Fondements de l'informatique

Florent Madelaine

23 février 2021

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 Lemme de la pompe 22

3 Turing **27**
3.1 Turing avec JFLAP 28
3.2 Définition 30
3.3 Robustesse 31

II Comparer **33**

4 Décidabilité et Indécidabilité **35**
4.1 Définitions. 35
4.2 Indécidabilité (argument dénombrement) 36
4.3 Machine de Turing Universelle. 37
4.4 Indécidabilité (problème de l'arrêt) 37
4.5 autres problèmes indécidables 38

5 Complexité **39**
5.1 Ressources, dont le temps. 39
5.2 Mesurer le temps 39
5.3 classes de complexité robustes 40
5.4 Vers la notion de réduction. 42
5.5 NP-complet 42

5.6 Complément complexité 46

III Divers **49**

6 Codes correcteurs **51**
6.1 Transmission et stockage de l'information 51
6.2 Distance minimale et efficacité garantie d'un code 55
6.3 Codes linéaires 57
6.4 Correction automatique 65
6.5 Construction du tableau standard 68
6.6 Retour sur la distance minimale du code 69
6.6.1 Colonnes de la matrice de contrôle 69
6.6.2 Calcul de la distance minimale du code 69

Première partie

Jouer avec les modèles

Chapitre 1

Que veut-on calculer ?

Le cas des fonctions discrètes $f : \mathbb{N} \to \mathbb{N}$ 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 entrevoir deux : l'automate (fini) et la machine de Turing. Avec JFLAP, un outil pédagogique pour faciliter la compréhension de modèls de calculs, vous pouvez en découvrir d'autres. Il eexiste 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 parenthésés [1].

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 nombreuse

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

JFLAP [2] 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.

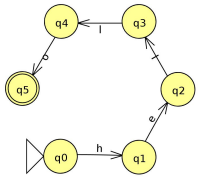
Prise en main : simuler

1. Lancez JFLAP et ouvrez l'automate sauvegardé ici `https://www.lacl.fr/fmadelaine/Download/M1Droit/Automates/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).

2. `http://www.jflap.org/`

Hello world

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



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

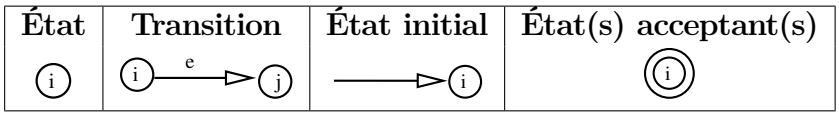
Les automates finis

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



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

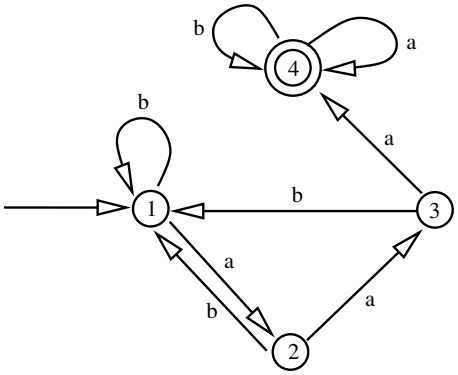
- `Donnée` : w
- `Début` : État initial
- `Lire` w `de gauche à droite lettre par lettre en suivant les transitions`
- `Fin` : mot terminé / pas de transition possible
- `Réponse` : `Accepté / Refusé`

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

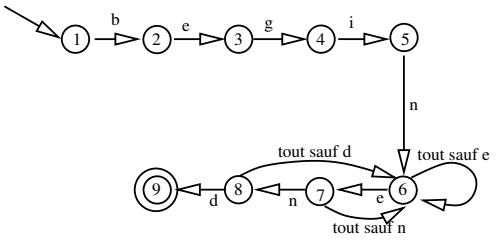
- Alphabet $\Sigma = \{a,b\}$
- Ensemble (fini) d'états $Q = \{1,2,3,4\}$
  - État initial $q_0 = 1$
  - État(s) acceptant(s) $Q_A = \{4\}$

- **Table de transition** δ

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

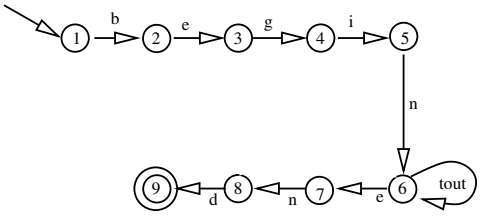
État de rebut

- 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 $abcabcabc...$;
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 régulière [1]

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

.

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 $\overline{n}$ une **représentation binaire** de ce nombre, avec éventuellement des zéros à gauche. Par exemple, si $n=$ cinq, on peut avoir $\overline{n}=101$ ou bien $\overline{n}=000101$, etc. On a alors $\overline{2n}=\overline{n}0$ et $\overline{2n+1}=\overline{n}1$. 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 $n\equiv 1 \mod 3$. Ajouter un 0 à droite de w, signifie qu'on multiplie n par 2 ; On calcule alors que $2n\equiv 2 \mod 3$.

