

MATH.4.2 Automates et langages

Florent Madelaine

mars 2023

Table des matières

I	Modèles	5
1	Que veut-on calculer ?	7
1.1	Modèles de calcul	8
1.2	Découverte de JFLAP	9
2	Automates	11
2.1	Définition	11
2.2	Décrire les langages	17
2.3	Automates équivalents	18
2.4	Déterminisation	20
2.5	Lien entre automates et expressions régulières	21
2.5.1	Expression régulière vers automate	22
2.5.2	Automate vers Expression régulière	28
2.5.3	Conclusion	32
2.6	Lemme de la pompe	33
3	Turing	39
3.1	Turing avec JFLAP	40
3.2	Définition	42
3.3	Robustesse	43
II	Comparer	45
4	Décidabilité et Indécidabilité	47
4.1	Définitions.	47
4.2	Indécidabilité (argument dénombrement)	48
4.3	Machine de Turing Universelle.	49
4.4	Indécidabilité (problème de l'arrêt)	49
4.5	autres problèmes indécidables	50

5	Complexité	51
5.1	Ressources, dont le temps.	51
5.2	Mesurer le temps	51
5.3	classes de complexité robustes	52
5.4	Vers la notion de réduction.	54
5.5	NP-complet	54
5.6	Complément complexité	58

Première partie
Jouer avec les modèles

Chapitre 1

Que veut-on calculer ?

Le cas des fonctions discrètes $f: \mathbb{N} \rightarrow \mathbb{N}$ est essentiellement celui qui est étudié.

Le cas des fonctions indicatrices est assez général pour illustrer le domaine avec peu de perte de généralité. Autrement dit on cherche à décider des ensembles d'entiers.

En pratique, comme on va travailler sur des modèles de calcul pour lesquels on pourra mesurer de manière fine les ressources utilisées (temps, espace) on va devoir expliciter comment les entrées sont codées. En pratique, ce sera des mots.

Un peu de vocabulaire.

- *Alphabet* : Σ ensemble fini fixé de symboles.
- *Entrée* : un mot comportant un nombre fini de symboles de Σ .
- *Problème de décision (langage)* : ensemble de mots.

Exemple. On fixe $\Sigma = \{0,1\}$. On considère le problème consistant à reconnaître les mots qui sont des *palindromes*, à savoir qui sont identiques quel que soit le sens de lecture.

- 101101 est un mot du langage des palindromes.
- 100 n'est pas un mot de ce langage.

Autres exemples de langages.

1. L'ensemble des mots sur $\{0,1\}$ représentant un entier pair.
2. L'ensemble des mots sur $\{0,1\}$ dont la première et dernière lettre sont identiques.
3. L'ensemble des mots sur $\{0,1\}$ dont le nombre de 0 et de 1 est égal.
4. L'ensemble des mots sur $\{0,1\}$ représentant un nombre premier.

1.1 Modèles de calcul.

Nous allons en voir un en détail : l'automate (fini). Nous allons évoquer rapidement le modèle le plus général, qui correspond à ce qu'on peut calculer en général avec n'importe quel langage de programmation : la machine de Turing.

Avec JFLAP, un outil pédagogique pour faciliter la compréhension de modèles de calculs, vous pouvez en découvrir d'autres. Il existe en particulier un modèle intermédiaire naturel : l'automate à pile (*push down automata*). *Grosso Modo* ce modèle correspond bien aux calculs dont on a besoin pour faire un compilateur, au sens où il permet de faire des calculs sur des expressions parenthésés¹.

Ici on compare les modèles en terme de langages reconnus. On verra que tous les langages reconnus par un automate, sont aussi reconnaissables par une machine de Turing. Par contre l'inverse n'est pas vrai : le troisième exemple ci-dessus est reconnu par une machine de Turing mais aucun automate ne reconnaît ce langage.

En effet un automate a une mémoire finie et intuitivement il ne pourra pas compter au delà d'une certaine valeur.

Comparer des modèles de calculs en terme de leur expressivité mais aussi d'autres propriétés est typique en informatique. On trouve ce genre de questions dans des travaux précurseurs plutôt en mathématiques au début du XXème siècle (Turing, Church, etc) puis en *théorie des langages* à partir de la seconde moitié du XXème siècle (Schützenberger, Chomsky etc). Plus tard, lorsque les ordinateurs existent et sont plus accessibles les chercheurs se penchent de manière systématique sur des aspects liés à l'efficacité des méthodes de calcul existantes, sur ce qu'on appelle maintenant *la complexité algorithmique*. Il s'agit en fait de choses plus anciennes qui existent depuis que le calcul existe, et cette tentative d'optimisation naturelle se retrouve dans de nombreuses

Étudier La notion d'expressivité n'est pas cantonnée à l'informatique fondamentale, on retrouve aussi ses questions en *bases de données* quand il s'agit de savoir par exemple si un langage de requêtes comme le SQL permet d'obtenir des réponses à des questions très concrètes qu'on souhaite avoir étant donné une base de données relationnelle.

1. Compiler veut dire traduire un programme en jargon informatique. Le plus souvent un compilateur analyse un programme dans un langage de programmation de haut niveau, par exemple du C, pour produire, souvent en plusieurs étapes, un programme de bas niveau compréhensible par le processeur, typiquement de l'assembleur.

1.2 Découverte de JFLAP

JFLAP² est un logiciel libre qui permet de faciliter l'enseignement de modèles théoriques de calcul comme celui des automates. Le J de JFLAP signifie Java, qui est un langage de programmation populaire dont l'intérêt principal est qu'il utilise une *machine virtuelle* rendant très portable les applications écrites dans ce langage.

Sur les machines virtuelles vous pouvez démarrer JFLAP en ouvrant un terminal à l'emplacement du JAR et en tapant.

```
java -jar JFLAP.jar
```

Alternativement, vous pouvez installer java et sauver le fichier JFLAP.jar sur votre ordinateur personnel. Le lien vers le fichier est disponible sur la page web du cours. Il faut installer java au préalable selon la méthode préconisée pour votre système d'exploitation et autoriser l'exécution du fichier.

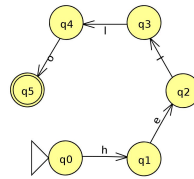
Prise en main : simuler

1. Lancez JFLAP et ouvrez l'automate sauvegardé sur le git nommé TP2NDA.jff.
2. Allez dans le menu **Input** et choisissez **Step by State**. Entrez le mot **babb** et observez la simulation des exécutions parallèles étape par étape en appuyant sur **step**.
3. Testez ensuite avec le mot **baaa**.
4. Relancez les tests ci-dessus. Cette fois essayez les autres options.
 - **Reset** permet de revenir au début du calcul
 - **Trace** après sélection en bas d'une exécution permet de montrer cette exécution depuis le début.
 - **Remove** élimine l'exécution sélectionnée
 - **Freeze/Thaw** permet de mettre en pause ou de reprendre une exécution (on perd la synchronisation)
5. Vous pouvez aussi tester sans les étapes avec **Fast Run** voir plusieurs entrées en même temps avec **Multiple Run** (tous les deux dans le menu **Input**). Faites le par exemple pour les deux mots précédents.
6. Pour information, **Step with closure** fait la même chose que **step by State** sauf pour les automates ayant des transitions avec le mot-vide (mais on ne les considère pas dans notre cours).

Hello world

Votre but est de créer l'automate dessiné ci-dessous.

2. <http://www.jflap.org/>



1. Cliquez sur **File** puis **New**.
2. JFLAP permet de gérer différents modèles de calcul. Nous allons nous en tenir aux automates finis. Cliquez sur **Finite Automaton**. Vous avez accès à une nouvelle fenêtre en mode éditeur.
 - Cliquez sur le petit cercle avec un q à l'intérieur, puis sur le canevas pour ajouter des états.
 - Les deux derniers boutons permettent de faire du undo/redo.
 - La flèche fine permet d'ajouter une transition entre 1 ou 2 états (ne pas oublier de mettre une lettre).
 - La flèche plus épaisse à gauche permet de sélectionner, déplacer, éditer les propriétés des différents éléments (par exemple de faire un état initial).
 - La tête de mort permet de détruire des états ou des transitions.

Chapitre 2


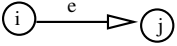
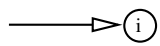

Les automates finis

Nous allons maintenant nous concentrer sur un modèle de machine qui a une **mémoire finie** et qui va lire le mot une seule fois. Pendant le calcul / la lecture d'un mot la mémoire de l'automate ne contiendra que l'état actuel.

2.1 Définition

L'automate est une machine très simple qu'on représente par un graphe orienté dont les arcs sont étiquetées par des lettres. Un sommet représente un état de la machine ; et, un arc représente une transition. Certains états sont spéciaux : l'état initial (début du calcul) et le ou les états finaux (état(s) acceptants).

Notation

État	Transition	État initial	État(s) acceptant(s)
			

Les objets qu'on manipule En plus des lettres de l'alphabet qui permettent de construire des mots, et des ensembles de mots qu'on appelle un langage, on considère :

- **le mot vide** ε (qui ne contient aucune lettre) parfois noté λ .
- Σ^* (l'ensemble de tous les mots finis composés de lettres de l'alphabet)

Un langage \mathcal{L} est un ensemble de mots qui peut être fini ou non. On peut aussi écrire $\mathcal{L} \subseteq \Sigma^*$. Quelques cas particuliers : Σ^* (tous les mots), \emptyset (aucun mot), $\{\varepsilon\}$ (uniquement le mot vide).

Le calcul

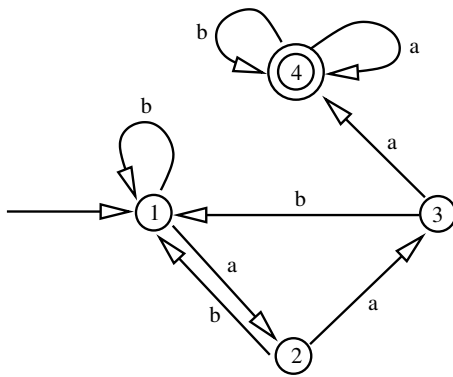
- Donnée : w
- Début : État initial
- Lire w de gauche à droite lettre par lettre en suivant les transitions
- Fin : mot terminé / pas de transition possible
- Réponse : Accepté / Refusé

Le langage de l'automate

- Ensemble des mots acceptés par l'automate
- On parle aussi du langage (ou des mots) reconnu(s) par l'automate.

Exemple

à compléter



- $abaab$
- $aababaaa$
- $baabaaaaabaab$
- Accepter/Refuser
- Cas général ?

Autre manière de donner un automate

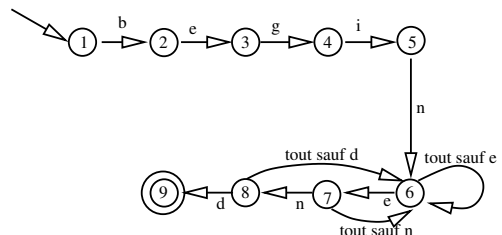
- Alphabet $\Sigma = \{a, b\}$
- Ensemble (fini) d'états $Q = \{1, 2, 3, 4\}$
 - État initial $q_0 = 1$
 - État(s) acceptant(s) $Q_A = \{4\}$

• **Table de transition δ**

	a	b
1	2	1
2	3	1
3	4	1
4	4	4

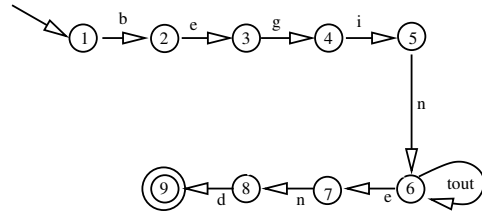
État de rebut

- Rebut pas obligatoire
- Si pas de rebut, le calcul peut se bloquer avant la fin
 - Le mot est alors refusé
- Si pas de rebut, la table de transition peut contenir des cases vides



Non-déterminisme

- Plusieurs flèches étiquetées par la même lettre partent d'un même état
- Plusieurs calculs possibles pour un même mot
 - Un mot est accepté quand il est accepté par **au moins un calcul**
- La table de transition peut contenir plusieurs états par case



Vocabulaire

- * État initial, acceptant, les deux ou ni l'un ni l'autre
- * Transition
- * Mot accepté par l'automate
- * Mot rejeté par l'automate
- * Langage de l'automate
- * Table de transition
- * État de rebut
- * Non déterminisme

Programmer avec des automates

Vous pouvez résoudre ces exercices sur papier ou plus simplement avec JFLAP. Dans ce second cas, il est plus facile de tester votre réponse avec un jeu de test adapté (des mots acceptés et des mots refusés).

Exercice 1. Dessinez les automates qui reconnaissent : (pour l'alphabet $\{a,b\}$)

1. les mots de 2 lettres ;
2. les mots contenant exactement 2 a ;
3. les mots contenant 2 a consécutifs ;
4. les mots commençant par b ;
5. les mots se terminant par bab ;
6. le langage réduit au mot vide ;
(pour l'alphabet $\{a,b,c\}$)
7. les mots qui contiennent $aabcc$;
8. les mots de taille multiple de 3 ;
9. les mots de la forme $abcabcabc\dots$;
10. les mots dont la première lettre est égale à la dernière.

Exercice 2. Construire un automate d'alphabet $\{a,b\}$ correspondant à l'expression régulière¹

$$a(a+b)^*b$$

Un peu plus technique : le calcul modulaire en base 2

L'objectif est de construire des automates qui trient les mots selon le reste modulaire des nombres qu'ils représentent.

Calcul modulaire. Le reste modulaire (ou modulo en jargon arithmétique) est simplement le reste de la division entière que vous avez vu à l'école primaire. Si on calcule modulo 2, on souhaite juste savoir si un nombre est pair ou impair. Quand on calcule modulo 3, on veut juste savoir si le nombre est de la forme 3 fois quelque chose plus un reste qui est soit nul soit un soit deux. En pratique vous pouvez faire des sommes et des produits comme d'habitude sauf que vous pouvez remplacer le modulo avec lequel vous travaillez (ce sera 3 ci-dessous) par 0.

