ne s'arrête pas sur $\text{enc}(M^*)$ par définition de H_1
- ...mais la machine qui semi-décide $\neg H_1$ doit s'arrêter sur $\text{enc}(M^*)$.

Or la machine qui semi-décide $\neg H_1$ est M^* par définition de semi-décidabilité, d'où une contradiction. H n'est donc pas récursif, bien que récursivement énumérable.

Corollaire : les langages récursivement énumérables ne sont pas clôtés par compléments.

9) Calculabilité

Un problème est une fonction $f: \text{entrées} \rightarrow \text{sortie}$. Un langage vu par les automates est une fonction booléenne : oui si l'entrée est dans le langage, et non pour le reste. Examinons certains problèmes classiques en donnant leur énoncé, puis en les transcrivant sous forme de langages.

Problème du voyageur de commerce (*travelling salesman*). Soit I une matrice d'adjacence, on veut une permutation π des $\{1, \dots, n\}$ telle que $d_{\pi_1\pi_2} + d_{\pi_2\pi_3} + \dots + d_{\pi_{n-1}\pi_n} + d_{\pi_n\pi_1}$ (car c'est un cycle) soit minimale.

Définition du langage pour le voyageur de commerce. Etant donné (d_{ij}) , $k \geq 0$, y a-t-il une permutation π de $\{1, 2, \dots, n\}$ telle que $d_{\pi_1\pi_2} + d_{\pi_2\pi_3} + \dots + d_{\pi_{n-1}\pi_n} + d_{\pi_n\pi_1} \leq k$ (on veut minimiser).

Problème du mariage stable (*stable marriage*). On a un graphe non-orienté $G = (V, E)$ et on veut trouver l'ensemble maximal C de sommets tels que pour tout $v_i, v_j \in C$ alors $(v_i, v_j) \in E$; on recherche le plus gros graphe fortement connexe.

Définition du langage pour le mariage stable. Etant donné $G = (V, E)$, $k \geq 0$, y a-t-il un $C \subseteq V$ tel que pour tout $v_i, v_j \in C$ alors $(v_i, v_j) \in E$ et $|C| \geq k$ (on veut maximiser).

En général, on parle du langage correspondant à un problème lorsqu'on s'occupe de la théorie. Par abus, dans la suite du cours on identifiera un langage comme étant équivalent à un problème.

On a vu par le *halting problem* qu'il existe des langages indécidables. Ceci nous permettra de répondre à des questions comme : étant donné une machine de Turing M , est-ce qu'elle s'arrête sur une bande vide ? Est-ce qu'il existe une entrée telle que la machine s'arrête ? Ces problèmes sont toujours indécidables, et on peut le montrer avec des preuves par diagonalisation. Cependant, il existe un outil extrêmement pratique qui est la réduction. Une réduction de \mathcal{L}_1 à \mathcal{L}_2 (langages sur Σ^*) est une fonction récursive (i.e. il doit y avoir une machine de Turing qui peut la faire) $\zeta: \Sigma^* \rightarrow \Sigma^*$ telle que $w \in \mathcal{L}_1$ ssi $\zeta(w) \in \mathcal{L}_2$.

Si \mathcal{L}_1 n'est pas récursif et qu'il y a une réduction de \mathcal{L}_1 à \mathcal{L}_2 alors \mathcal{L}_2 n'est pas récursif. Preuve par contradiction : si \mathcal{L}_2 était récursif alors il y aurait une machine de Turing qui le déciderait, et une M_ζ ; comme cela équivaut à $M_\zeta M_2$ qui déciderait \mathcal{L}_1 , c'est une contradiction.

La réduction va d'un langage connu non-récursif à un langage inconnu (c'est le contraire de la théorie de la complexité). Etant donné M , $w = w_1 w_2 \dots w_n$, on veut $M_w = w_1 R w_2 R \dots w_n R M$ par ζ ; autrement dit on fait une réduction du halting problem en celui de savoir si la machine s'arrête sur une bande vide. On détruit w de la bande et on devine (non-deterministic guess) une entrée x , puis on lance M .