Regardons maintenant le cas où le mot binaire w code pour un nombre n tel que $n\equiv \mathbf{2} \mod 3$. Le mot binaire $w1$, code pour $\mathbf{2}n+\mathbf{1}$ et on peut calculer que

$$2n+1\equiv \mathbf{2}(\mathbf{2})+\mathbf{1} \mod 3\equiv 5 \mod 3\equiv 3+2 \mod 3\equiv 0+2 \mod 3\equiv 2 \mod 3$$

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 : $n \equiv \mathbf{2} \mod 3$ veut dire $n = 3k + \mathbf{2}$. Donc $\mathbf{2}n + \mathbf{1} = \mathbf{2}(3k + \mathbf{2}) + \mathbf{1} = 6k + \mathbf{2}(\mathbf{2}) + \mathbf{1} = 6k + 3 + 2 = 3(k + 1) + 2$. 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 cete 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	$2n$	$2n+1$
$\equiv 0 \mod 3$	$\equiv 0 \mod 3$	$\equiv 1 \mod 3$
$\equiv 1 \mod 3$	$\equiv 2 \mod 3$	
$\equiv 2 \mod 3$		$\equiv 2 \mod 3$

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, complètez la table de transition ci-dessous :

	0	1
zéro	*zéro*	*un*
un	*deux*	
deux		*deux*

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 :

$$\begin{cases} n & \neq 1 \mod 3 \\ n & = 2 \mod 5 \end{cases}$$

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 ambigüe. 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. $L=\{aa,bb\} \rightsquigarrow L^*=\{\varepsilon,aa,bb,aaaa,aabb,bbaa,bbbb,aaaaaa,aaaabb,aabbaa,aabbbb,...\}$

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

à compléter

(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

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.

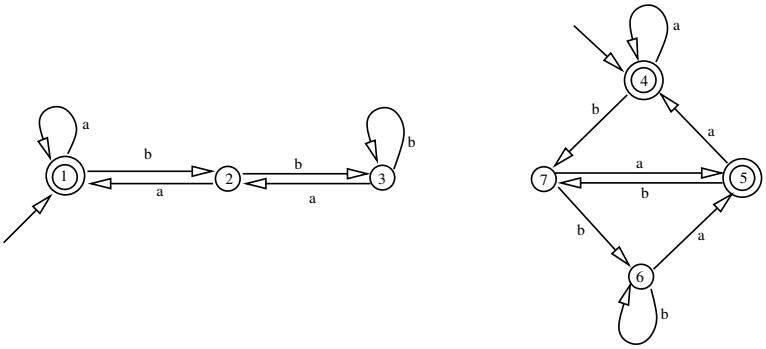
Vocabulaire

⋆ 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 ?



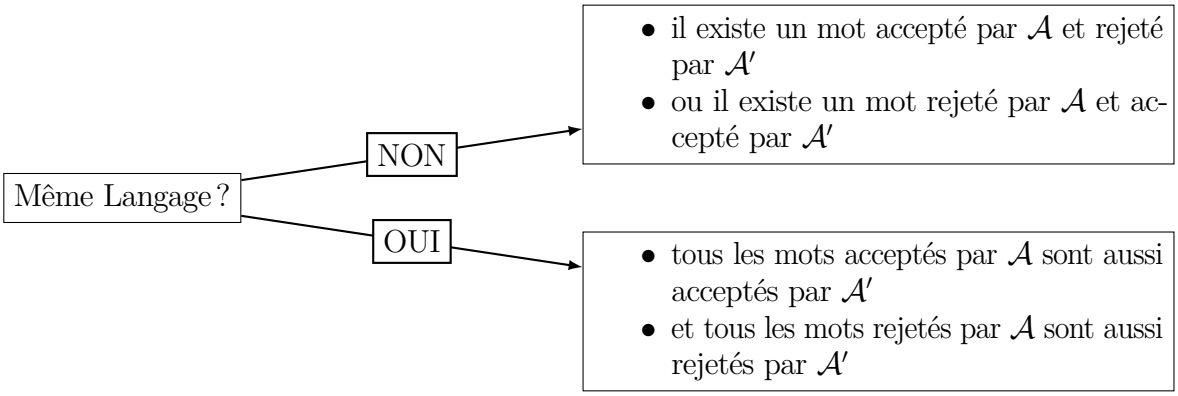
Définition. Deux automates qui acceptent le méme langage s'appellent des automates **équivalents**

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

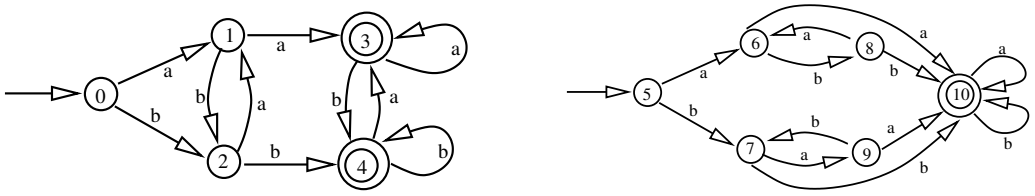
- On suit des chemins "parallèles" dans les deux automates **déterministes** $\mathcal{A}$ et $\mathcal{A}'$.
- Si on rencontre un couple d'états (q,q') avec q acceptant et q' non acceptant (ou le contraire), alors on a trouvé un mot w accepté par $\mathcal{A}$ et rejeté par $\mathcal{A}'$ (ou le contraire). 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
(1,4)	(1,4)	(2,7)
(2,7)	(1,5)	(3,6)
(1,5)	(1,4)	(2,7)
(3,6)	(**2**,**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)	(3,10)	(2,8)
(2,7)	(1,9)	((4,10))
(3,10)	(3,10)	(4,10)
(2,8)	(1,6)	(4,10)
(1,9)	(3,10)	(2,7)
(4,10)	(3,10)	(4,10)

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

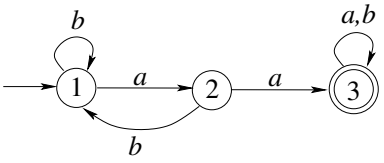
Vocabulaire

⋆ Automates équivalents
⋆ Méthode pour vérifier si 2 automates déterministes sont équivalents

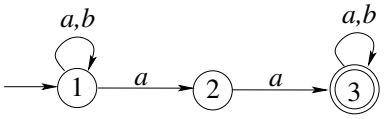
2.4 Déterminisation

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**.