Représentation en base 2. On note n un nombre entier et \bar{n} une **représentation binaire** de ce nombre, avec éventuellement des zéros à gauche. Par exemple, si $n =$ cinq, on peut avoir $\bar{n} = 101$ ou bien $\bar{n} = 000101$, etc. On a alors $2\bar{n} = \bar{n}0$ et $2\bar{n}+1 = \bar{n}1$. Autrement dit, ajouter un 0 à droite d'un nombre écrit en binaire revient à multiplier ce nombre par 2, tandis qu'ajouter un 1 revient à le multiplier par 2 puis ajouter 1 au résultat. Par exemple 1010 est une représentation binaire de dix (deux fois cinq), et 0001011 est une représentation binaire de onze (deux fois cinq plus un).

Calcul modulaire en base 2. Si on connaît le reste pour le nombre codé par les bits d'un mot w , on peut facilement en déduire le reste pour le nombre codé par le mot w suivi d'un 0. Ce nombre est deux fois le précédent.

Ainsi si w code un nombre de la forme 3 fois quelque chose plus 1, disons $3k+1$, on sait que $w0$ code le double de ce nombre (on est en base 2, en base 10 on multiplierait par 10) et donc que $w0$ code un nombre de la forme $2(3k+1) = 6k+2 = 3(2k)+2$. Autrement dit $w0$ code un nombre de la forme trois fois quelque chose plus 2.

En jargon on dit que ce nombre codé par w – disons n – est congru à 1 modulo 3 et on écrit $n \equiv 1 \pmod{3}$. Ajouter un 0 à droite de w , signifie qu'on multiplie n par 2; On calcule alors que $2n \equiv 2 \pmod{3}$.

Regardons maintenant le cas où le mot binaire w code pour un nombre n tel que $n \equiv 2 \pmod{3}$. Le mot binaire $w1$, code pour $2n+1$ et on peut calculer que $2n+1 \equiv 2(2)+1 \pmod{3} \equiv 5 \pmod{3} \equiv 3+2 \pmod{3} \equiv 0+2 \pmod{3} \equiv 2 \pmod{3}$. Ici ci dessus notez comment on remplace 3 par 0 car on calcule modulo 3.

1. Nous allons voir en détail les expressions régulières en § 2.2.

Si vous êtes allergique à cette nouvelle notation et que vous perdez l'intuition, vous pouvez travailler avec la notation usuelle comme tout à l'heure : $n \equiv 2 \pmod{3}$ veut dire $n = 3k + 2$. Donc $2n + 1 = 2(3k + 2) + 1 = 6k + 2(2) + 1 = 6k + 3 + 2 = 3(k + 1) + 2$. On a donc bien que le mot binaire $w1$ qui code pour le nombre $2n + 1$ est de la forme 3 fois quelque chose plus 2.

De manière similaire si le nombre représenté par w est divisible par trois, donc de la forme $3k$ alors $w0$ représente le double de ce nombre qui est de la forme $6k = 3(2k)$ qui est aussi divisible par trois. Avec cette même hypothèse, $w1$ représente $6k + 1 = 3(2k) + 1$, un nombre dont le reste par la division par trois vaut 1.

De cette manière, on va donc pouvoir mécaniquement compléter le tableau ci-dessous.

n	$2n$	$2n + 1$
$\equiv 0 \pmod{3}$	$\equiv 0 \pmod{3}$	$\equiv 1 \pmod{3}$
$\equiv 1 \pmod{3}$	$\equiv 2 \pmod{3}$	
$\equiv 2 \pmod{3}$		$\equiv 2 \pmod{3}$

à compléter

Vers un automate. Les automates « modulo 3 » auront trois états correspondant aux trois restes modulo trois possibles. À partir du tableau précédent, en passant aux représentations binaires des entiers, complétez la table de transition ci-dessous :

	0	1
<i>zéro</i>	<i>zéro</i>	<i>un</i>
<i>un</i>	<i>deux</i>	
<i>deux</i>		<i>deux</i>

à compléter

Exercice 3. Construisez un automate déterministe d'alphabet $\{0,1\}$, reconnaissant parmi les mots binaires rencontrés, ceux qui représentent des entiers congrus à 1 modulo 3. Indications : *zéro* est l'état initial, *un* est le seul état acceptant. La table de transition est ci-dessus.

Vous pouvez utiliser JFLAP pour tester votre design.

Exercice 4. Construisez un automate avec JFLAP qui accepte les multiples de 9 (les nombres sont représentés sur un alphabet binaire).

Exercice 5. Construisez un automate avec JFLAP qui accepte les mots binaires représentant des entiers n tels que :

$$\begin{cases} n \not\equiv 1 \pmod{3} \\ n \equiv 2 \pmod{5} \end{cases}$$

Vocabulaire

- ★ reste modulaire (modulo)
- ★ représentation binaire
- ★ Calcul modulaire
- ★ Automate pour le calcul modulaire

2.2 Décrire les langages

La description d'un langage en français est le plus souvent ambiguë. On a besoin d'une façon plus mathématique de décrire les langages des automates, appelée « les expressions régulières ».

On utilise pour cela trois opérations :

- La **concaténation** L_1L_2 : un mot du langage L_1 suivi d'un mot du langage L_2
Exemple. $L_1 = \{aa,bb\}$ et $L_2 = \{ab,bba,b\} \rightsquigarrow L_1L_2 = \{aaab,aabba,aab,bbab,bbbba,bbb\}$
- La **réunion** L_1+L_2 : un mot du langage L_1 ou bien du langage L_2
Exemple. $L_1 = \{aa,bb\}$ et $L_2 = \{ab,bba,b\} \rightsquigarrow L_1+L_2 = \{aa,bb,ab,bba,b\}$
- L'**étoile** L^* : concaténation d'un nombre quelconque (peut-être nul) de mots du langage L
Exemple. $L = \{aa,bb\} \rightsquigarrow L^* = \{\varepsilon,aa,bb,aaaa,aabb,bbba,bbbb,aaaaaa,aaaabb,aabbaa,aabbbb,\dots\}$

Remarques.

- Langage composé d'un seul mot : $\{abba\}$ est noté $abba$
- Langage fini : $\{ab,bba,b\}$ est noté $ab+bba+b$
- Langage composé de tous les mots : Σ^* est aussi noté $(a+b)^*$

Expression régulière représentant un langage donné

(pour l'alphabet $\{a,b\}$)

- Le langage constitué des mots qui contiennent aa ou bb
 $(a+b)^*(aa+bb)(a+b)^*$
- Le langage constitué des mots qui commencent par aa et finissent par bb
 $aa(a+b)^*bb$
- Le langage constitué des mots qui commencent par aa ou finissent par bb
- Le langage constitué des mots qui ne contiennent pas aa (attention pas simple)
- Sur l'alphabet $\{a,b,c\}$, le langage constitué des mots qui commencent par aa et finissent par bb

à compléter

Remarque. On peut montrer que les langages qu'on peut décrire par une expression régulière correspondent exactement aux langages reconnus par les automates. Le passage de l'un à l'autre peut se faire mécaniquement. JFLAP illustrent ses constructions.

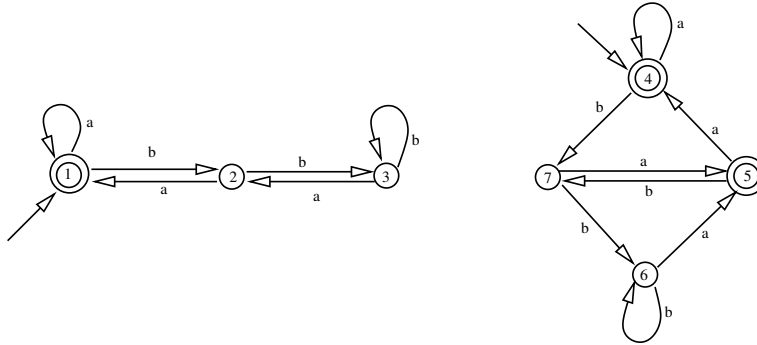
Vocabulaire

- ★ Expression régulière
- ★ Concaténation (notation $.$ ou rien)
- ★ Réunion (notation $+$)
- ★ Étoile (notation $*$)

2.3 Automates équivalents

Le modèle des automates est assez simple pour permettre de résoudre des questions **sur les programmes des automates**. En particulier on peut décider si deux automates sont équivalents.

Question : Les automates \mathcal{A} et \mathcal{A}' acceptent-ils le même langage ?



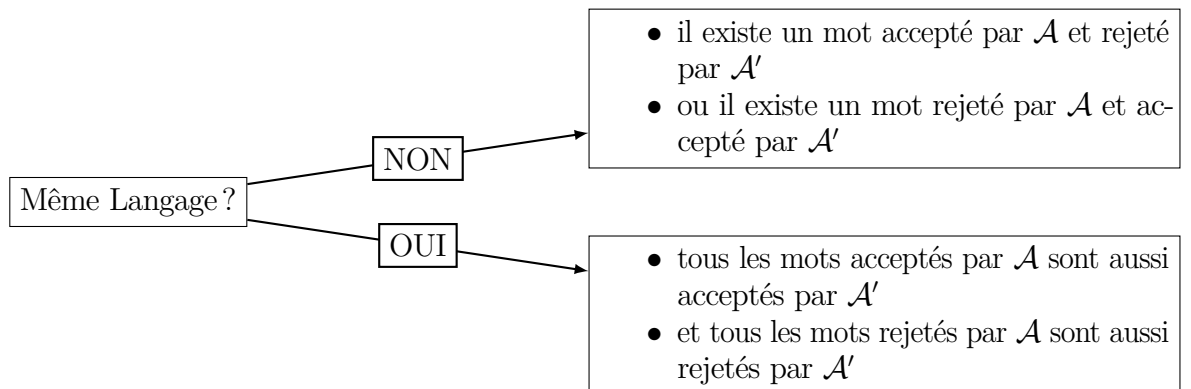
Définition. Deux automates qui acceptent le même langage s'appellent des automates **équivalents**

Idée générale pour montrer l'équivalence

- Pour prouver que \mathcal{A} et \mathcal{A}' ne sont pas équivalents, il suffit de trouver un mot accepté par l'un et rejeté par l'autre
- Pour prouver que \mathcal{A} et \mathcal{A}' sont équivalents, on doit montrer qu'ils acceptent exactement les mêmes mots (ce qui paraît plus difficile)

Exemple Revenons à notre exemple précédent. Le mot bba est rejeté par \mathcal{A} et accepté par \mathcal{A}'

Conclusion : \mathcal{A} et \mathcal{A}' **ne sont pas** équivalents



Méthode pour les automates déterministes

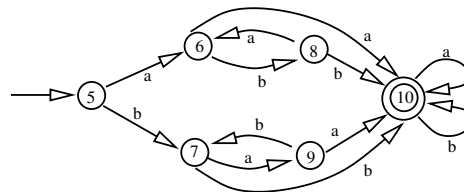
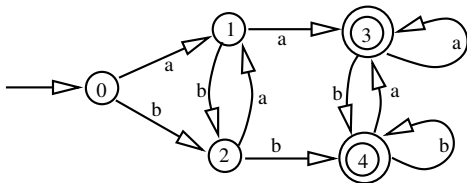
- On suit des chemins "parallèles" dans les deux automates **déterministes** \mathcal{A} et \mathcal{A}' .
- Si on rencontre un couple d'états (q, q') avec q acceptant et q' non acceptant (ou le contraire), alors on a trouvé un mot w accepté par \mathcal{A} et rejeté par \mathcal{A}' (ou le contraire). Donc \mathcal{A} et \mathcal{A}' ne sont pas équivalents.
- Il n'y a qu'un nombre fini de couples d'états (q, q') , donc le calcul se termine au bout d'un temps fini.
- Si le calcul se termine sans avoir trouvé un mot w accepté par \mathcal{A} et rejeté par \mathcal{A}' (ou le contraire), alors les deux automates \mathcal{A} et \mathcal{A}' sont équivalents, c.-à-d. ils acceptent le même langage.

Mise en œuvre. On présente la méthode en construisant la table de transition d'un automate « bi-processeurs » qui simule simultanément et de manière synchrone les deux automates (NB. on souligne un état acceptant).

	a	b
$(\underline{1}, \underline{4})$	$(\underline{1}, \underline{4})$	$(2, 7)$
$(2, 7)$	$(\underline{1}, \underline{5})$	$(3, 6)$
$(\underline{1}, \underline{5})$	$(\underline{1}, \underline{4})$	$(2, 7)$
$(3, 6)$	$(\underline{2}, \underline{5})$...

En remontant le calcul, on s'aperçoit que le mot bba est rejeté par \mathcal{A} et accepté par \mathcal{A}'

Autre exemple.



	a	b
$(0, 5)$	$(1, 6)$	$(2, 7)$
$(1, 6)$	$(\underline{3}, \underline{10})$	$(2, 8)$
$(2, 7)$	$(1, 9)$	$(\underline{4}, \underline{10})$
$(\underline{3}, \underline{10})$	$(\underline{3}, \underline{10})$	$(\underline{4}, \underline{10})$
$(2, 8)$	$(1, 6)$	$(\underline{4}, \underline{10})$
$(1, 9)$	$(\underline{3}, \underline{10})$	$(2, 7)$
$(\underline{4}, \underline{10})$	$(\underline{3}, \underline{10})$	$(\underline{4}, \underline{10})$

Le calcul se termine sans trouver de mot qui sépare les deux automates, donc ils sont **équivalents**.