Etant donné M_1 et M_2 , est-ce ce qu'ils décident le même langage ? Ce serait extrêmement pratique si on pouvait dire oui, pour tout ce qui est vérification de la correction de programmes et de modèles... Voir le théorème de Rice et la preuve, page 270, comme suit : *Pour toute propriété exhaustive non triviale*

Soit RE la classe des langages récursivement énumérables $\forall C \subsetneq RE$ tel que $C \neq \emptyset$, le problème « Etant donné une machine de Turing M , est-ce que le langage décidé par M est dans C » est indécidable.

Autrement dit : « toute propriété non triviale à propos du langage reconnu par une machine de Turing est indécidable ».

Deux problèmes indécidables :

- déterminer si une grammaire CF génère toutes les chaînes possibles ou si elle est ambiguë
 - étant donné deux grammaires CF, déterminer si elles génèrent le même ensemble, ou si l'une génère un sous-ensemble de l'autre, ou s'il existe des chaînes qu'elles génèrent toutes les deux
- (voir aussi : complexité de Kolmogorov, problème de correspondance de Post, Tiling de Penrose, 10^{ème} problème de Hilbert i.e. déterminer si une équation Diophantienne admet une solution, nombres de Gödel)

V. Théorie de la Complexité

1) Time Bounded Turing Machine

Soit $f : \mathbb{N} \rightarrow \mathbb{N}$ et une machine de Turing M , déterministe ou non. M est f -time bounded si pour tout w , $|w| = n$, alors il n'existe pas de configuration telle que $(s, \#w\#) \vdash_M^{f(n)+1} (q_1, \alpha)$. En commençant, la machine fonctionne avec $f(n)$ pour chaque transition au plus. De façon similaire, on pourra définir space bounded. M est polynomially time bounded s'il existe un polynôme p tel que M est p -time bounded, *i.e.* M est f -time bounded pour un $f(n) = n^{O(1)}$. P est la classe des langages décidés par une machine de Turing déterministe polynomially bounded. NP est la classe de langages pour la version non-déterministe. EXP est la classe des langages décidés par une machine de Turing déterministe exponentially bounded $f(n) = 2^{n^{O(1)}}$

$$P \subseteq NP \subseteq EXP$$

2) Prouver qu'un langage est dans NP : vérifiable en temps polynomial

Définition alternative pour NP : on peut vérifier une solution en temps polynomial.

Soit $\mathcal{L}' \in \Sigma^*, \Sigma^*$ le langage des paires. \mathcal{L}' is polynomially bounded balanced language if there exist a polynomial p such that $u, v \in \mathcal{L}' \rightarrow |v| \leq p(|u|)$.

\mathcal{L} est NP s'il existe un tel langage \mathcal{L}' avec les propriétés suivantes :

- $\mathcal{L}' \in P$
- $\mathcal{L} = \{x : \text{il existe } y : x, y \in \mathcal{L}'\}$ On dit que y est le succinct certificate (ou solution candidate)

Exemple : les nombres composites.

Soit $\mathcal{L} = \{w \in \{0,1\}^* : w \text{ est un nombre composite}\}$.

Soit $\mathcal{L}' = \{w, a, b \in \{0,1\}^* : w = a*b\}$. Ce langage est dans P et 'a,b' est le succinct certificate.

Comme $\mathcal{L}' \in P$ et on a un succinct certificate alors $\mathcal{L} \in NP$: on peut le vérifier en temps polynomial.

On peut vérifier en temps polynomial si un nombre est composite, à partir du moment où on propose les composants. Par contre, on pense qu'il n'existe pas de résolution en temps polynomial pour les trouver.