$\mathcal{A}$ déterministe



$\mathcal{A}'$ non-déterministe

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

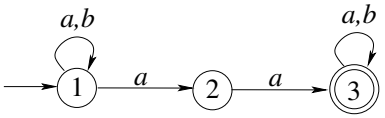
Réponse. À partir d'un automate non-déterministe qui reconnaît un langage L, on peut toujours calculer un automate **déterministe** qui reconnaît le méme langage L.

Idée générale

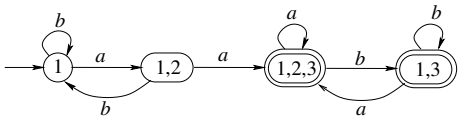
L'automate déterministe que l'on va construire sera composé de *super-états* qui vont *simuler* des ensembles d'états de l'automate initial. La méthode utilisée est la suivante :

- 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.



super-états	a	b
$\rightarrow\{1\}$	$\{1,2\}$	$\{1\}$
$\{1,2\}$	$\{1,2,3\}$	$\{1\}$
$\underline{\{1,2,3\}}$	$\{1,2,3\}$	$\{1,3\}$
$\underline{\{1,3\}}$	$\{1,2,3\}$	$\{1,3\}$



Remarques.

- Les automates non-déterministes sont aussi naturels et acceptables que les automates déterministes
- Mais il arrive qu'on soit obligé de déterminiser 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

⋆ Automate déterministe et non-déterministe
⋆ Méthode pour déterminiser un automate (construire un automate déterministe équivalent)

2.5 Lemme de la pompe

Nous n'avons pas vu beaucoup de détails sur les expressions régulières mais 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 ?

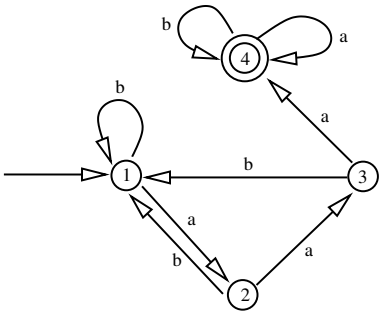
Si le langage L est donné par une **expression régulière**, on sait que la réponse est **Oui**, et on sait même **construire** un automate fini qui reconnaît ce langage. Mais si la réponse est **Non**, autrement dit s'il n'existe **aucun** automate fini qui reconnaît le langage L, comment le **prouver**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

Outil pour le Non.

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

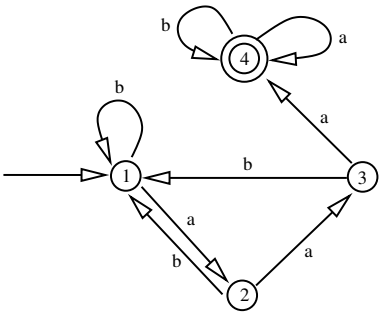
Par conséquent, si un langage **ne vérifie pas** cette propriété, il est **impossible** qu'il soit un langage régulier. Malheureusement, cet outil a ses limites puisqu'il existe des langages qui ne sont pas réguliers et pour lesquels la propriété reste vraie. Dans ce cas il existe une autre technique 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.



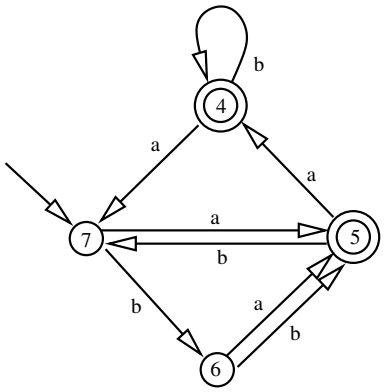
- $aabbaaaba$ accepté
- $aabaabbaaaba$ accepté
- $aabaabaabaabbaaaba$ accepté
- N'importe quel mot de la forme $(aab...aab)baaaba$ accepté
- Car il y a un cycle $1 \xrightarrow{a} 2 \xrightarrow{a} 3 \xrightarrow{b} 1$
- On dit qu'on peut **pomper** aab dans le mot $aabbaaaba$

Pomper ici ou pomper ailleurs.



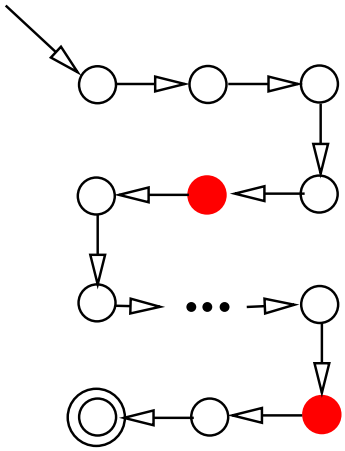
- *aabbaaaba* accepté
- *aabbbaaaba* accepté
- *aabbbbbbaaaba* accepté
- N'importe quel mot de la forme *aab*(*b*...*b*)*aaaba* accepté
- Car il y a un cycle $1 \xrightarrow{b} 1$
- On peut aussi **pomper** *b* dans le mot *aabbaaaba*

Peut-on toujours pomper ?



- *bbbaabaa* accepté
- Peut-on pomper quelque chose quelque part ?
- *bbbaabaa* (cycle $5 \xrightarrow{b} 7 \xrightarrow{a} 5$)
- ou *bbbaabaa* (cycle $5 \xrightarrow{a} 4 \xrightarrow{b} 4 \xrightarrow{a} 7 \xrightarrow{a} 5$)
- *bb* accepté
- Peut-on pomper quelque chose quelque part ?
- **Non** : aucune partie de *bb* ne correspond à un cycle sur l'automate
- Le mot *bb* est trop court !

Dans quels mots peut-on pomper ?