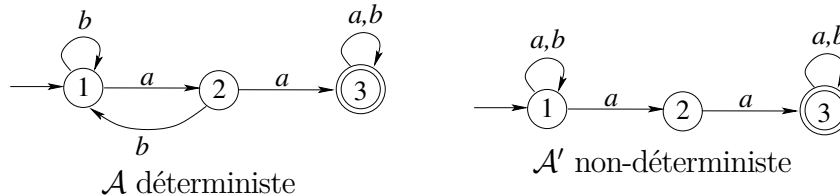
Vocabulaire

- ★ Automates équivalents
- ★ Méthode pour vérifier si 2 automates déterministes sont équivalents

2.4 Déterminisation

Il est souvent beaucoup plus simple de donner un automate non-déterministe. Par exemple, lorsqu'on souhaite décrire un automate reconnaissant les mots contenant deux a consécutifs ($\sigma^*(aa)\sigma^*$). L'automate non déterministe naturel « devine » qu'il va lire deux a consécutifs au moment adéquat. On pourrait penser que le non déterminisme permet de décider des langages qui ne sont pas décidables par des automates déterministes. On va voir qu'il n'en est rien et que les deux modèles ont la même expressivité, puisqu'on peut transformer un automate non-déterministe en automate déterministe qui décide le même langage. C'est ce qu'on appelle la déterminisation.

Définition. Un automate est **déterministe** si pour tout état et toute lettre de l'alphabet, il y a au plus une transition étiquetée par cette lettre qui part de cet état. Sinon, c'est un automate **non-déterministe**.



Remarque. L'automate non-déterministe \mathcal{A}' reconnaît le langage $L = \{\text{mots contenant deux } a \text{ consécutifs}\}$.

L'automate déterministe \mathcal{A} reconnaît le même langage L .

Question. Existe-t-il toujours un automate déterministe qui accepte le même langage qu'un automate non-déterministe donné ?

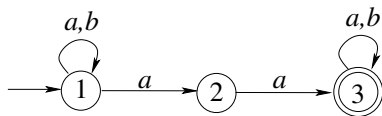
Réponse. À partir d'un automate non-déterministe qui reconnaît un langage L , on peut toujours calculer un automate **déterministe** qui reconnaît le même langage L .

Idée générale

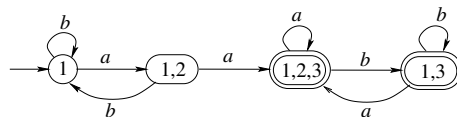
L'automate déterministe que l'on va construire sera composé de *super-états* qui vont *simuler* des ensembles d'états de l'automate initial. La méthode utilisée est la suivante :

- le *super-état* initial est formé de l'état initial ;
- le *super-état* S' issu d'un *super-état* S en appliquant une transition t est formé de l'ensemble des états obtenus en appliquant la transition t à partir de chacun des états formant S ;
- le ou les *super-états* acceptants sont les *super-états* contenant au moins un état acceptant.

Mise en œuvre Intuitivement, on construit en fait une sorte d'automate « multi-processeurs ». Un super-état représente l'ensemble des états dans lesquels l'automate multi-processeur se trouve simultanément.



<i>super-états</i>	<i>a</i>	<i>b</i>
$\rightarrow\{1\}$	$\{1,2\}$	$\{1\}$
$\{1,2\}$	$\{1,2,3\}$	$\{1\}$
$\{1,2,3\}$	$\{1,2,3\}$	$\{1,3\}$
$\{1,3\}$	$\{1,2,3\}$	$\{1,3\}$



Remarques.

- Les automates non-déterministes sont aussi naturels et acceptables que les automates déterministes
- Mais il arrive qu'on soit obligé de déterminer un automate (par exemple avant d'utiliser la méthode "automates équivalents"), c'est pourquoi vous devez savoir le faire
- Les états de l'automate déterministe obtenu correspondent à des ensembles d'états de l'automate de départ, il peut donc arriver qu'ils soient très nombreux (jusqu'à 2^n , si n est le nombre d'états de l'automate non-déterministe)

Vocabulaire

<ul style="list-style-type: none"> ★ Automate déterministe et non-déterministe ★ Méthode pour déterminer un automate (construire un automate déterministe équivalent)

2.5 Lien entre automates et expressions régulières

Nous allons étudier dans cette partie le lien assez exceptionnel entre le modèle des automates finis et celui des expressions régulières. Exceptionnel car le fait de pouvoir passer entre programme (l'automate) et spécification (expression régulière) n'est pas possible en général.

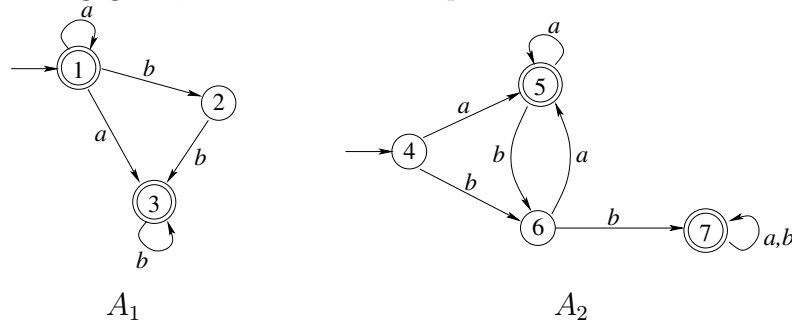
2.5.1 Expression régulière vers automate

On va procéder récursivement sur l'expression régulière depuis laquelle on souhaite synthétiser un automate.

Le cas de base : il est très simple de trouver un automate acceptant les lettres (par exemple l'automate qui accepte le mot a et rien d'autre).

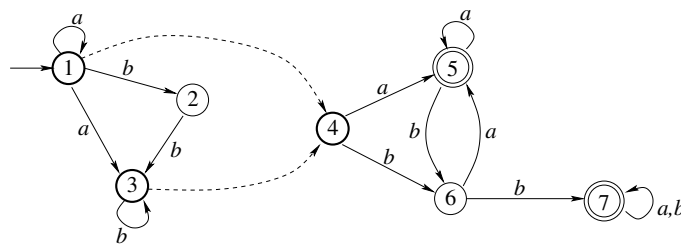
Le cas de récurrence : il faut être capable de réaliser étant donné un automate A_1 reconnaissant L_1 et un automate A_2 reconnaissant L_2 , un automate correspondant à la concaténation $L_1.L_2$, et un automate correspondant à la conjonction $L_1 + L_2$, et à l'étoile L_1^* .

Concaténation d'automates Soient A_1 un automate permettant de reconnaître le langage L_1 , et A_2 un automate permettant de reconnaître le langage L_2 .



Pour reconnaître le langage L_1L_2 , l'automate A à construire doit alors être conçu comme la *concaténation de A_1 et de A_2* .

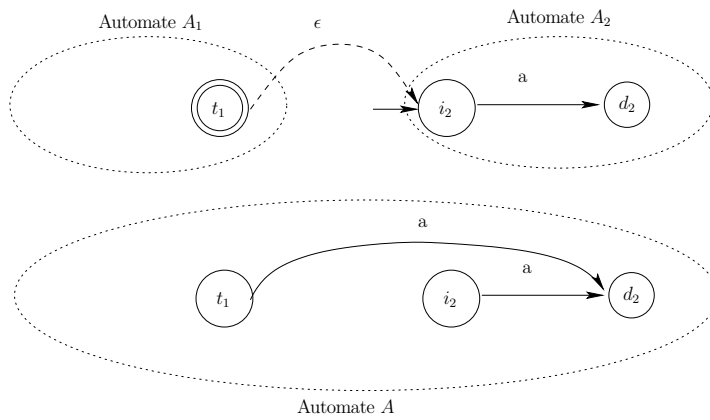
On aimerait pouvoir réaliser cette concaténation en *reliant les états acceptants de A_1 à l'état initial de A_2 par des transitions vides* (sans étiquette, ou en lisant le mot vide ϵ).



Mais dans notre modèle, un automate ne peut pas avoir des transitions vides, donc *il faudra s'en débarrasser par la suite*.

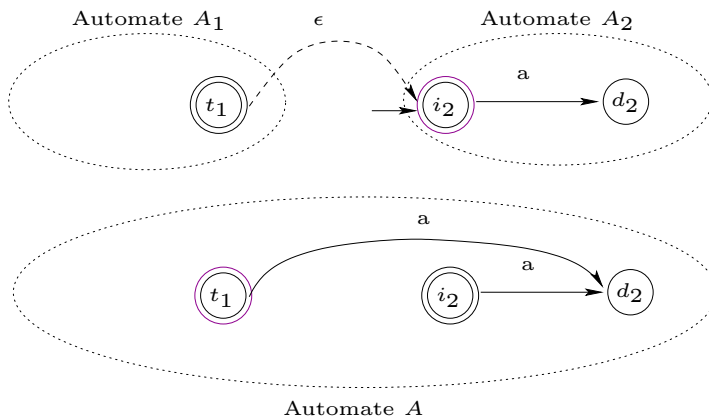
Remplacement des transitions vides : idée générale Appelons t_1 un état terminal de A_1 et i_2 l'état initial de A_2 .

On ajoute $t_1 \xrightarrow{a} d_2$ si il existe $i_2 \xrightarrow{a} d_2$ dans A_2 :



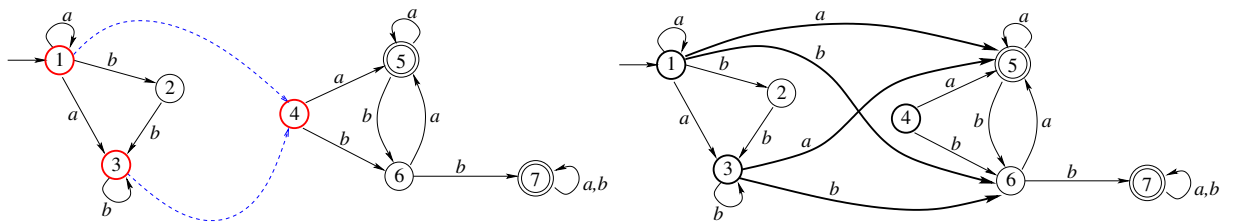
Pourquoi ? Car passer de t_1 à i_2 sans rien faire puis de i_2 à d_2 en lisant un a est équivalent à aller directement de t_1 à d_2 en lisant un a

De plus, t_1 sera terminal dans A si i_2 est terminal :



Pourquoi ? Car passer de t_1 à i_2 sans rien faire, puis sortir directement de A_2 est équivalent à sortir directement de A par t_1 .

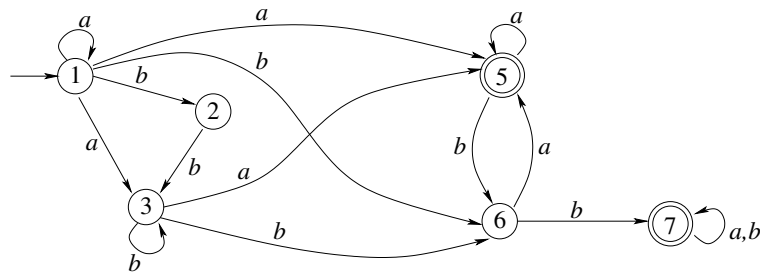
Exemple



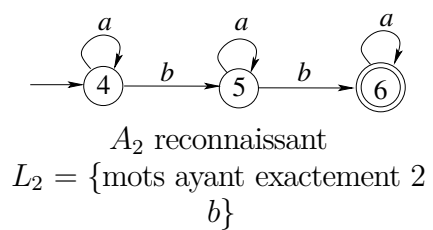
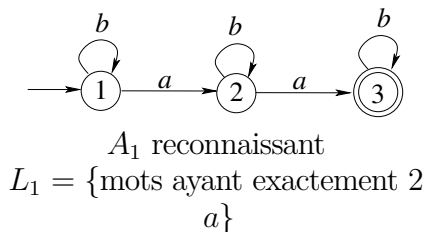
Méthode point par point

1. On pose les deux automates l'un à coté de l'autre (A_1 à gauche de A_2)
2. L'état initial i_1 de A_1 deviendra l'état initial du nouvel automate
3. L'état initial i_2 de A_2 ne sera plus initial
4. Les états acceptants t de A_1 ne seront plus acceptants (*dans un premier temps en tout cas*)
5. Les états acceptants de A_2 seront les états acceptants du nouvel automate
6. On ajoute les transitions nécessaires :
Si $i_2 \xrightarrow{a} d_2$ (avec $a \in \Sigma$) alors on ajoute une transition $t_1 \xrightarrow{a} d_2$ pour tout état acceptant t_1 de A_1
7. On ajoute les états acceptants nécessaires : *si i_2 est acceptant dans A_2 , alors t_1 sera acceptant dans A*

Remarque On peut *simplifier* l'automate obtenu en enlevant l'état 4 car aucune transition ne permet d'arriver dessus (état non accessible) ce qui nous donne finalement l'automate suivant :

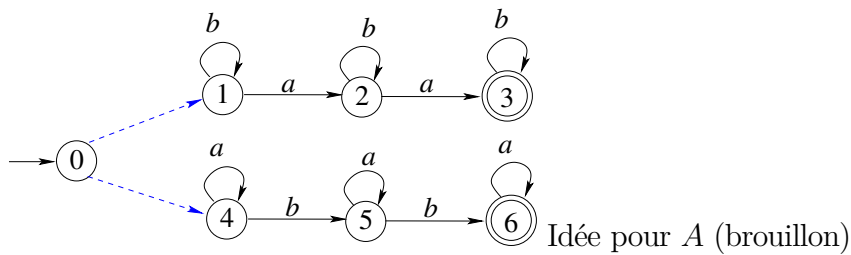


Somme d'automates Notons A_1 un automate permettant de reconnaître les mots de L_1 , et A_2 un automate permettant de reconnaître les mots de L_2 .