3) Réduction

Une réduction est une fonction $\zeta : \Sigma^* \rightarrow \Sigma^*$ telle que ζ est récursif et $x \in \mathcal{L}_1 \leftrightarrow \zeta(x) \in \mathcal{L}_2$. On s'intéresse au temps nécessaire pour faire la réduction : on parle de réduction polynomiale de \mathcal{L}_1 à \mathcal{L}_2 si la réduction est faisable avec un polynomially bounded deterministic Turing machine.

\mathcal{L} est NP -hard si pour tout $\mathcal{L}' \in NP$, il existe $\zeta : \Sigma^* \rightarrow \Sigma^*$ réduction polynomiale.

$$\mathcal{L}' \rightarrow \mathcal{L}$$

Le plus dur des problèmes NP est celui pour lequel on a une réduction polynomiale de tous les autres.

\mathcal{L} est NP -complet si $\mathcal{L} \in NP$ et \mathcal{L} est NP -hard.

4) Problèmes NP-Complet : SAT et le bounded tiling

Le premier problème prouvé comme étant NP -complet par Cook est SAT (Satisfiability).

Soit $X = \{x_1, \dots, x_n\}$ un ensemble de variables booléennes, et $/X = \{/x_1, \dots, /x_n\}$. Xu/X est dit l'ensemble des littéraux (c'est l'univers des booléens). Une formule s'écrit $\alpha_1 \wedge \dots \wedge \alpha_m$, $\alpha_i = x_{i1} \vee \dots \vee x_{ik}$, où les α_i sont des clauses (une formule est une conjonction de clause quand elle est écrite sous forme normale). Une interprétation est un ensemble d'affectations de valeurs aux variables, et cela engendre un résultat vrai ou faux. On dit qu'une formule est satisfiable s'il existe une interprétation telle qu'elle est vraie.

Le problème SAT est : étant donné une formule F , est-ce que F est satisfiable ?

Ce problème est dans NP car on peut trouver un certificate pour chaque instance de SAT. On peut aussi construire une machine de Turing qui devine une interprétation et la vérifie, en utilisant l'habituel non-deterministic guessing. On parle de *generate & test paradigm*, et c'est une méthode très commune.

Etudions le problème du tiling. Soit S et $N_S = N \cap [0, S]$. Un tiling system est un tuple $\mathcal{D} = \{D, d_0, H, V\}$:

- D est le genre de tiles qu'on met (on fait du carrelage, D caractérise les carreaux)
- $d_0 \in D$ est le tile initial dans le coin inférieur gauche
- $H \in D \times D, V \in D \times D$ sont les spécifications horizontales et verticales des motifs

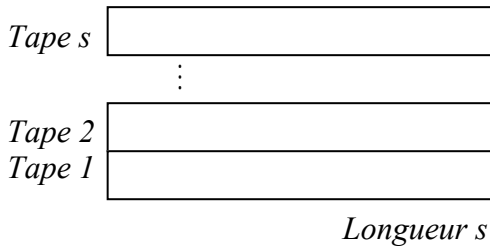
Un tiling est une fonction $f : N_S \times N_S \rightarrow D$ qui donne à chaque tile sa place, tel que $f(0,0) = d_0$ et :

- $(f(m,n), f(m+1,n)) \in H$, la restriction horizontale
- $(f(m,n), f(m,n+1)) \in V$, la restriction verticale

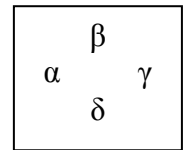
Le problème du bounded tiling : étant donné un tiling system \mathcal{D} et un tiling initial $f_0 : N_S \rightarrow D$, a-t-on un tiling $f : N_S \times N_S \rightarrow D$ qui étend f_0 ? Autrement dit : la 1^{ère} ligne est déjà carrelée, peut-on trouver une fonction qui continue le motif en accord avec les restrictions horizontales/verticales ?