- On peut pomper chaque fois qu'une partie d'un mot accepté correspond à un **cycle** sur l'automate
- Dès que la **longueur** d'un mot accepté est **plus grande que le nombre d'états** de l'automate, on doit forcément repasser par (au moins) un état en lisant ce mot
  $\Longrightarrow$ il y a un cycle
- Finalement, dès que la longueur d'un mot accepté est plus grande que le nombre d'états de l'automate, on est sûr de pouvoir **pomper** une partie du mot

Théorème (**lemme**[2] **de la pompe**). *Soit L un langage reconnu par un automate $\mathcal{A}$ avec n états.*

Alors on peut pomper quelque chose dans tout mot de L de longueur $\geq n$.

Autrement dit :

Tout mot w de L de longueur $\geq n$ se décompose sous la forme $w = uxv$ de telle sorte que :

1. $|ux| \leq n$

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.

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

Remarque. L'expression ci-dessus *ressemble* a une expression régulière mais ce n'en ai pas une. On a le droit d'écrire a^k pour k **fixé**, qui est une abréviation pour $aa...a$ 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, $w' = uxxv$ **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 jeu [3] entre 2 joueurs, qui pour la logique des prédicats correspondent aux deux quantificateur universel ($\forall$) et existentiel ($\exists$). Le premier Abélard (pour for**A**ll, 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 **E**xists, en notant que typographiquement le $\exists$ est un E retourné) essaye au contraire de montrer que l'énoncé est vrai.

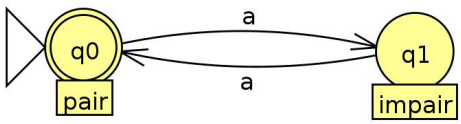
En particulier, quand on applique le lemme de la pompe pour un langage donné, on peut voir la preuve comme un tel jeu. C'est ce qui se passe dans l'activité lié au lemme de la pompe dans le logiciel JFLAP.

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 $\{a^nb^n|n>0\}$ 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.

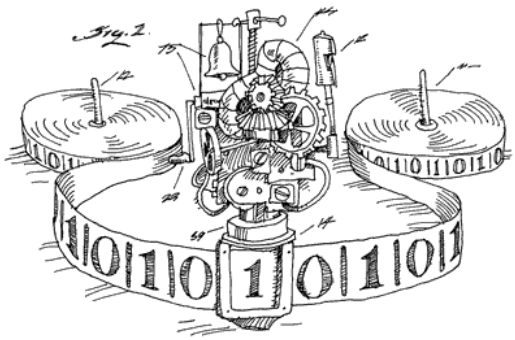


FIGURE 3.1 – La machine de Turing et son ruban arbitrairement grand sur lequel on peut écrire.

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 brouillons [1]. C'est un mathématicien qui a eu une influente forte en informatique, malgré une vie trop courte. En droit britannique, une loi porte son nom [2].

1. Sa page wikipedia comporte de nombreux liens, dont cette url `http://www.turingarchive.org/`.
2. `https://en.wikipedia.org/wiki/Alan_Turing_law`.

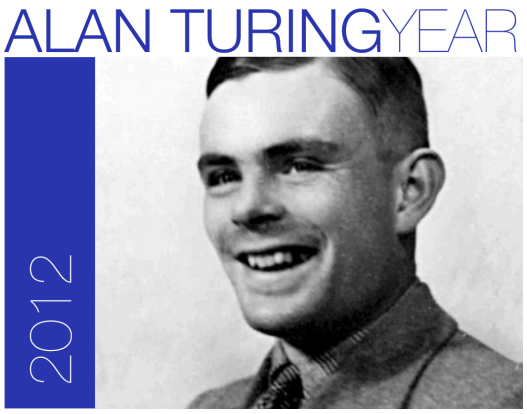


FIGURE 3.2 – Alan Turing (1912-1954).

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 acceptant [3].

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 □ 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 □ pour une transition, il suffit de ne rien saisir dans cette cellule pour la transition et d'appuyer sur entrée.

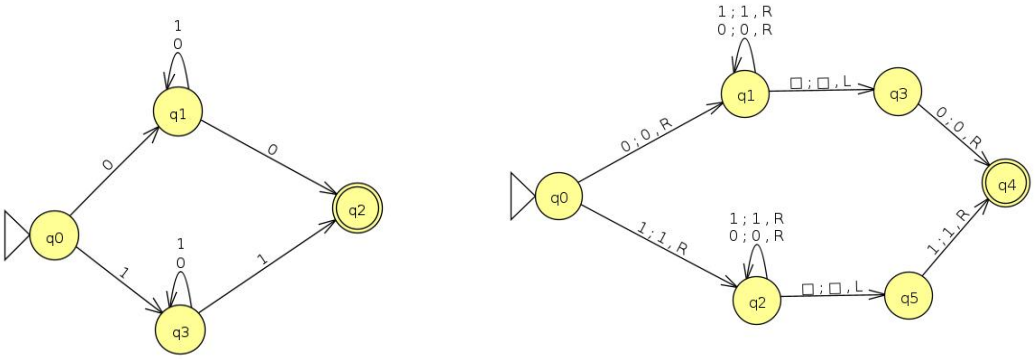
Exercice 6.
1. Écrire un programme qui lit les caractères du mot d'entrée de la gauche vers la droite et s'arrête en acceptant.
2. Même question pour un programme qui accepte si et seulement si la première et la dernière lettre sont les mêmes.
3. Même question pour les mots qui sont des *palindromes* (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.