Pour reconnaître les mots de $L = L_1 + L_2$, l'automate A à construire doit alors être conçu comme donnant la possibilité de *parcourir soit A_1 soit A_2* .

On aimerait pouvoir réaliser ceci en posant l'un au-dessus de l'autre A_1 et A_2 , en ajoutant un nouvel état initial et en reliant celui-ci aux états initiaux de A_1 et A_2 par des transitions vides.

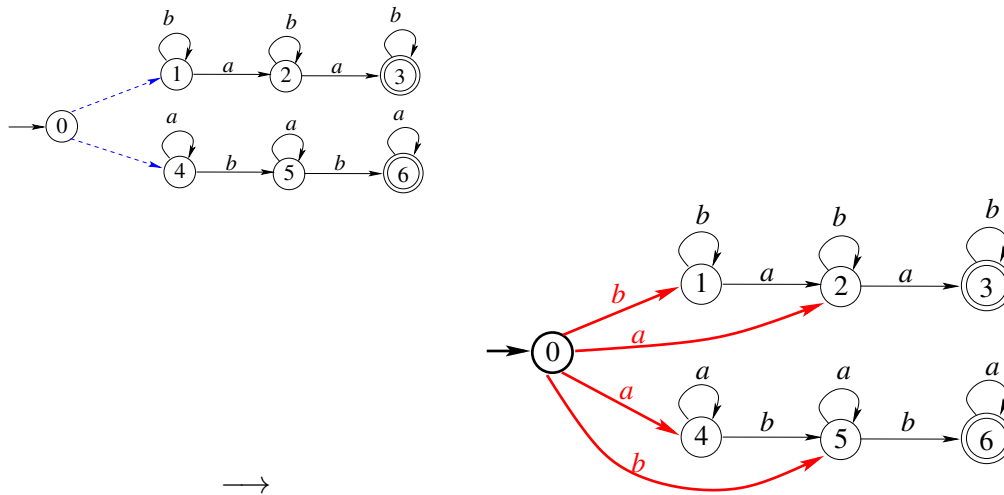


Mais un automate ne peut pas avoir des transitions vides, donc *il va falloir s'en débarrasser*.

Idée générale

- Appelons i , i_1 et i_2 les états initiaux de A , A_1 et A_2 .
- On remplace la transition vide de i vers i_1 par :
 - $i \xrightarrow{a} d$ s'il existe $i_1 \xrightarrow{a} d$ dans A_1 (car passer de i à i_1 sans rien faire puis de i_1 à d en faisant un a est équivalent à faire un a pour aller directement de i à d).
 - i sera terminal dans A si i_1 est terminal (car passer de i à i_1 sans rien faire, puis sortir directement de A_1 est équivalent à sortir directement de A par i).
- On fait pareil pour remplacer la transition vide de i vers i_2 .

Exemple

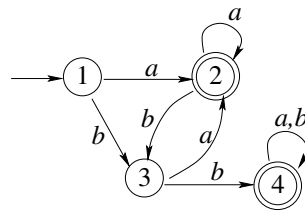


Méthode point par point

- On pose les deux automates l'un au dessus de l'autre
- Les états initiaux i_1 et i_2 des automates A_1 et A_2 ne sont plus initiaux pour le nouvel automate A

- On ajoute un état i qui sera l'état initial du nouvel automate
- On ajoute les transitions nécessaires :
(en notant d_1 un état de A_1 et d_2 un état de A_2)
 - si $i_1 \xrightarrow{a} d_1$ alors on ajoute une transition $i \xrightarrow{a} d_1$
 - si $i_2 \xrightarrow{a} d_2$ alors on ajoute une transition $i \xrightarrow{a} d_2$
 - si i_1 ou i_2 est également terminal alors i aussi

Étoile d'un automate Soit A_1 l'automate permettant de reconnaître les mots de L_1 .



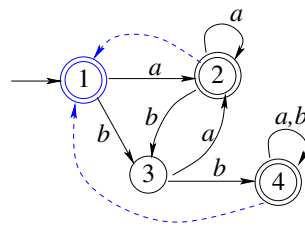
A_1 reconnaissant L_1

On a donc envie de pouvoir :

- *sortir tout de suite* et reconnaître ainsi le mot vide (ϵ),
- ou *parcourir l'automate autant de fois qu'on veut* pour concaténer plusieurs mots de A_1 et ainsi former notre mot.

Pour reconnaître les mots de L , l'automate A à construire doit alors être conçu comme « une boucle » de A_1 sur lui-même, permettant de sortir dès l'état initial.

On aimerait pouvoir réaliser cette boucle en reliant les états terminaux de A_1 à l'état initial de A_1 par des transitions vides (sans étiquette) et en faisant de l'état initial i_1 de A_1 un état également terminal.

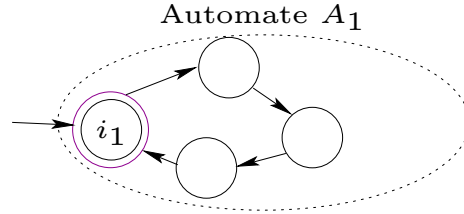


idée pour A

Ensuite, si la vie était simple, on n'aurait plus qu'à remplacer les transitions vides par des transitions étiquetées comme dans les constructions précédentes.

Mais en réalité c'est un peu plus compliqué : il peut être nécessaire de passer par une étape préalable de *standardisation*, sinon on risque de reconnaître des mots qui ne sont pas dans L_1 en rendant l'état initial acceptant pour reconnaître le mot vide.

Problème



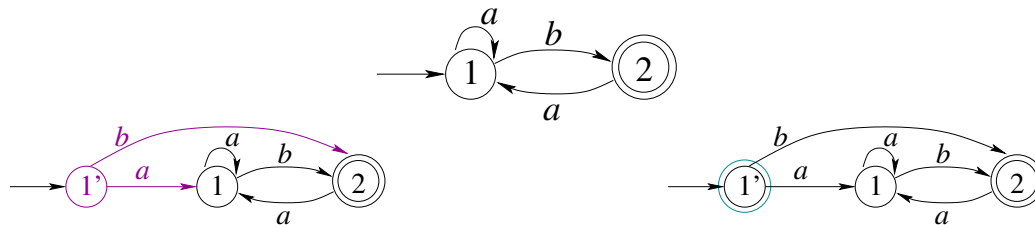
On transforme l'état initial i_1 en état acceptant pour pouvoir accepter le mot vide. Mais il ne faut pas que cette transformation permette de sortir de l'automate au milieu de la reconnaissance d'un mot de L_1 . Donc si i_1 est aussi *une étape intermédiaire* lors de la reconnaissance d'un mot de A_1 , il faut tout d'abord *transformer* A_1 pour différencier l'étape initiale des autres étapes se réalisant en cet état.

Standardisation



1 est un état initial par lequel on peut repasser lors du parcours de l'automate pour reconnaître un mot. *En faisant de 1 un état également terminal, on permettrait de sortir n'importe quand au milieu d'un tel parcours, ce qu'on ne veut pas.*

Pour que ce ne soit pas le cas on « éclate » l'état 1 en deux états : 1 et 1', et on relie 1' à tous les états auxquels 1 était lié (*recopiage des transitions partant de 1*).

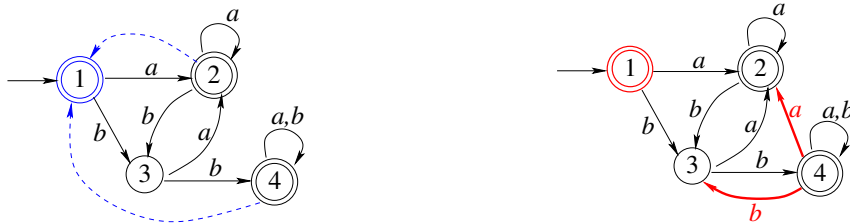


À partir de ce dernier automate, *si on transforme 1' en état également terminal*, on permet bien de reconnaître le mot vide ou un mot de L_1 et non pas autre chose.

Retour à la construction Un automate ne peut pas avoir des transitions vides, donc nous devons remplacer ces transitions vides par ce qui pourrait leur être équivalent

Sur un automate *standard*, on procède de la même manière que dans le cas de la concaténation ou de la somme : ne faire rien entre t_1 et i_1 suivi d'un a ($a \in \Sigma$) pour aller de i_1 vers un état d de A_1 est équivalent à aller directement de t_1 vers d par une transition étiquetée a .

Exemple



Méthode point par point

- Rendre A_1 **standard** en faisant en sorte qu'aucune transition ne revienne sur i_1 : si besoin dupliquer l'état initial ainsi que les transitions qui en partent.
- L'état initial i_1 de A_1 devient également un état terminal (car $\epsilon \in L_1^*$).
- Pour chaque lettre a et chaque état q
 - si $i \xrightarrow{a} q$ est une transition de A_1
 - alors on ajoute les transitions $t \xrightarrow{a} q$ pour chaque état terminal t .

2.5.2 Automate vers Expression régulière

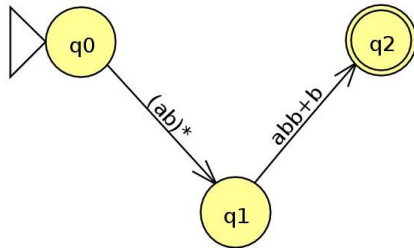
Soit A un automate qu'on veut convertir en expression régulière. On procède en 2 étapes.

1. On voit A comme un « super automate », dont les transitions sont des langages décrits par des expressions régulières. On transforme progressivement des super-automates jusqu'à obtention d'un super-automate avec 2 états, l'un initial et pas acceptant, l'autre acceptant (les expressions régulières deviennent de plus en plus compliquées mais le nombre d'état diminue).
2. On sait comment déduire l'expression régulière correspondant à un tel super-automate à 2 états.

Super-automate dont les transitions sont des langages La définition étend celle des automates. Au lieu que les transitions correspondent nécessairement à une lettre, elles correspondent maintenant à un langage qui est décrit par une expression régulière.

Lors d'une exécution dans ce super-automate, on passe d'un état à un autre en lisant non pas une seule lettre mais un mot appartenant au langage décrit par l'expression régulière.

Exemple. On considère le super-automate suivant.



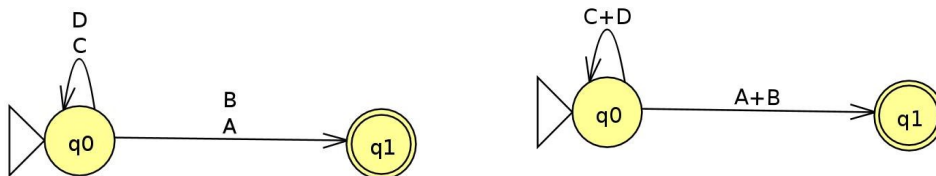
Si le mot en entrée est **abababb**, une exécution possible est de passer de l'état q_0 à l'état q_1 en lisant **ababab** puis de passer de l'état q_1 à l'état q_2 en lisant le reste du mot, c'est-à-dire la lettre **b** et finalement d'accepter.

Notez que ce modèle plus général est naturellement **non-déterministe** puisqu'on pourrait aussi faire : q_0 à q_1 en lisant **abab** puis q_1 à q_2 en lisant **abb** et d'accepter. On rappelle qu'un automate non déterministe accepte un mot si il existe une exécution qui accepte. En particulier, si d'autres exécutions rejettent le mot **abababb**, il n'en est pas moins accepté.

Notez aussi que l'on a maintenant le droit d'avoir des **transitions qui lisent le mot vide** qui est une expression régulière comme une autre. Le mot vide (qu'on note ϵ dans le cours et que JFLAP note λ) fait partie du langage $(ab)^*$. Une exécution possible est donc de passer de q_0 à q_1 en lisant ϵ puis de rejeter puisque il n'y a pas de transition possible depuis q_1 (aucun préfixe de **abababb** appartient au langage $abb+b$).

Une première simplification : au plus une transition entre 2 états. On peut toujours s'arranger pour qu'un super-automate ait au plus une transition entre deux états (qui peuvent être le même état dans le cas d'une boucle). Il suffit de faire la **somme des super-transitions**.

Exemple. Ci-dessous, A, B, C et D sont des expressions régulières décrivant des langages quelconques.



Langage vide et mot vide On autorisera les 2 comme expression régulière. Ils sont très différents.

- ϵ (ou λ en JFLAP) est le mot vide (1 mot accepté).

- \emptyset est le langage vide (aucun mot accepté).

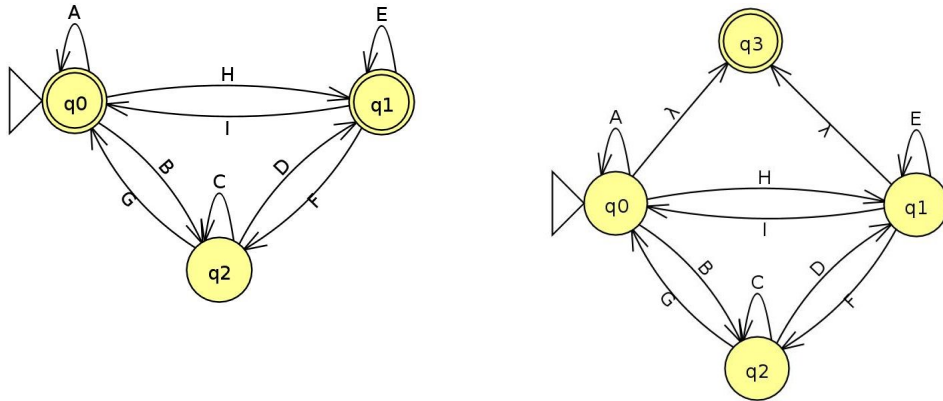
Les deux sont très pratiques pour simplifier le super-automate.