Il est facile de montrer que le problème est NP : on peut construire la machine de Turing associée avec un non-deterministic guess pour f ; tester que f_0 est la ligne inférieure se fait en temps linéaire ainsi que de tester les restrictions (4 voisins à chaque fois). *Rappel : pour tout $p \in NP$ on a une machine de turing non-déterministe et polynomially time-bounded.*

Ce problème est NP complet. Prouvons qu'il est NP-hard, en montrant qu'il y a une réduction polynomiale des machine de Turing de tous les autres problèmes NP vers une instance du bounded tiling.



A chaque étape, on fabrique une tape. Comme le nombre d'étapes est polynomially bounded, un jour on finira par arrêter de rajouter. Le tiling est limité par une taille s , qui est un polynôme de la taille de l'entrée.



Un tile est labellé sur chaque côté. Deux tiles vont ensemble ssi ils ont le même label sur le côté par lequel on les joint. La machine de Turing M équivaut au système \mathcal{D} et f_0 équivaut à w ; donc un f qui étend f_0 , c'est un M acceptant w .

On va créer des blocs de bases qu'on va stocker dans notre machine. Pour chaque, on veut connaître l'état courant et vers où pointe la tête. On dénote par $\langle \text{symbol}, \text{state} \rangle$ le tile actuellement pointé.

Pour tout $a \in \Sigma$, on a :

a
a

} *Dans cet exemple, le tile pointé est le 3ème*

Pour tout $(q, a, p, b) \in \Delta, b \in \Sigma$, on a :

(b,p)
(a,q)

Quand on a le symbole a et qu'on est en q , alors on va en b avec le symbole p

Pour tout $(q, a, p, L) \in \Delta, b \in \Sigma$, on a les deux tiles :

(b,p)
Lq
b

}

a
Lq
(a,q)

On va en b avec la tête, on se déplace d'un cran à gauche

Pour tout $(q, a, p, R) \in \Delta, b \in \Sigma$, de même.

a
Rq
(a,q)

}

(b,p)
Rq
b

On a vu que f_0 est équivalent à $\#w\#$, soit la première ligne :

$\#$	w_1	\dots	w_n	$(\#,s)$	$\#$	\dots	$\#$
$\#$	w_1	\dots	w_n	$\#$	$\#$	\dots	$\#$

Quand la machine veut s'arrêter et dire non, il ne faut pas qu'on continue de se propager : on fait un arrêt du tiling. Ainsi le bloc « NN » est exclu. Cependant, le bloc « YY » est autorisé ;

On note l'équivalence par la réduction avec M accepte w ssi $\zeta(M,w)$ est une chaîne dans le langage du bounded tiling. Construire un tile se fait en temps constant. Il y a $|\Sigma|$ tiles « a » ; pour chaque (q, a, p, L) on a $|\Delta| \cdot |\Sigma| \cdot 2$ (car il faut deux tiles), de même pour (q, a, p, R) , d'où un $O(|\Delta| \cdot |\Sigma|)$ soit $O(|\Delta|)$ nombre de tiles comme la taille de l'alphabet est fixée. Le nombre de tiles est linéaire en la taille de l'entrée, donc nous avons un polynôme. Comme le Tiling System est calculable en temps linéaire, il y a une réduction polynomiale et donc c'est un problème NP-Complet.

5) Comme pourrait-on prouver que $P = NP$

Soit \mathcal{L} un problème NP-complet, alors $P = NP$ ssi $\mathcal{L} \in P$ (i.e. \mathcal{L} NP-complet et $\mathcal{L} \in P \leftrightarrow P = NP$)

Montrons que si $P = NP$ et \mathcal{L} est un problème NP-complet, alors $\mathcal{L} \in P$. C'est assez direct.

\mathcal{L} est NP complet donc $\mathcal{L} \in NP$ et comme $P = NP$ alors $\mathcal{L} \in P$.

Montrons le dans l'autre sens.