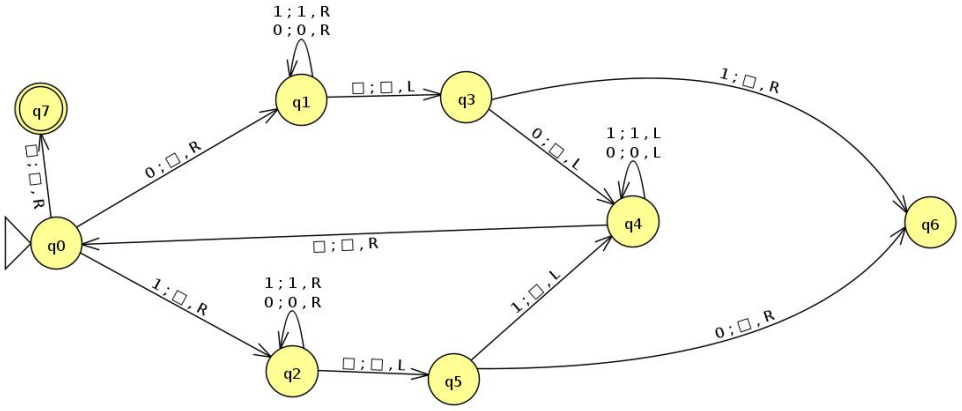
Si on disposez d'une machine de Turing à **2 rubans**, on pourrait : recopier le premier ruban sur le second ; ramener la tête de lecture du premier ruban tout à gauche ; puis décaler la tête du premier ruban vers la droite et celle du second ruban vers la gauche en vérifiant une par une que les lettres 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 avions **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 suivant 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 $Q \backslash F$ et d'une lettre de l'aphabet 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 veux 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 générales 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. □

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 $s_0 = 0, s_1 = 1, s_2, 01, s_3 = 10, s_4 = 11, \ldots$.

On suppose par l'absurde que l'ensemble de tous les langages sur $\{0,1\}$ est dénombrable et donc qu'on peu énumérer sous forme de liste $L_0, L_1, L_2, \ldots$ (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 $s_j \in L \iff s_j \notin L_j$. □

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)**.**

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

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

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. □

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 $f : \mathbb{N} \to \mathbb{N}$) 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, $f(n)\leq c.g(n)$.
- $g(n)=\Omega(f(n))$ 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 $2+n+c.f(n)$.*

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. □

Définition. On note $\texttt{TIME}(f(n))$ 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 $\texttt{TIME}(f(n))=\texttt{TIME}(2^{f(n)})$*

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 $N\texttt{TIME}(f(n))$ 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.

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

(la machine de Turing est déterministe)

***Calcul* en temps Polynomial ≈ calcul efficace.**

$$\texttt{NP} = \bigcup_{c \geq 1} \texttt{NTime}(\texttt{n}^\texttt{c})$$

(la machine de Turing peut être non-déterministe)

***Vérification* en temps Polynomial ≈ 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 : $(x \vee y \vee z) \wedge (x \vee \neg t \vee \neg u \vee v)$...

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.

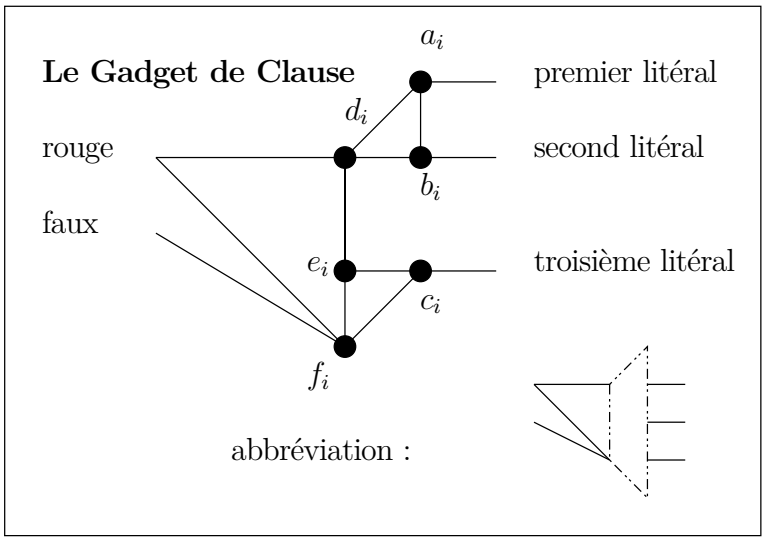
La page wikipedia propose une preuve complète : `https://en.wikipedia.org/wiki/Cook-Levin_theorem` □

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.