Notez d'abord que ϵ est l'élément neutre pour la concaténation, donc si dans $L_1.L_2.L_3.....L_n$ l'un des langage est ϵ alors on peut simplifier en l'enlevant.

Notez ensuite que si dans $L_1.L_2.L_3.....L_n$ l'un des langage est \emptyset alors le tout donne \emptyset .

Comme l'étoile d'un langage L est $\epsilon + L + L.L + \dots$, on obtient $(\emptyset)^* = (\epsilon)^* = \epsilon$.

Une seconde simplification : 1 état initial non acceptant et 1 seul état final distinct. Si on a plusieurs états acceptants, on les transforme en état non-acceptant, on ajoute un nouvel état acceptant et des transitions avec le **mot vide** depuis les anciens états acceptants vers ce nouvel état unique état acceptant.



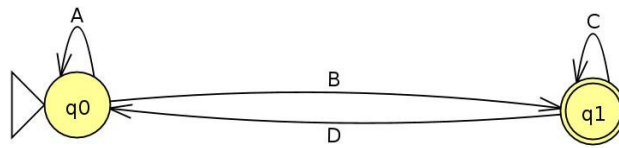
Une troisième simplification : exactement une transition entre 2 états.

On peut toujours s'arranger pour qu'un super-automate ait une et une seule transition entre deux états (qui peuvent être le même état dans le cas d'une boucle). On a vu précédemment comment utiliser la somme pour avoir au plus une transition. D'autre part, si il n'y a pas de transition, on ajoute une super-transition dont l'expression régulière est le **langage vide** \emptyset .



L'expression régulière d'un super-automate à 2 états Quand on a un super-automate à 2 états seulement, dont l'un est initial mais pas acceptant et l'autre est acceptant, on peut facilement déduire l'expression régulière.

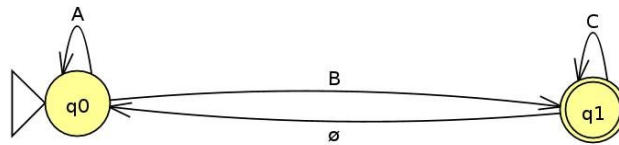
Le super-automate



a pour expression régulière

$$(A^*BC^*D)^*A^*BC^*$$

Lorsque certains langages sont \emptyset l'expression régulière est plus simple. Le super-automate



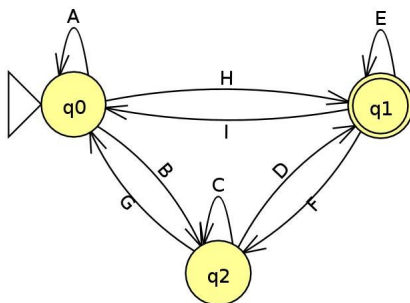
a pour expression régulière

$$A^*BC^*$$

Le cœur de la méthode : supprimer un état non initial non acceptant.

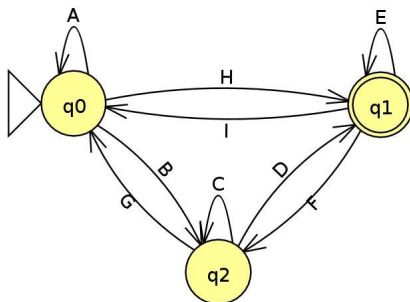
Avec les 3 premières méthodes, on a maintenant un automate qui a tous les arcs possibles, un état initial non acceptant et un unique état final. Si il n'y a pas d'autres états, on a terminé (voir ci-dessus). Sinon, on va supprimer un état en ajoutant dans le reste du super-automate des expressions régulières représentant les **détours** qu'on aurait pu faire en passant par cet état qu'on supprime.

Illustrons le principe sur un exemple : nous cherchons à éliminer l'état q_2 .



Pour aller de q_0 à q_0 :

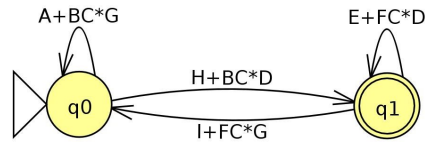
- (sans détour) on peut prendre la super-transition boucle dont l'expression régulière est A ; ou bien,
- (avec détour par q_2) on peut passer par l'état q_2 et revenir ce qui donne BC^*G .



Pour aller de q_1 à q_0 :

- (sans détour) on peut prendre la super-transition dont l'expression régulière est I ; ou bien,
- (avec détour par q_2) on peut passer par l'état q_2 avant d'aller à q_0 ce qui donne FC^*G .

On peut donc se passer de l'état q_2 si on ajoute pour chaque transition directe n'empruntant pas q_2 , l'expression régulière correspondant au détour par q_2 .



La méthode détaillée.

1. Première simplification : au plus une transition entre 2 états (avec +)
2. Seconde simplification : au plus un état acceptant distinct de l'état initial (avec ϵ)
3. Troisième simplification : exactement une transition entre 2 états (avec \emptyset)
4. Cœur de la méthode : on supprime l'un après l'autre tous les états sauf 2 en tenant compte de détours (ajout d'expressions régulières décrivant les détours).
5. Fin : on lit l'expression régulière sur le super-automate à 2 états.

2.5.3 Conclusion

On obtient donc le résultat suivant.

Théorème. *Un langage est reconnaissable par un automate si, et seulement si, il peut être décrit par une expression régulière.*

Dans la littérature, on parle des langages réguliers. C'est le premier niveau de hiérarchie de Chomsky en théorie des langages.

En informatique, c'est globalement ce qu'on manipule dans la partie *lexing* avant le *parsing* en compilation.

Vocabulaire

- ★ Concaténation d'automates
- ★ Somme d'automates
- ★ Étoile d'un automates
- ★ Standardisation d'un automate
- ★ super-automate (transitions sont des langages)
- ★ Langage réguliers

2.6 Lemme de la pompe

Nous avons vu que les automates et les expressions régulières sont en correspondance, puisqu'il existe des méthodes qui permettent de passer d'une expression régulière à un automate et inversement. La classe des langages concernés s'appelle en jargon les **langages réguliers**.

Nous allons voir une méthode qui permet dans certains cas de montrer qu'un langage **n'est pas régulier**. On se penche donc sur la question suivante.

Question. Étant donné un langage L , existe-t-il un automate fini qui reconnaît ce langage ?

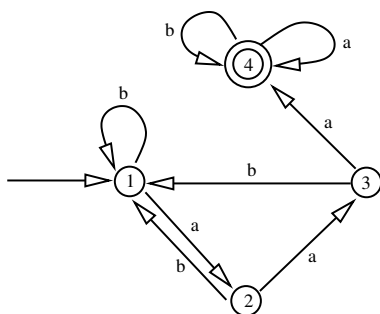
Si le langage L est donné par une **expression régulière**, on sait que la réponse est **Oui**, et on sait même **construire** un automate fini qui reconnaît ce langage. Mais si la réponse est **Non**, autrement dit s'il n'existe **aucun** automate fini qui reconnaît le langage L , comment le **prouver** ?

Outil pour le Non.

On va voir une propriété (appelée le **lemme de la pompe**) qui est vraie pour **tous** les langages réguliers.

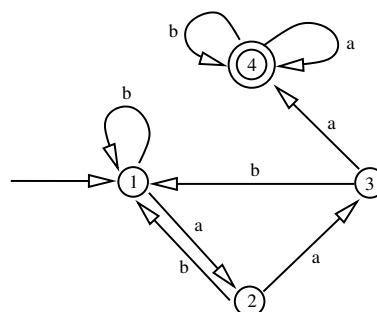
Par conséquent, si un langage **ne vérifie pas** cette propriété, il est **impossible** qu'il soit un langage régulier. Malheureusement, cet outil a ses limites puisqu'il existe des langages qui ne sont pas réguliers et pour lesquels la propriété reste vraie. Dans ce cas il existe une autre technique plus complexe qu'on ne verra pas ensemble mais pour lequel la caractérisation est nécessaire et suffisante (implication vs équivalence), il s'agit du théorème de Myhill–Nerode.

Ce que pomper veut dire.



- $aabbaaaba$ accepté
- $aabaabbaaaba$ accepté
- $aabaabaabaaaba$ accepté
- N'importe quel mot de la forme $(aab\dots aab)baaaba$ accepté
- Car il y a un cycle $1 \xrightarrow{a} 2 \xrightarrow{a} 3 \xrightarrow{b} 1$
- On dit qu'on peut **pomper** aab dans le mot $aabbaaaba$

Pomper ici ou pomper ailleurs.



2. $x \neq \varepsilon$
3. Tout mot de la forme $u(x...x)v$ appartient à L

Utilisation du lemme de la pompe.

On considère le langage suivant.

$$L = \{a^k b^k, k \geq 0\} = \{\varepsilon, ab, aabb, aaabbb, \dots\}$$

Remarque. L'expression ci-dessus *ressemble* à une expression régulière mais ce n'en est pas une. On a le droit d'écrire a^k pour k **fixé**, qui est une abréviation pour $aa\dots a$ k fois, par exemple pour $k=5$ ça veut dire $aaaaa$. Ci-dessus, le k est arbitrairement grand.

On peut se dire qu'on peut le faire avec l'étoile, c'est vrai pour une seule lettre, ou un seul mot, comme dans $(ab)^* = \{(ab)^k, k \geq 0\}$. Si on utilise plusieurs étoiles comme dans a^*b^* cela correspond à $\{a^k b^{k'}, k, k' \geq 0\}$.

On va montrer que L n'est pas régulier par un **raisonnement par l'absurde**.

Démonstration.

- **Supposons** que L soit reconnu par un automate \mathcal{A} et appelons n le nombre d'états de \mathcal{A} .
- Le mot $w = a^n b^n \in L$ avec $|w| = 2n \geq n \implies$ **pomper**
- C.-à-d. $w = uxv$ avec $|ux| \leq n$, $x \neq \varepsilon$ et $u(x...x)v \in L$
- Donc $x = a^d$ avec $d > 0$
- Par exemple, $w' = uxv$ **appartient à L**
- Mais $w' = a^{n+d} b^n$, donc w' **n'appartient pas à L**
- **Contradiction**
- Donc L n'est pas reconnu par \mathcal{A}

□

Conclusion.

Il n'existe aucun automate fini qui reconnaisse le langage L , c.-à-d. **L n'est pas un langage régulier**

D'autres langages non réguliers.

- Le langage des palindromes
 - Mot qui se lit pareil de droite à gauche et de gauche à droite
 - En français : RADAR, KAYAK, RESSASSER, ...
 - On note u^R le mot u renversé : bba devient abb

2. En jargon de mathématicien, un lemme est un petit théorème qui sert à montrer un gros théorème. C'est souvent un outil qui n'est pas intéressant seul.

- Tout mot w de la forme uu^R est un palindrome
- Le langage des parenthèses bien formées
 - $()()()$ ou $((()))$
 - mais pas $()()$ ni $((()$
- Le langage composé des mots qui contiennent le même nombre de a que de b
- etc.

Énoncé, quantificateurs et jeu

On peut voir la preuve d'un énoncé comme un jeu³ entre 2 joueurs, qui pour la logique des prédicats correspondent aux deux quantificateur universel (\forall) et existentiel (\exists). Le premier Abélard (pour **forAll**, en notant que typographiquement le \forall est un A sur sa tête) essaye de montrer que l'énoncé est faux alors que la seconde joueuse Éloïse (pour **Exists**, en notant que typographiquement le \exists est un E retourné) essaye au contraire de montrer que l'énoncé est vrai.

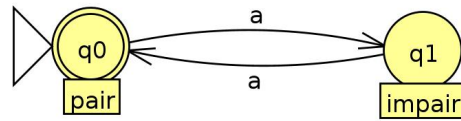
En particulier, quand on applique le lemme de la pompe pour un langage donné, on peut voir la preuve comme un tel jeu. C'est ce qui se passe dans l'activité lié au lemme de la pompe dans le logiciel JFLAP.

Pour vous aider à bien comprendre les étapes du jeu, nous allons retranscrire le lemme de la pompe en explicitant les quantificateurs.

Joueur	action
Éloïse	Il existe un automate à m -états
Abélard	Pour tout mot accepté w de m lettres ou plus
Éloïse	Il existe un cycle y tel que $w = xyz$ avec xy ayant au plus m lettres
Abélard	Pour toute répétition i de ce cycle (pouvant être 0)
Victoire Éloïse : Le mot $w' = xy^iz$, est accepté.	

Si le langage est bien régulier, l'automate existe et permet à Éloïse d'avoir une stratégie gagnante. c'est le cas pour le langage contenant un nombre pair de a reconnu par l'automate suivant.

3. https://en.wikipedia.org/wiki/Game_semantics



Si le langage n'est pas régulier, on souhaite montrer que l'automate n'existe pas. Autrement dit on essaye de montrer la négation de l'énoncé précédent. Le calcul est assez mécanique, les quantificateurs s'inversent et on prend **la négation de la condition de victoire** (voir la loi de De Morgan en logique et la notion de dualité pour plus de détails).

Joueur	action
Abélard	Pour tout automate à m -états
Éloïse	Il existe un mot accepté w de m lettres ou plus
Abélard	Pour tout cycle y tel que $w = xyz$ avec xy ayant au plus m lettres
Éloïse	Il existe une répétition i de ce cycle (pouvant être 0)
Victoire Éloïse : Le mot $w' = xy^iz$, n'est pas accepté.	

C'est bien cette variante du jeu pour lequel on a une stratégie gagnante quand on montre que $\{a^n b^n \mid n > 0\}$ n'est pas régulier.