Pour tout $\mathcal{L} \in P$ on a une machine de Turing polynomially time bounded M telle que le langage décidé par M est \mathcal{L} (par définition de la classe P). Pour tout $\mathcal{L}' \in NP$ on veut construire une machine de Turing polynomially time bounded M' telle que $\mathcal{L}(M') = \mathcal{L}'$. On a $M' = M_{\zeta}M$ où M_{ζ} est la machine de Turing calculant la réduction polynomiale de \mathcal{L}' à \mathcal{L} . Or, cette machine M_{ζ} existe par définition de NP-complet.

Le même principe marche entre deux autres classes de problèmes où on ne sait pas si l'inclusion est stricte, comme $NL \subseteq P$ (c.f. assignment 4). Notons au passage que les algorithmes de la classe NL peuvent être parallélisés, tandis que si un algorithme est dans P (de façon stricte) et qu'on a plusieurs processeurs alors on pense qu'il ne peut pas en bénéficier.

6) Réductions entre problème

Rappel : si ζ_1 et ζ_2 sont des réductions polynomiales alors il en va de même pour $\zeta_1 \circ \zeta_2 = \zeta_1(\zeta_2(x))$. On associe M_1 à ζ_1 et M_2 à ζ_2 des machines de Turing déterministes, alors M_2M_1 est déterministe et polynomially bounded (on lance M_2 jusqu'à arrêt puis lancement de M_1 sur le contenu de la bande).

Les réductions polynomiales sont transitives : si on a un nouveau problème et qu'on a une réduction polynomiale vers un problème NP-complet, alors ce nouveau problème est lui aussi NP-complet. Il est très important de connaître une liste raisonnable de problèmes NP-complets pour faire ses réductions.

Montrons que SAT est NP-complet en utilisant le fait qu'on puisse le réduire au bounded tiling. On a X_{mnd} les variables booléennes telles que $f(m, n) = d$ (true ou false).

$$\triangle X_{mnd1} \vee X_{mnd2} \vee \dots \vee X_{mndk} \text{ pour } D = \{d_1, \dots, d_k\} \text{ pour tout } 0 \leq m, n < s$$

Pour tout carré, il y a au moins un tile.

$$\triangle /X_{mnd} \vee /X_{mnd'} \text{ pour tout } d \neq d', \text{ pour tout } 0 \leq m, n < s \quad \text{Il n'y a pas deux tiles sur le même carré}$$

$$\triangle /X_{nmd} \vee /X_{n(m+1)d} \text{ et } /X_{nmd} \vee /X_{(n+1)md} \text{ pour tout } (d, d') \in D^2 - V \quad \text{Restrictions horizontal/vertical}$$

Il suffit de mettre un ET logique entre toutes ces formules et on a l'équivalence. Réduction polynomiale.

Examinons le problème SAT3. Il s'agit de la même chose que SAT mais où chaque clause peut avoir au plus trois littéraux. Le modèle est $(\alpha_{11} \vee \alpha_{12} \vee \alpha_{13}) \wedge \dots \wedge (\alpha_{n1} \vee \alpha_{n2} \vee \alpha_{n3})$. On opère de réduction de SAT à SAT3 : l'idée est de simplifier les clauses d'un modèle SAT pour les exprimer comme un modèle SAT3. Exemple : on a $\alpha_1 \vee \alpha_2 \vee \alpha_3 \vee \alpha_4$. Le modèle équivalent dans SAT3 est $(\alpha_1 \vee \alpha_2 \vee y_1) \wedge (/y_1 \vee \alpha_3 \vee \alpha_4)$, tel que $y_1 = /(\alpha_1 \vee \alpha_2)$. Ainsi, si $\alpha_1 \vee \alpha_2$ est vrai alors $/y_1$ est vrai donc l'ensemble est vrai ; si $\alpha_1 \vee \alpha_2$ est faux alors y_1 est vrai et on délègue la responsabilité de rendre la formule vraie à $\alpha_3 \vee \alpha_4$ (car $/y_1$ faux i.e. neutre). D'où : $\alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_n \equiv (\alpha_1 \vee \alpha_2 \vee y_1) \wedge (/y_1 \vee \alpha_3 \vee y_2) \wedge (/y_2 \vee \alpha_4 \vee y_3) \wedge \dots \wedge (/y_{n-3} \vee \alpha_{n-1} \vee \alpha_n)$. La réduction est-elle polynomiale ? Au pire, on a trois fois plus de clauses et chaque clause se construit en temps linéaire, donc la réduction est en $O(n)$ et ainsi SAT3 est un problème NP-complet.