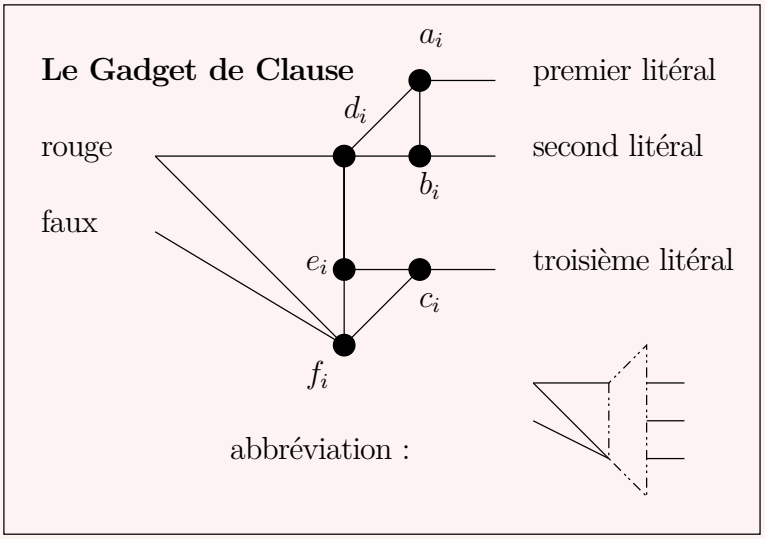


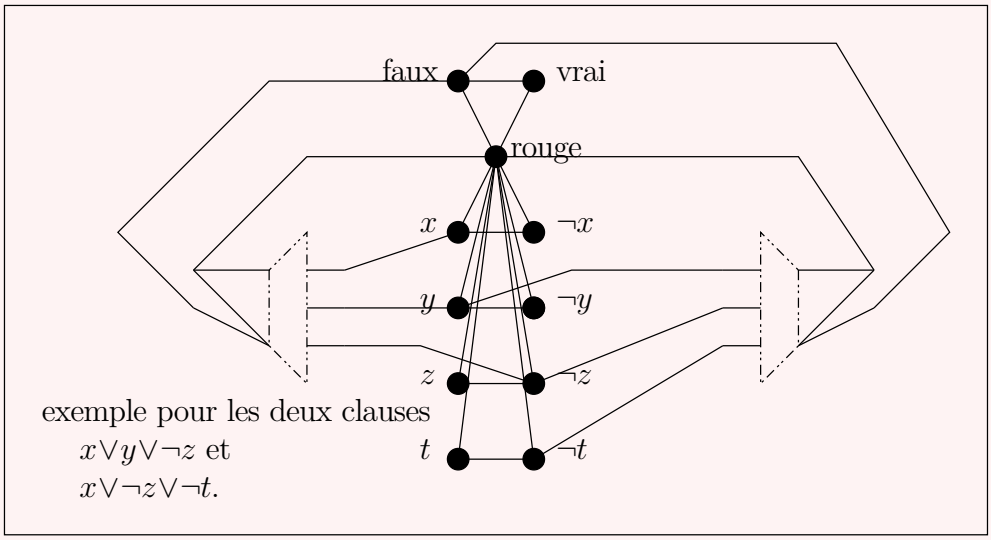
solution.

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 connais 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 allez 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 $\texttt{Time}(f(n)) \subseteq \texttt{NTime}(f(n))$.

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

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). $\mathtt{NTime}(f(n)) \subseteq \mathtt{Time}(c^{f(n)})$ (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* $\mathtt{Time}(f(n))$ *est strictement contenue dans* $\mathtt{Time}(f(2n+1)^3)$

Ce résultat a un analogue pour l'espace

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

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* $\mathtt{NSpace}(f(n)) \subseteq \mathtt{Space}(f(n) \times f(n)))$.

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.

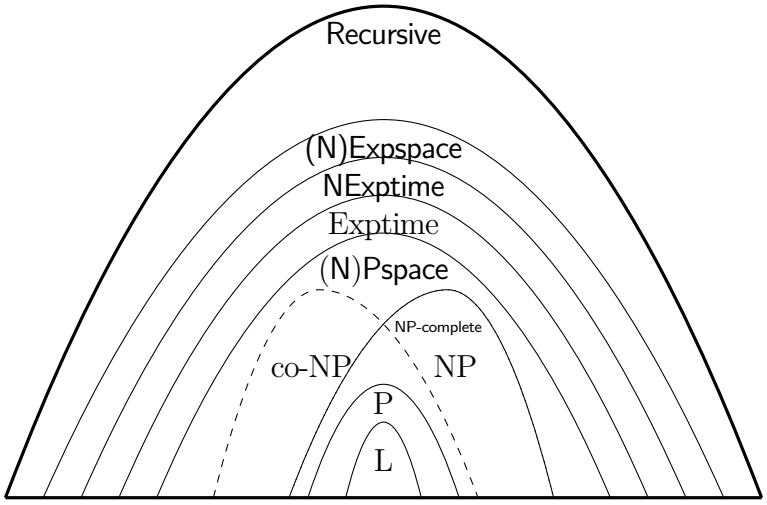


FIGURE 5.1 – Quelques classes de complexité et leur inclusion

Troisième partie

Divers

Chapitre 6

Détecter et corriger des erreurs

6.1 Transmission et stockage de l'information

Les Canaux de transmission et supports de stockage sont forcément imparfaits. Lors du transfert et/ou du stockage d'information, des erreurs surviennent. On souhaite détecter les erreurs de transmission et/ou de stockage et les corriger. C'est le rôle des *codes détecteurs et correcteurs d'erreurs*. On utilise une *stratégie de surcodage*. Plus efficace que de recommencer trop fréquemment la transmission ou de stocker l'information en la dupliquant.

Le Canal Binaire Symétrique

Nous utilisons le modèle idéalisé et quelque peu simpliste du *canal binaire symétrique* paramétré par une probabilité $0 \leq p \leq 1$ fixée.

Chaque bit b transmis est mal envoyé, c'est-à-dire transformé en $b+1 \mod 2$ avec probabilité p.

Les erreurs sont indépendantes au cours du temps [1].

Exemple.
- probabilité de mal envoyer 2 bits de suite : $p \times p$

  (car les deux événements sont indépendants)
- probabilité de mal envoyer 1 bit, puis envoyer correctement le suivant, puis un autre à nouveau mal : $p \times (1-p) \times p$

Codage et Décodage

Avant émission, l'information est surcodée par ajout de *bits de contrôle*. Le mot binaire obtenu est le *message émis*. C'est le *codage*. Après réception, à partir du

1. Cette hypothèse n'est pas très réaliste : une poussière ou une rayure sur un cdrom va causer des soucis sur des bits proches ; une interférence pendant une transmission va affecter des bits consécutifs. Nous choisissons ce modèle simple pour des raison pédagogiques.

message reçu, en utilisant les bits de contrôle, on *détecte* les erreurs, puis on *corrige* si possible le message reçu pour retrouver l'information initiale. C'est le *décodage*. Voir Figure 6.1, pour une illustration.