Vocabulaire

- ★ Langage régulier ou non
- ★ Lemme de la pompe
- ★ Sémantique par un jeu pour une preuve

Chapitre 3

Les machines de Turing

Nous allons maintenant nous pencher sur un modèle de machine qui semble au premier abord n'être que légèrement différent de celui des automates. D'une part, on va pouvoir revenir en arrière sur le mot, et d'autre part on va pouvoir écrire ce qui permet d'avoir une mémoire arbitrairement grande. On va voir que ces deux changements permettent d'obtenir *in fine* un modèle très riche puisque *n'importe quelle procédure de calcul connue à l'heure actuelle* peut être traduite théoriquement par une machine de Turing.

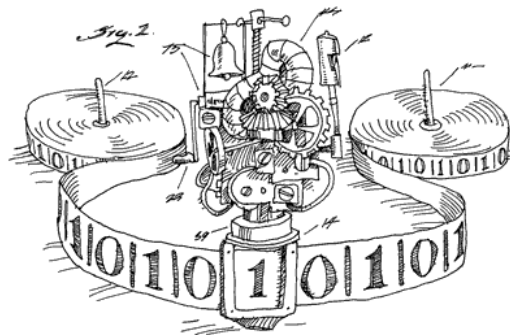


FIGURE 3.1 – La machine de Turing et son ruban arbitrairement grand sur lequel on peut écrire.

Il y a beaucoup de ressources sur Alan Turing disponibles en ligne, en particulier depuis son centenaire en 2012. Vous pouvez trouver ses articles originaux et ses brouillons¹. C'est un mathématicien qui a eu une influence forte en informatique, malgré une vie trop courte. En droit britannique, une loi porte son nom².

1. Sa page wikipedia comporte de nombreux liens, dont cette url <http://www.turingarchive.org/>.

2. https://en.wikipedia.org/wiki/Alan_Turing_law.

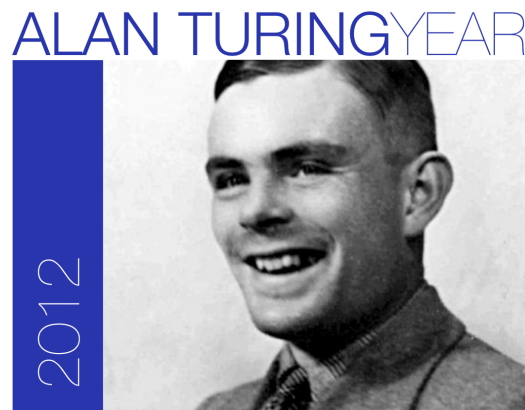


FIGURE 3.2 – Alan Turing (1912-1954).

Nous allons voir les machines de Turing de manière très informelle en les manipulant avec JFLAP avant de donner une définition formelle.

3.1 Turing avec JFLAP

Dans le menu choisissez *Machine de Turing*. L'éditeur graphique vous permet de créer rapidement le programme d'une machine de Turing. Les cercles sont les états de la machine (comme pour les automates). Un clic droit permet de rendre initial un état, ou final des états. Un état final correspond ici en fait à un état acceptant³.

La machine s'arrête si il n'y a pas de transition appropriée (en rejetant si l'état dans laquelle elle se trouve n'est pas final).

Dans la suite on utilise l'alphabet $\{0,1\}$ en plus du blanc (ressemblant à \square pour JFLAP). Notez que le ruban est infini à gauche et à droite pour JFLAP. Il y a de nombreuses variations possibles pour la machine de Turing qui ne font pas de différence.

Un point technique avec JFLAP. Si vous voulez fabriquer une transition qui écrit un \square pour une transition, il suffit de ne rien saisir dans cette cellule pour la transition et d'appuyer sur entrée.

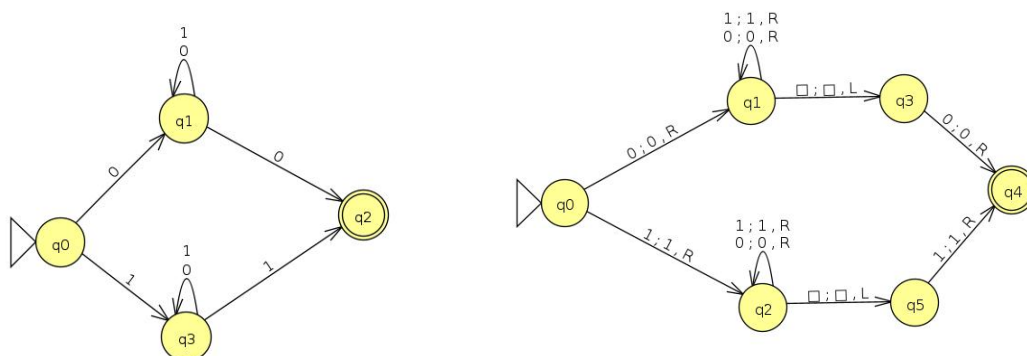
- Exercice 6.**
1. Écrire un programme qui lit les caractères du mot d'entrée de la gauche vers la droite et s'arrête en acceptant.
 2. Même question pour un programme qui accepte si et seulement si la première et la dernière lettre sont les mêmes.
 3. Même question pour les mots qui sont des *palindromes* (exemple de palindrome en français KAYAK). On peut dans un premier temps ne gérer correctement que les mots qui sont de longueur paire.

3. La nomenclature pas tout à fait adaptée du logiciel pour les machines de Turing vient du fait que ce dernier permet de simuler toutes sortes de machines dont des automates finis.

Notez qu'on peut toujours transformer un automate en une machine de Turing. LA MdT fonctionne comme l'automate au sens où les transitions sont les mêmes sauf qu'en version Turing il faut écrire explicitement qu'on va vers la droite et qu'on recopie le symbole lu.

La seule petite différence potentielle concerne l'arrêt. Pour repérer la dernière lettre, vous pouvez revenir à gauche lorsque vous lisez un blanc comme dans l'exemple ci-dessous puis traiter la dernière lettre comme dans l'automate. Alternativement, vous pouvez juste vous souvenir dans l'état de la dernière lettre lue, ce qui évite d'aller vers la gauche.

L'automate ci-dessous à gauche devient la machine de Turing ci-dessous à droite.



Exercice 7. Reprenez l'exemple ci-dessus pour que la machine de Turing n'aille jamais à gauche. Indications : il faut remplacer $q1$ par deux états, l'un qui sait que la dernière lettre lue est un 0, l'autre que la dernière lettre lue est un 1 ; et adapter le traitement lorsqu'on lit un blanc selon la dernière lettre lue. Il faut procéder de même pour la branche du bas).

Exercice 8. Argumentez qu'une machine de Turing qui ne peut ni écrire ni déplacer sa tête vers la gauche est un automate.

Pour les palindromes pairs, de la forme $w.w^R$ (où w^R correspond au renversé de w) on ne peut pas se souvenir dans nos états du début du mot w qui est arbitrairement grand et donc pour un mot assez grand forcément plus grand que le nombre (fini) d'états de notre machine.

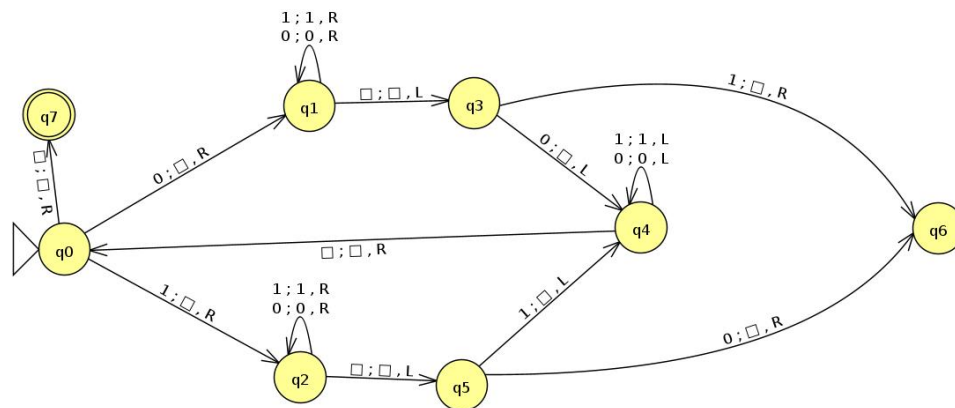
Si on dispose d'une machine de Turing à **2 rubans**, on pourrait : recopier le premier ruban sur le second ; ramener la tête de lecture du premier ruban tout à gauche ; puis décaler la tête du premier ruban vers la droite et celle du second ruban vers la gauche en vérifiant une par une que les lettres sont identiques.

Comme nous ne disposons que d'un ruban, la seule mémoire non bornée à notre disposition est l'unique ruban. L'idée consiste donc à vérifier l'un après l'autre les

couples de lettres suivantes : (première lettre, dernière lettre) puis (seconde lettre, avant-dernière lettre) etc.

Si nous avons **deux têtes** il suffirait de placer notre seconde tête à la fin du mot et de procéder comme dans la version à 2 rubans.

Puisqu'on a une seule tête, cette dernière va devoir faire des aller-retours. Il va falloir marquer d'une manière ou d'une autre les lettres qu'on a vu pour éviter de repasser au même endroit. L'exemple ci-dessous enlève tout simplement les lettres en cours de vérification.



Exercice 9. La machine de Turing autorise un alphabet de travail plus grand que l'alphabet d'entrée (en particulier pour marquer des symboles). Introduisez les symboles 2 et 3 « de travail » comme copie marquées de 0 et 1 et adaptez la réponse ci-dessus pour que le calcul laisse le ruban inchangé en cas de palindrome.

3.2 Définition

Nous suivent le modèle de JFLAP. Il y a une certaine liberté sur la définition.

Une machine de Turing (avec un ruban infini à gauche et à droite) est donnée par : un alphabet d'entrée Σ non vide, contenu dans un alphabet de travail Γ qui contient un symbole spécial \square (le blanc), un ensemble fini d'états Q contenant un état initial q_0 et un sous-ensemble F de Q formant les états finaux.

Le ruban est infini mais le seul symbole qui peut apparaître infiniment souvent est le blanc \square .

On dispose en plus d'une **fonction de transition** *delta* qui fait correspondre à certaines paires d'un état non final de $Q \setminus F$ et d'une lettre de l'alphabet de travail Γ un triplet d'un élément de Q , une lettre de Γ et un **mouvement** (L,R ou S pour left, right ou stay).

Cette fonction partielle est le **programme** de la machine dans le même esprit que celui d'un automate, sauf que maintenant nous avons besoin de deux informations en plus du prochain état, à savoir la lettre qu'on va écrire sur le ruban (on peut écrire la même qu'en entrée si on ne veut pas vraiment écrire) et le déplacement de la tête de calcul (pas forcément à droite comme pour les automates).

Au début du calcul la machine est dans l'état initial q_0 , la tête de lecture est positionnée sur le premier symbole qui n'est pas un \square puis successivement vers la droite les autres lettres de l'entrée. Le ruban contient donc initialement uniquement un nombre fini de lettres de Σ (le mot d'entrée) entourées d'une infinité de blancs.

Le calcul procède selon la fonction de transition δ . Comme pour les automates, si dans une situation δ étant une fonction partielle, le calcul est bloqué, on arrête en rejetant. Si le calcul se termine, le résultat du calcul est sur le ruban.

3.3 Robustesse de notre modèle

On considère 2 variantes du modèle de la Machine de Turing. Notre alphabet d'entrée est $\{0,1\}$ (celui sur lesquels les mots des langages associés à nos machines sont définis). Les machines peuvent avoir d'autres symboles propres, formant un *alphabet de travail*.

Le premier modèle est celui de JFLAP à un ruban, c'est-à-dire le modèle avec un ruban infini et rempli de blancs à gauche et à droite du mot d'entrée.

Le second modèle est infini seulement à droite : il y a un caractère spécial \triangleright à gauche du mot d'entrée ; le mot d'entrée puis une infinité de blancs à droite de ce mot.

On peut montrer que l'ensemble des langages calculables est identique pour ces deux modèles.

- Exercice 10.**
1. Montrez comment transformer le programme d'une MdT avec un ruban infini à droite en celui d'une MdT infinie à gauche et à droite. Indication : il faut que le ruban de la seconde machine ressemble à celui de la première.
 2. Montrez comment transformer le programme d'une MdT avec un ruban infini à gauche et à droite en celui d'une MdT infinie à droite. Indication : quand on plie en 2 un ruban infini à droite et à gauche, on obtient un ruban infini à droite dont les cellules sont doubles.

On a évoqué tout à l'heure la possibilité d'avoir deux têtes de calcul. On peut à nouveau montrer que ceci n'augmente pas l'expressivité du modèle.

Il existe de nombreuses autres variantes : par exemple des machines qui calculent sur un espace à 2 dimensions au lieu d'un ruban, ou encore des modèles plus proche de l'architecture des machines actuelles (le modèle RAM) qui peuvent stocker des adresses de la mémoire (du ruban) dans des **registres** et lire/écrire dans les registres les valeurs stockées à une adresse mémoire sans avoir à déplacer péniblement la tête le long du ruban. En terme de ce qu'on peut décider / calculer, il n'y a pas de différence.

Il y a des formalismes historiques différents, plus éloignés du modèle de Turing comme la théorie générale des fonctions récursives ou encore celui du λ -calcul. Tous sont équivalents au formalisme de Turing.

La thèse de Church Turing. Pour tout modèle raisonnable de calcul, on obtient la même notion de ce qui est calculable.

Arguments en faveur de cette thèse

- Théorie générale des fonctions récursives (Gödel, Herbrand 1933).
- λ -calcul (Church, 1936).
- Machines de Turing (Turing, 1936).
- Trois modèles équivalents (Church 1936, Turing 1937).
- Machines de Post (1936).
- Machines de Turing avec plusieurs rubans (Minsky).
- Machines à compteurs.
- Machines à registre.
- *etc.*

Deuxième partie

Comparer

Chapitre 4

Limites de ce qu'on peut calculer

Au vu de la thèse de Church-Turing, on peut choisir n'importe quel formalisme connu permettant de définir une notion de calcul. En particulier on va choisir ici le modèle de la machine de Turing.