En revanche, $SAT2 \in P$. Par exemple, on a $(X_1 \vee X_2) \wedge (/X_1 \vee X_2) \wedge (/X_2 \vee X_1)$. Affectons $X_1 = T$ et réécrivons : il ne reste plus que X_2 , donc la combinaison $X_1 = X_2 = T$ permet de satisfaire la formule. Exemple : $(X_1 \vee X_2) \wedge (/X_1 \vee X_2) \wedge (/X_2 \vee X_1) \wedge (/X_1 \vee /X_2)$. Affectons $X_1 = T$, il reste $X_2 \wedge /X_2$; si on essaye $X_2 = F$ alors il reste $X_2 \wedge /X_2$; la formule n'est donc pas satisfaisable. Notons bien qu'on ne génère pas toutes les possibilités, ce qui serait exponentiel.

Montrons que le problème des cycles hamiltoniens est NP-complet en utilisant une réduction avec SAT. On sait que le problème est dans NP puisqu'on a une machine qui devine toutes les permutations et la vérification se fait en temps linéaire. Les arcs sont les $d_{ij} \in \{0, \infty\}$ et on veut le cycle qui passe par tout sommet une fois et une seule, donc les sommets sont les x_{ij} , $0 < i, j \leq n$ tels que $\Pi_j = i$ (le sommet i du graphe est le sommet j du cycle). On sait une instance de SAT satisfaisable ssi le graphe admet un cycle hamiltonien.

La technique consiste à énoncé en posant des clauses, et de les lier à la fin. Ainsi :

- Il doit y avoir au moins 1 sommet pour chaque position, soit $X_{1j} \vee X_{2j} \vee \dots \vee X_{nj}$, $1 \leq j \leq n$
- Tout sommet doit être visité : $X_{i1} \vee X_{i2} \vee \dots \vee X_{in}$, $1 \leq i \leq n$
- Il doit y avoir au plus 1 sommet pour chaque position. Si on a une position, alors $X_{ik} \wedge X_{jk}$ doit être faux puisque l'un des deux ne doit pas y être, autrement dit on a $\neg X_{ik} \vee \neg X_{jk}$, $1 \leq i, j, k \leq n, i \neq j$
- De même, on a $\neg X_{ki} \vee \neg X_{kj}$, $1 \leq i, j, k \leq n, i \neq j$

Ceci établit que x_{ij} représente une bijection sur l'ensemble des sommets (i.e. une permutation). Maintenant, on établit une dernière clause pour spécifier l'existence du cycle :

- Pour tout arc $(i, j) \notin G$, $\neg X_{ik} \vee \neg X_{j(k+1)}$, $1 \leq k \leq n$ et $n+1 \equiv 1$ (cyclique)

Pour les deux premières clauses on a n clauses à n littéraux, pour les autres on a n^3 clauses à nombre constant de littéraux. La réduction de $O(n^3)$ soit polynomiale. Notons que pour la dernière clause il y a peut-être un check à faire mais au pire on est en $O(n^5)$ et ça reste polynomiale quoiqu'il en soit. Montrons donc que $\zeta(G)$ la conjonction est satisfaisable ssi G admet un cycle hamiltonien :