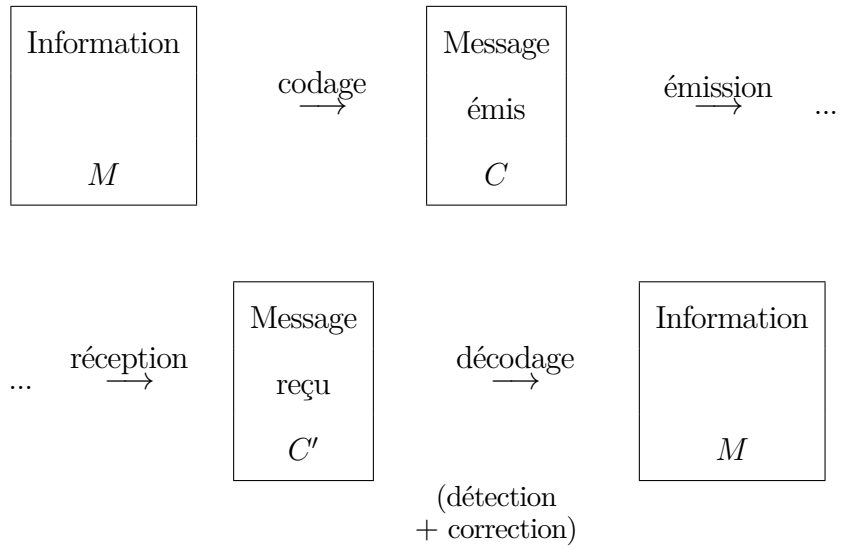
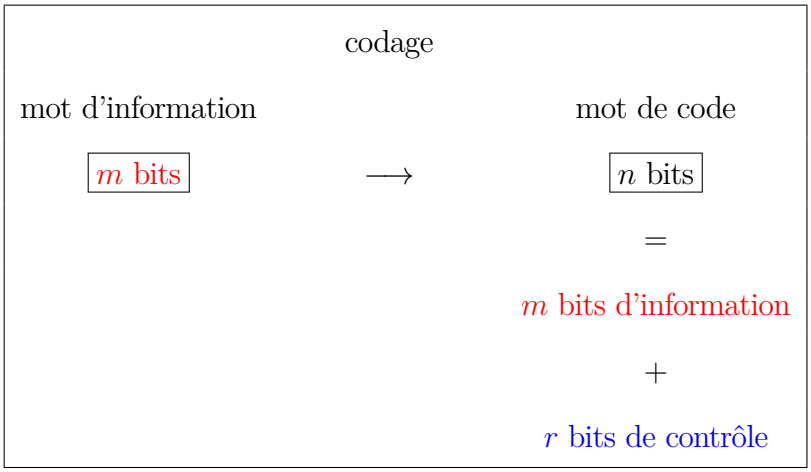


FIGURE 6.1 – Codage et Décodage

Codage par blocs

- L'information est découpée en **blocs de longueur fixe** :



- *Redondance* : $r = n - m =$ nombre de bits de contrôle
- *Rendement* : $\rho = \frac{m}{n} = \frac{m}{m+r}$

Le code de parité : un code pour détecter mais pas corriger

Le principe de ce code correspond à l'idée employée pour la clé RIB ou la clé INSEE.

On ajoute 1 seul bit de contrôle qui est la somme des autres bits modulo 2. Si il y a 1 erreur (et plus généralement un nombre impair d'erreurs) alors on peut le détecter au décodage Il suffit de calculer la somme des bits reçus modulo 2 : si on obtient 1, alors il y a nécessairement une erreur. Dans ce cas, $r=1$

Exemple. (Pour $m=4$).

Info. 1001	Info. 1001
Code. 10010	Code. 10010
Reçu. 10110	Reçu. 10111
Somme non nulle modulo 2 : erreur détectée.	Somme nulle modulo 2 : 2 erreurs mais pas détectées.

Le code par répétition : un code pour détecter et corriger

On répète chaque bit du message k fois (k impair). Si il y a strictement moins de k erreurs alors on peut le détecter puisque tous les bits reçus devraient être les mêmes et que ce n'est pas le cas. Pour la correction, on corrige au bit le plus fréquent (toujours possible puisque k est impair). Dans ce cas, $r=(k-1)m$

Exemple. (Pour $m=1$ et $k=3$ répétitions, soit $n=3$ et $r=2$).

Info. 1	Info. 1
Code. 111	Code. 111
Reçu. 110	Reçu. 010
Pas tous égaux : erreur détectée.	Pas tous égaux : erreur détectée.
On corrige à 1 (bit le plus fréquent).	On corrige à 0 (bit le plus fréquent).

La perfection n'est pas possible

On vient de voir sur nos deux exemples que

- on ne peut pas espérer détecter toutes les erreurs ; et,
- on ne peut pas espérer corriger correctement à chaque fois.

On a donc plusieurs cas à considérer (voir Figure)

Hypothèse de travail Pour $0 \leq k < k' \leq n$, il est beaucoup plus probable de recevoir un message avec k erreurs qu'avec k' erreurs Quand on ne détecte aucune erreur dans le message reçu, il est très probable qu'il ne contient effectivement aucune erreur Quand on corrige un message erroné par le mot de code le plus proche, il est très probable qu'il soit corrigé correctement

Remarque. Pour tous les exemples réalistes, ceci est vrai. On verra que c'est presque toujours le cas dans notre cadre de travail théorique (le canal binaire symétrique).