4.1 Définitions.

Définition. Un langage L est *décidable* si il existe une machine de Turing M qui accepte exactement les mots de ce langage, c'est-à-dire que pour tout mot x sur son ruban *la machine M arrête son calcul dans un état acceptant ssi x est dans L .*

Définition. Une fonction f est *calculable* si il existe une machine de Turing M qui étant donné x sur son ruban s'arrête avec comme contenu de ruban $f(x)$.

Un peu de dénombrement.

L'ensemble des programmes est dénombrable mais l'ensemble des langages n'est pas dénombrable.

Limite forte de ce qu'on peut calculer. Il existe « beaucoup » de langages pour lesquels nous n'avons pas de programme.

Lemme. *L'ensemble des programmes est dénombrable*

On peut rester informel comme ci-dessus et parler de langage de programmation quelconque (du C par exemple) et via le codage ascii faire correspondre au texte du programme un nombre en binaire. La croyance en la thèse de Church-Turing faisant le reste.

Alternativement si on souhaite être plus précis on peut montrer ceci.

Lemme. *L'ensemble des machines de Turing est dénombrable.*

Démonstration. On peut coder en binaire les états, les symboles de l'alphabet et les actions de déplacement de la tête de lecture.

À l'aide de symboles séparateurs adaptés (parenthèses, virgules), on peut lister les transitions. Par exemple une liste de mots de la forme (état avant, symbole lu, état après, symbole écrit, déplacement).

On peut ensuite reprendre l'argument précédent avec le code ascii. \square

Digression : code d'une machine de Turing. On va appeler *code d'une machine de Turing* le mot suivant :

nombre d'états en binaire , nombre de symboles en binaire , liste des transitions dans la preuve précédente séparées par des virgules.

Quitte à ajouter des 0 à gauche, on fait en sorte que tous les mots binaires codant des états, des symboles ou des actions ont la même longueur. Notez que ce mot est sur l'alphabet comportant les symboles 0 1 , ()

Quand on travaille sur les MdT il est pratique de pouvoir décrire « la photo » de la machine à un instant du calcul. On parle de *configuration* : il s'agit du contenu du ruban, de la position de la tête et de l'état.

Dans l'esprit de notre code, nous pouvons donner un *code pour une configuration*.

codes des lettres à gauche de la tête , code de l'état , codes des lettres à droite de la tête .

Notez que ce mot est sur l'alphabet comportant les symboles 0 1 ,

4.2 Trop de langages.

Le principe de la preuve est intéressant car c'est un argument de diagonalisation, argument qui est central en calculabilité et qu'on retrouve souvent.

Lemme. *L'ensemble des langages sur l'alphabet $\{0,1\}$ n'est pas dénombrable.*

Démonstration. On énumère les mots binaires dans l'ordre lexicographique $s_0 = 0, s_1 = 1, s_2 = 01, s_3 = 10, s_4 = 11, \dots$

On suppose par l'absurde que l'ensemble de tous les langages sur $\{0,1\}$ est dénombrable et donc qu'on peut énumérer sous forme de liste L_0, L_1, L_2, \dots (l'index est donné par la bijection de \mathbb{N} dans l'ensemble de tous les langages).

Soit L le langage diagonal : $\{s_i \text{ tel que } s_i \notin L_i\}$.

L apparaît dans la liste, donc il existe un index j pour lequel $L = L_j$. Ceci est absurde puisque $s_j \in L \iff s_j \notin L_j$. \square

On a donc démontré ce qu'on souhaitait.

Théorème. *Le nombre de machines de Turing est dénombrable. Par contre l'ensemble des langages sur l'alphabet $\{0,1\}$ ne l'est pas. En conséquence, il existe des langages sur l'alphabet $\{0,1\}$ qui ne sont pas décidables par une machine de Turing.*

Nous allons voir dans la suite qu'on peut exhiber un exemple concret : le *problème de l'arrêt* d'un programme.

4.3 Machine de Turing Universelle.

On a vu qu'on peut coder (le programme) d'une machine de Turing M comme un mot sur l'alphabet comportant les symboles $0, 1, (,)$

On peut coder l'entrée x d'une telle machine par un mot dans le même esprit.

La *machine de Turing universelle* U prend sur son ruban le code d'une machine M , suivi d'un $;$ suivi du code d'une entrée x . Cette machine universelle travaille donc sur l'alphabet comportant les symboles $0, 1, (,) ;$

Notation simplifiée pour l'entrée : on écrira juste $M;x$

Programme de la machine de Turing Universelle.

En gros la machine va travailler sur x en utilisant le programme M . Le principe est similaire à la manière dont un programme assembleur est traité par un processeur.

Pour décrire proprement le fonctionnement de cette machine il est plus facile de considérer une machine de Turing avec plusieurs rubans (on pourra toujours la simuler par une machine à un seul ruban dans un second temps).

Le premier ruban reste initialement inchangé. Le second ruban contient le code de la configuration de la machine M . La machine U simule le calcul de M pas à pas : scan de l'état actuel de M sur le second ruban, recherche d'une règle adaptée sur le premier ruban, changement adapté de la configuration sur le second ruban.

Si l'entrée de U est incohérente et ne correspond pas au code d'une machine de Turing, la machine U déplace sa tête indéfiniment vers la droite.

4.4 Un exemple concret de problème indécidable.

Définition (problème de l'arrêt).

$$H = \{M;x \text{ tel que } M \text{ s'arrête sur l'entrée } x\}$$

($M;x$ est codé comme expliqué précédemment)

Théorème. *Le problème de l'arrêt est indécidable.*

La preuve se fait par l'absurde avec un argument de diagonalisation.

Esquisse de preuve. Par l'absurde. Soit M_H une machine qui décide H .

Soit D une machine qui prend en entrée le code d'une machine de Turing M . D accepte M ssi M_H ne s'arrête pas sur l'entrée $M;M$.

La contradiction apparaît lorsqu'on se penche sur le calcul de la machine D sur son propre code en entrée. \square

4.5 Réduction de Turing et indécidabilité d'autres problèmes.

On peut utiliser le concept de *réduction* pour montrer que d'autres problèmes sont indécidables.

Exemple.

$$S := \{M \text{ tel que } M \text{ s'arrête sur toute entrée}\}$$

Si S était décidable, on pourrait y réduire le problème H (détails ci-dessous).

- Entrée de H : $M;x$.
- Construction de la machine M' qui va prendre en entrée y et s'arrêter si y est différent de x , sinon en cas d'égalité la machine M' répond comme M sur l'entrée x .
- M' appartient à S ssi $M;x$ appartient à H .

Le **théorème de Rice** indique que toute propriété sur les machines sont indécidables à condition d'être non triviale. On peut en déduire par exemple qu'il est impossible de savoir si une MdT est équivalente à un automate.

Il est possible de dépasser l'indécidable en acceptant les **machines à Oracle**. On obtient alors une hiérarchie stricte, le problème d'arrêt d'un niveau est indécidable à ce niveau. Cette hiérarchie s'appelle la hiérarchie arithmétique (voir *théorème de Post* pour plus de détails).

Chapitre 5

Existence d'algorithmes efficaces

5.1 Ressources, dont le temps.

Quand on exécute un algorithme, deux ressources ont un sens particulièrement important : le *temps* d'exécution (souvent évalué approximativement à partir de primitives utilisées comme accéder à une valeur dans un tableau) et l'*espace* utilisé (stockage de calculs intermédiaires).

Sur notre modèle de Machine de Turing à un ruban, on peut étudier la notion de temps facilement : c'est simplement *le nombre d'étapes de calcul*.

Pour l'espace, on utilise le plus souvent au moins trois rubans avec un modèle plus contraint : le ruban d'entrée (lecture uniquement), le ruban de sortie (écriture seule), et le ruban de travail. L'espace utilisé par la machine est alors vu comme le nombre de cellules du *ruban de travail* qui sont utilisées au cours de calcul.

On se concentre sur le temps. On va regarder le *pire des cas*¹ On dit que la machine de Turing calcule en temps $f(n)$ (pour une fonction $f: \mathbb{N} \rightarrow \mathbb{N}$) si pour toutes les entrées de taille n , le calcul s'arrête en au plus $f(n)$ étapes.

5.2 Mesurer le temps

En pratique on choisira pour f une fonction « raisonnable » comme n^2 ou 2^n . On travaille aussi le plus souvent sans tenir compte des constantes multiplicatives et on utilise la notation « grand O ».

Le reste de cette section est un peu technique. Vous pouvez ignorer les esquisses de preuve et essayer de lire et digérer les intitulés des résultats. Ces résultats formalisent ce qui est expliqué au paragraphe précédent.

1. La complexité en moyenne existe aussi, mais ça demande de connaître une distribution raisonnable des entrées. C'est aussi un cadre de travail plus difficile pour démontrer des choses. En pratique, l'expérimentation sur des instances bien choisies permet de comparer divers algorithmes.

Notation Bachmann–Landau.

- $f(n) = O(g(n))$ dénote qu'il existe des entiers positifs c et N tels que pour tout n plus grand que N , $f(n) \leq c \cdot g(n)$.
- $g(n) = \Omega(f(n))$ pour $f(n) = O(g(n))$
- $g(n) = \Theta(f(n))$ pour $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$

Vous allez retrouver cette notation dans le cours d'algorithmique.

Phénomène d'accélération linéaire.

Théorème. *Pour toute machine de Turing M , calculant en temps $f(n)$ et toute constante $0 < c < 1$, il existe une machine de Turing M' équivalente en temps $2+n+c \cdot f(n)$.*

Idée de la preuve. La machine M' a des « grosses » cellules (un alphabet augmenté de symboles qui correspondent à la concaténation de quelque chose comme $\lceil \frac{6}{c} \rceil$ lettres de l'alphabet original).

Initialement la machine va zipper l'entrée (un bloc de cellules deviennent un seul symbole). Ceci a un coût linéaire puisqu'il faut au moins lire l'entrée. Ensuite la machine M' simule c étapes de M en un nombre borné d'étapes (6 étapes), d'où le gain linéaire. \square

Définition. On note $\text{TIME}(f(n))$ la classe des problèmes décidables par une machine de Turing déterministe en temps $f(n)$.

Bien choisir une fonction pour mesurer. Pour mesurer le temps, on ne peut pas prendre n'importe quelle fonction.

Théorème (Borodin 1969 - Trakhtenbrot 1967). *Il existe une fonction calculable f telle que $\text{TIME}(f(n)) = \text{TIME}(2^{f(n)})$*

La fonction du théorème ci-dessus est particulièrement non standard. En pratique, les fonctions discrètes usuelles analogues de celles des cours de math de lycée ne posent pas de problèmes. *Grosso Modo* Ce sont celles pour lesquelles on peut écrire une machine de Turing qui sert d'*horloge* pour compter le bon nombre d'étapes.

5.3 Quelle machine ?

Il existe une dépendance polynomiale entre divers modèles de calcul

Théorème. *Pour toute machine à k rubans en temps $f(n)$ il existe une machine de Turing équivalente à 1 ruban et en temps de l'ordre de $k^2 f(n) \times f(n)$.*

Pour le modèle de machine à registre RAM (modèle théorique le plus proche d'un ordinateur), il existe un résultat similaire avec à l'exposant 7 plutôt que 2.

Temps non déterministe.

On retrouve la notion de non déterminisme qu'on avait vu pour les automates.

Définition. On note $\text{NTIME}(f(n))$ la classe des problèmes décidables par une machine de Turing *non-déterministe* en temps $f(n)$.
(Attention : absence de symétrie mais dualité entre acceptation \exists et rejet \forall)

Deux classes de complexité robustes.

$$\text{PTime} = \bigcup_{c \geq 1} \text{Time}(n^c)$$

(la machine de Turing est déterministe)

Calcul en temps Polynomial \approx calcul efficace.

$$\text{NP} = \bigcup_{c \geq 1} \text{NTIME}(n^c)$$

(la machine de Turing peut être non-déterministe)

Vérification en temps Polynomial \approx calcul efficace. Ces deux classes sont robustes au sens où elles ne dépendent pas vraiment du modèle de machine utilisé. Ce n'est pas le cas par exemple du temps linéaire déterministe qui donne des classes différentes selon qu'on travaille avec des machines de Turing ou des machines RAM.

Exemples.

Calcul en temps polynomial	Vérification en temps polynomial
Graphe Eulérien ?	graphe Hamiltonien ?
Graphe 2-colorable ?	Graphe 3-colorable ?
Circuit	SAT

Un petit soucis. Si je trouve un algorithme polynomial, super je suis content. Que faire si je réfléchis très fort et que je ne trouve pas d'algorithme polynomial pour mon problème favori ? En particulier quand ce problème est dans NP ?

Exemple. Je voudrais savoir si on peut tester si un graphe est 3 colorable en temps polynomial. Après tout j'ai trouvé un truc pour les graphes 2-colorables.

Problème SAT.

- Entrée : formule propositionnelle (CNF)
- Question : satisfaisable ?

Exemple. Une entrée ressemble à ceci : $(x \vee y \vee z) \wedge (x \vee \neg t \vee \neg u \vee v) \dots$

On doit trouver des valeurs vrai ou faux pour chaque variable de sorte que chaque *clause* soit vraie (au moins un *littéral* par clause doit être vrai).

Notation. \wedge et, \vee ou, \neg négation.

5.4 Vers la notion de réduction.

Solveur SAT : logiciel permettant de résoudre le problème SAT. Typiquement capable de résoudre des instances à plusieurs millions de variables.

Modéliser avec SAT. On prend l'entrée d'un problème et on fabrique une formule propositionnelle. La réponse du solveur SAT permet de résoudre le problème initial.

Exemple. Essayons avec le problème de 3 colorabilité.

Modéliser = Réduire.