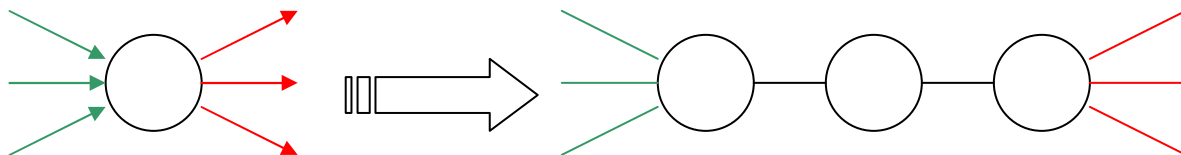
Si $\Pi = (\pi_1, \dots, \pi_n)$ représente un cycle hamiltonien, alors $\zeta(G)$ est satisfaisable.

Notons $\text{Interpretation}(X_{ij}) = T$ ssi $\Pi_j = i$. X_{ik} ou $X_{j(k+1)}$ de la dernière clause doit être faux par définition du cycle, donc la clause doit être vraie.

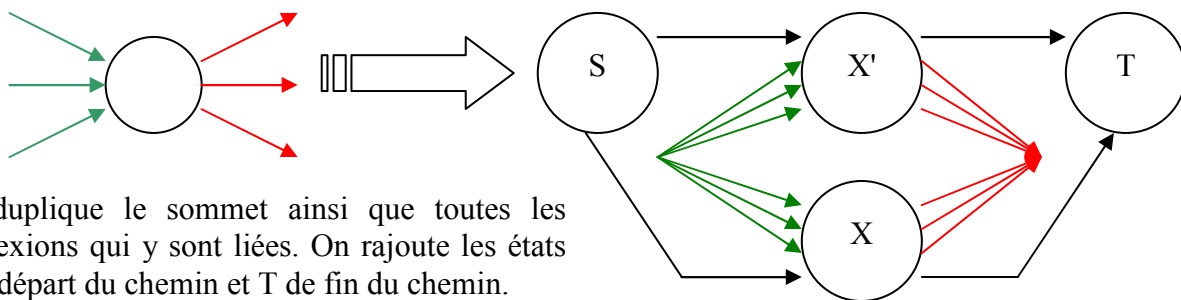
Si $\zeta(G)$ est satisfaisable alors il doit y avoir une permutation $\Pi = (\pi_1, \dots, \pi_n)$

Il existe une interprétation telle que la formule est vraie. Il y a un arc $(i, j) \in G$, $\neg X_{ik} \vee \neg X_{j(k+1)}$ est faux donc $\text{Interpretation}(X_{ik}) = \text{Interpretation}(X_{j(k+1)})$.

Montrons que trouver un cycle hamiltonien dans un graphe non-orienté est aussi un problème NP-complet. Il nous suffit de transformer le graphe orienté de la façon suivante, puisqu'on ne peut passer qu'une fois par un sommet :



De même, montrons que trouver un chemin hamiltonien dans un graphe orienté est aussi NP-complet.



On duplique le sommet ainsi que toutes les connexions qui y sont liées. On rajoute les états S de départ du chemin et T de fin du chemin.

Il y a un cycle hamiltonien passant par x . Donc, on commence notre chemin à S, on passe par X et comme c'est un cycle on le fait finir sur X' qui termine le chemin en passant sur T.

7) Problèmes ouverts

P est clôt par complément. Pour NP, on ne sait pas... On note son complément $CO-NP = \{\mathcal{L} \mid \mathcal{L} \in NP\}$; actuellement, on pense plutôt que NP n'est pas clôt par complément, *i.e.* que $CO-NP \neq NP$. L'autre suspicion serait que si $\mathcal{L} \in NP$ et $\mathcal{L} \in CO-NP$ alors \mathcal{L} n'est pas complet ; une fois encore, on ne sait pas.