Définition. Soit deux problèmes de décisions Ω_1 et Ω_2 . Une *réduction* de Ω_1 à Ω_2 est une fonction calculable en temps polynomial r qui transforme une entrée x_1 du problème Ω_1 en une entrée $x_2 = r(x_1)$ du problème Ω_2 telle que $x_1 \in \Omega_1$ ssi $x_2 \in \Omega_2$.

- Remarque.*
- On peut composer les réductions (transitivité de l'existence d'une réduction).
 - Si le problème cible Ω_2 est polynomial alors le problème de départ Ω_1 est lui aussi polynomial.

5.5 SAT comme étalon pour NP.

Théorème (Cook 1971, Levin 1973). *Tout problème Ω de NP se réduit à SAT.*

Démonstration. Idée de la preuve Ω étant dans NP, il existe une machine non-déterministe T en temps polynomial $p(n)$ (p est un polynôme).

Pour une entrée x de Ω , le calcul sur la machine T occupe à tout instant au plus $p(t)$ cellules et la machine passe par au plus $p(t)$ étapes de calcul. On peut donc décrire par un nombre polynomial de variables booléennes les $p(t)$ configurations successives de la machine.

Par exemple, on a des variables $T_{p,t,s}$ avec $1 \leq p, t \leq p(n)$ qui seront vrai ssi la cellule à position p au temps t contient le symbole s de l'alphabet.

Ensuite on peut écrire des clauses pour contraindre ces variables à bien modéliser le calcul.

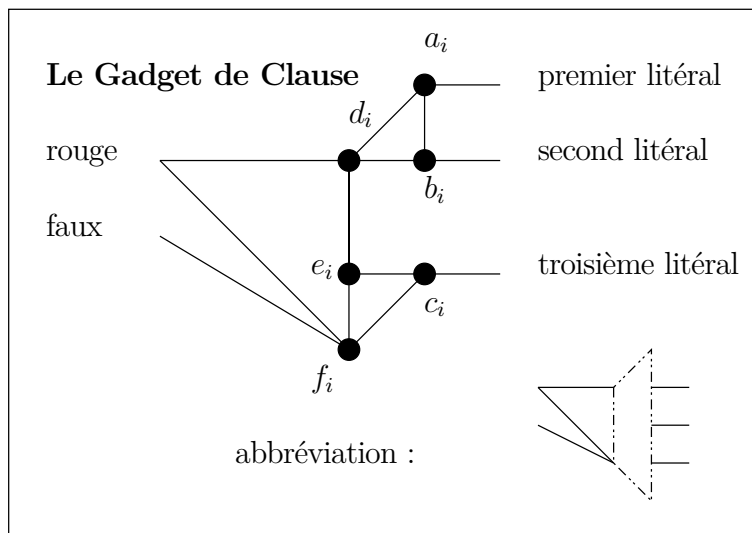
La page wikipedia propose une preuve complète : https://en.wikipedia.org/wiki/Cook-Levin_theorem \square

D'autres problèmes comme SAT ?

On peut se demander si à l'instar de SAT, il y a d'autres problèmes de NP auxquels tout problème de NP se réduit (en jargon on dira *NP-complet*).

On pourrait employer la même stratégie que Cook et coder un calcul dans le cadre du problème étudié. En pratique, on va plutôt réduire un problème NP-complet connu (SAT par exemple) au problème étudié. La transitivité des réductions permettant de conclure à la NP-complétude du problème étudié.

Exercice 11 (Réduire SAT à 3-Col). Il s'agit de réduire le problème SAT au problème 3-Col. Indication. Il faut utiliser le gadget ci-dessous.

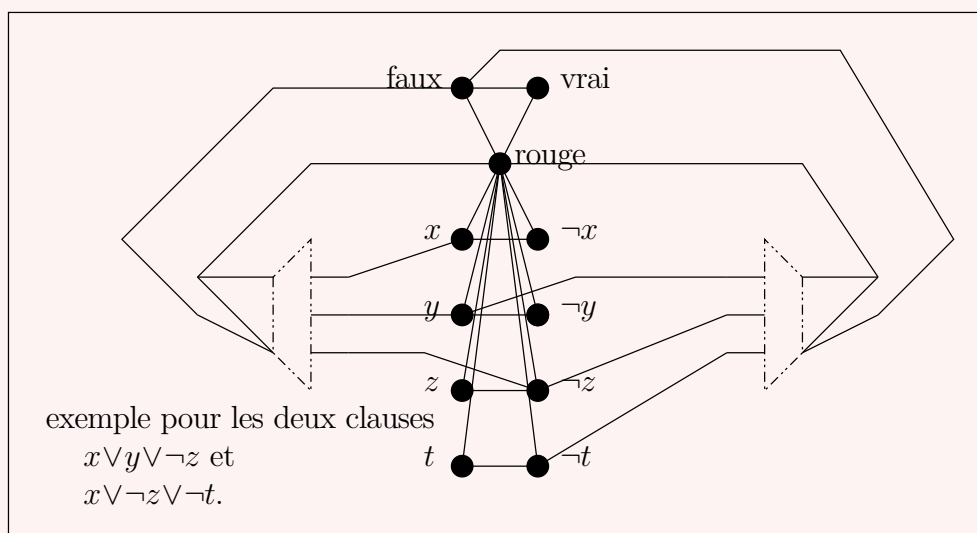
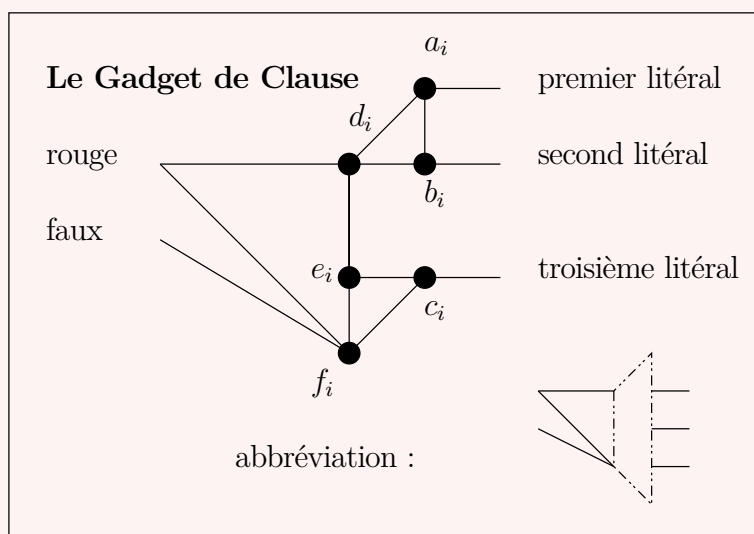


solution.

Réduisons 3-SAT à 3-COL. Pour une entrée φ de 3-SAT, on construit le graphe G_φ comme suit :

- G_φ a deux sommets x et $\neg x$ pour chaque variable x apparaissant dans φ ;
- G_φ a également trois sommets spéciaux, *rouge*, *vrai* et *faux* ;
- *rouge*, *vrai* et *faux* forment un triangle ;
- il y a une arête entre chaque sommet x et $\neg x$;
- x et *rouge* sont adjacents ;
- $\neg x$ et *rouge* sont adjacents ;
- pour tout clause $C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i$ de φ , on ajoute un « gadget » se composant de 6 sommets $a_i, b_i, c_i, d_i, e_i, f_i$ où a_i, b_i, d_i et c_i, e_i, f_i forment des triangles, d_i et e_i sont adjacents. Ce gadget est « branché » comme suit sur le reste du graphe ; d_i et *rouge* sont adjacents, f_i est adjacents à *rouge* et à *faux*, ℓ_1^i est adjacent à a_i , ℓ_2^i est adjacent à b_i et ℓ_3^i est adjacent à c_i .

Un exemple de réduction et de gadget sont donnés ci-dessous.



On va maintenant montrer que si G_φ est 3-coloriable alors φ est satisfaisable. Supposons que G_φ soit colorié en « rouge », « vrai » et « faux » de telle sorte que deux sommets adjacents aient des couleurs différentes. On peut alors supposer sans perte de généralité que *rouge* est colorié en « rouge », *vrai* en « vrai » et *faux* en « faux » comme *rouge*, *vrai*, *faux* forment un triangle. Comme chaque sommets x et $\neg x$ forment un triangle avec *rouge*, il faut que l'un des deux soit colorié en « vrai », l'autre en « faux ». Notons maintenant que pour chaque gadget, le sommet f_i forme un triangle avec *rouge* et *faux* : il doit donc être colorié en « vrai ». Il y a donc deux possibilités pour c_i et e_i qui forment un triangle avec f_i : soit (1) c_i est colorié « faux » et e_i en « rouge » ou bien (2) c_i en « rouge » et e_i en « faux ». Dans le premier cas, cela implique que le sommet ℓ_3^i étant adjacent à un sommet

colorié en « rouge » (le sommet *rouge*) et un sommet colorié « faux » doit être colorié en « vrai ». Dans le second cas, cela implique que d_i est colorié en « vrai », donc que a_i ou b_i est colorié en « faux », c'est-à-dire que ℓ_1^i ou ℓ_2^i est colorié en « vrai ». Dans chacun des cas, on a bien que l'un des trois sommet ℓ_1^i , ℓ_2^i ou ℓ_3^i est colorié en « vrai ». On voit donc bien qu'un coloriage valide induit une valuation satisfaisant la formule, puisque au moins un élément de chaque clause est vrai.

Pour la réciproque, il suffit de vérifier que l'on peut bien étendre le coloriage induit par une valuation correcte de la formule aux gadgets.

P différent de NP et NP-complétude.

On ne sait pas si P est véritablement différent de NP, c'est une conjecture en théorie de la complexité.

En pratique il y a maintenant une très large collection de problèmes NP-complets pour lesquels personne ne connaît d'algorithme polynomial.

Retour à notre petit soucis.

Si je trouve un algorithme polynomial, super je suis content.

Que faire si je réfléchis très fort et que je ne trouve pas d'algorithme polynomial pour mon problème favori? En particulier quand ce problème est dans NP?

Si je montre que le problème est NP-complet, alors je sais que personne ne connaît à l'heure actuelle d'algorithme polynomial pour mon problème.

5.6 Complexité : au delà de P et NP

D'autres classes. Quelques classes de complexité usuelles.

- Espace logarithmique, polynomial, exponentiel (déterministe ou pas)
- Temps polynomial, exponentiel (déterministe ou pas)

En dehors des inclusions faciles à voir de ces classes, on ne sait en général pas les séparer.

Il y a beaucoup d'autres classes de complexité. Vous pouvez aller explorer ce zoo ici :

https://complexityzoo.uwaterloo.ca/Petting_Zoo

Comparer les classes de complexité.

Non-déterminisme pas moins puissant que déterminisme Une machine déterministe est un cas particulier d'une machine non-déterministe $\text{Time}(f(n)) \subseteq \text{NTime}(f(n))$.

Temps vs Espace. Une machine ne peut pas écrire plus que son temps d'exécution $(\mathbb{N})\text{Time}(f(n)) \subseteq (\mathbb{N})\text{Space}(f(n))$.

Se passer du non déterminisme? On peut simuler avec du *backtrack* une machine non déterministe en temps par une machine déterministe (pour un coût exponentiel). $\text{NTIME}(f(n)) \subseteq \text{TIME}(c^{f(n)})$ (la constante c dépend de la machine initiale mais pas de n).

Quelques théorèmes importants. On peut relativiser la preuve de diagonalisation de l'arrêt sur des familles de machines avec horloge ou mètre (ceci nous donne des hiérarchies en temps et en espace).

Théorème (Hiérarchie en temps, Hartmanis et. al. 1965). *Si $f(n)$ est une fonction superlinéaire raisonnable alors $\text{TIME}(f(n))$ est strictement contenue dans $\text{TIME}(f(2n+1)^3)$*

Ce résultat a un analogue pour l'espace

Théorème (Hiérarchie en espace, Hartmanis et. al. 1965). *Si $f(n)$ est une fonction raisonnable alors $\text{SPACE}(f(n))$ est strictement contenue dans $\text{SPACE}(f(n)\log(f(n)))$*

On peut tester l'existence d'un chemin dans un graphe à n sommets avec un espace $\log n \times \log n$ (Théorème de Savitch).

Conséquence. *Si $f(n) \geq \log n$ est une fonction raisonnable alors $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f(n) \times f(n))$.*

Quelques classes de complexité sont représentées en Figure 5.1. On sait que L est strictement inclus dans Pspace et que P est strictement inclus dans Exptime.

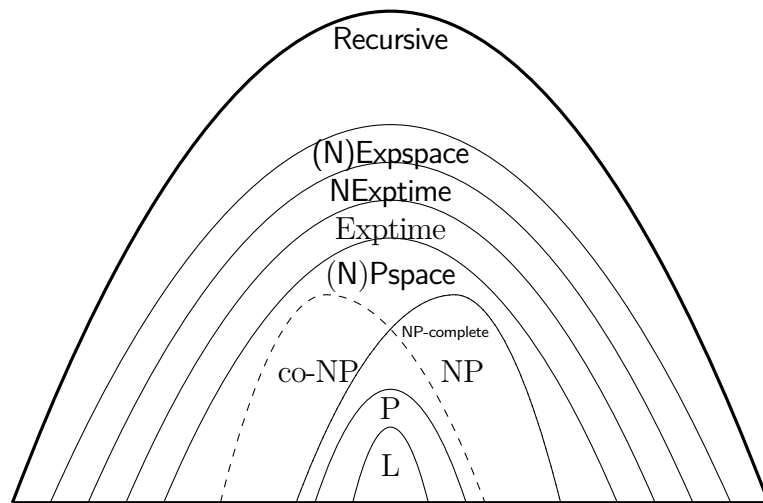


FIGURE 5.1 – Quelques classes de complexité et leur inclusion