Une variante de la factorisation de nombres s'énonce de la façon suivante : étant donné $n \in \mathbb{N}$ et $m \leq n$, y a-t-il un facteur de n plus petit que m ? Ce problème est dans NP, et c'est justement la base de bons nombres de techniques de cryptographie. Ce problème est également dans CO-NP. On pense qu'il est en dehors de P, en dehors de NP-complet et en dehors de CO-NP-Complete; il est « quelque part au milieu ».

8) Le principe du Model Checking (preuve de correction)

Etant donnés deux expressions régulières R_1 et R_2 , est-ce qu'elles génèrent le même langage ? Regardons le problème opposé : a-t-on $\mathcal{L}(R_1) \neq \mathcal{L}(R_2)$. On peut trouver un certificate : n'importe quelle chaîne générée par R_1 mais pas par R_2 . Le certificate est-il un succinct ? Il n'y a pas de limite évidente, et en fait on ne sait même pas si le problème est dans NP...

Examinons alors un problème plus simple. On se contraint à n'avoir que des expressions régulières qui n'utilisent pas la fermeture de Kleene. On a donc : pour tout $x \in \mathcal{L}(R)$, R une *-free R.E., $|x| \leq |R|$. Ici, le certificate a une borne et il devient un succinct. Ce problème là est donc dans NP, et il est aussi NP-complet comme on le montre avec une réduction depuis SAT.

L'entrée est $C = c_1 \wedge c_2 \wedge \dots \wedge c_m$ sur $X = \{X_1, \dots, X_n\}$.

En sortie : R_1, R_2 tels que $\mathcal{L}(R_1) \neq \mathcal{L}(R_2)$ ssi C est satisfaisable.

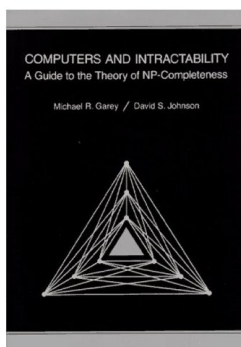
On a $R_1 = (0 \text{ u } 1)(0 \text{ u } 1) \dots (0 \text{ u } 1)$, n fois. On a donc $\mathcal{L}(R_1) = \{0, 1\}^n \equiv$ affectations de X qui rend C vrai.

On a $R_2 = \alpha_1 \cup \alpha_2 \cup \dots \cup \alpha_m$ tel que $\alpha_i = \alpha_{i1}\alpha_{i2} \dots \alpha_{in}$, avec $\alpha_{ij} =$
0 si X_j apparaît dans C_i
1 si $\neg X_j$ apparaît dans C_i
{0 u 1} sinon

On a donc $\mathcal{L}(\alpha_i)$ le langage des interprétations qui ne satisfont pas C_i , et ainsi $\mathcal{L}(R_2)$ est l'ensemble de toutes les interprétations qui ne satisfont pas certains C_i . Donc $\mathcal{L}(R_2) = \{0, 1\}^n$ si C est insatisfaisable
 $\subsetneq \{0, 1\}^n$ sinon

$\mathcal{L}(R_2) \subsetneq \{0, 1\}^n$ si C est satisfaisable car il y a au moins une des interprétations qui satisfait C et $\mathcal{L}(R_2)$ est l'ensemble des interprétations qui ne satisfont pas C : ça ne peut donc pas être $\{0, 1\}^n$ tout entier puisqu'une des interprétations satisfait.

Ainsi C satisfaisable ssi $\mathcal{L}(R_1) \neq \mathcal{L}(R_2)$, et donc la comparaison d'expressions régulières sans la fermeture de Kleene est un problème NP-complet. On en déduit que le problème en autorisant la fermeture de Kleene est plus compliqué encore, et on sait qu'il est « au moins » NP-complet.



Pour continuer l'étude des problèmes NP-complets, on recommande l'ouvrage suivant : Michael R. Garey and David S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, Freeman, 1991. Call number QA 76.6 .G35 1991.