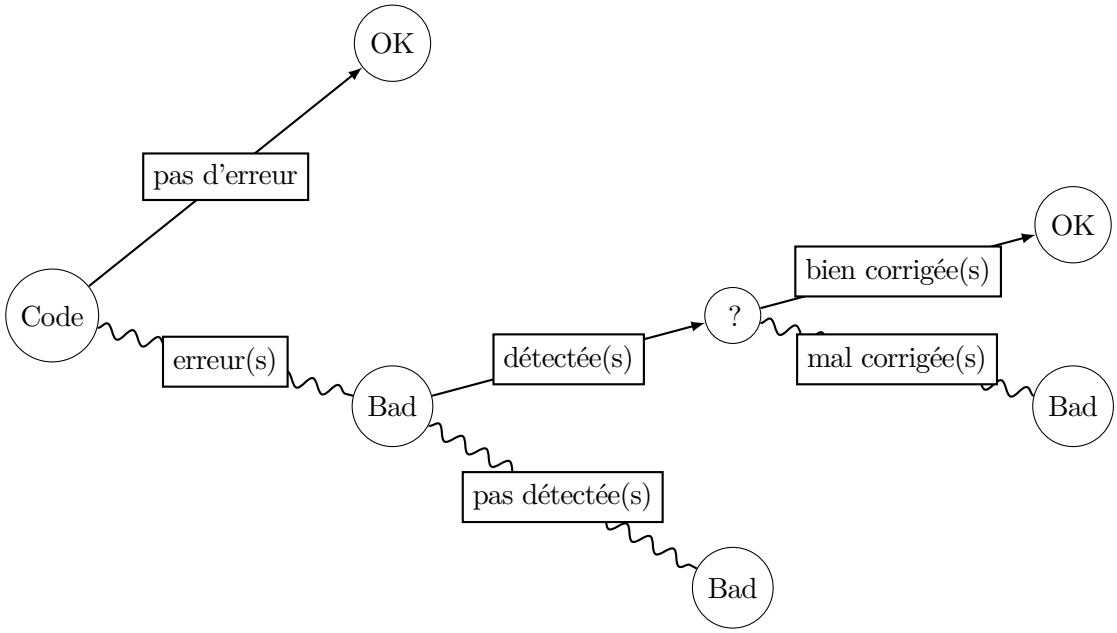


FIGURE 6.2 – Différents cas de figures lors du décodage en fonction de l'occurrence ou non d'erreurs lors de la transmission.

Exemple. Considérons le code de parité sur un message de 3 bits d'information, le code transmis consiste en 4 bits.

Soit p la probabilité d'avoir une erreur sur un bit (on choisira $p=\frac{1}{100}$ dans nos calculs).

La probabilité d'avoir aucune erreur sur les 4 bits est $(1-p)^4$ car il y a d'après notre modèle une erreur sur un bit ou pas indépendemment des autres bits. On fait donc le produit des probabilité des événements « ne pas avoir d'erreur sur le bit 1,2,3 et 4 », dont la valeur est $1-p$. Si on note X le nombre d'erreur, on a donc : $P(X=0)=(1-p)^4=(\frac{99}{100})^4\approx 96\%$

Pour une erreur, il y a 4 positions possible pour l'erreur d'une probabilité p et l'absence d'erreur aux trois autres positions est $(1-p)^3$ on a donc : $P(X=1)=4*p*(1-p)^3=4*\frac{1}{100}(\frac{99}{100})^3\approx 3.88\%$

De même, $P(X=2)=6*p^2*(1-p)^2\approx 0.06\%$ et $P(X=3)=4*p^3*(1-p)\approx 0{,}0004\%$ et $P(X=4)=p^4=0{,}0000001\%$.

Si il n'y a pas d'erreur tout se passe bien on décode (dans $\approx 96\%$ des cas), si il y a un nombre impair d'erreur on le détecte (dans un peu plus de 3.88% des cas), sinon il y a un nombre pair d'erreur (dans $\approx 0.06\%$ des cas qui est presque toujours 1 seule erreur).

Si nous n'avions pas fait de surcodage, nous aurions eu environ 1 bit sur 100 transmis incorrectement (en moyenne au sens de l'espérance en statistique).

Avec notre nouvelle méthode, nous allons transmettre plus de bits pour transmettre 100 bits d'information : *grosso modo* un tiers de plus puisqu'on ajoute un bit tous les trois bits d'information. Pour 99 bits d'information, on va transmettre 33 paquets de 4 bits (3 d'info, 1 redondant). En moyenne au vu de nos calculs 3.88% de 33 nous donne 1 ou 2 paquets qui vont avoir une erreur qu'on va détecter correctement et on aura retransmission (probablement correcte pour simplifier notre estimation). On va donc transmettre en gros 35 fois 4 bits soit 140 bits pour 99 bits d'information. Un nombre pair d'erreur va se produire dans $\approx 0.06\%$ des paquets de 4 lettres. En moyenne un paquet non corrigé tous les 1666 paquets. Au vu des probabilité calculées au dessus ce sera presque toujours sur 2 bits du paquet (pas forcément ceux d'information mais au moins une erreur dans l'information) : donc tous les 1666 paquets transmis on aura 1 ou 2 bits d'information faux. Ceci représente 1 ou 2 bits faux sur 5000 bits d'information en moyenne.

Notre méthode permet donc de passer d'un canal avec un taux d'erreur de 1 % à un canal qui est plus lent (25 % du temps à envoyer des choses redondantes) mais qui permet de transmettre l'information avec un taux d'erreur de 0,04 %.

En pratique, ajouter un bit de parité tous les 7 bits est une procédure assez standard. Avec un canal avec un taux de 1 % come ci-dessus, la probabilité d'erreur détectée sur un paquet de 8 bits est de 6,6 % contre 0,2 % d'erreurs non détectées (probablement 2 erreurs sur le paquet de 8 bits). Soit 1 ou 2 bits faux tous les 3500 bits d'information en n'envoyant une information redondante seulement 12,5 % du temps.

6.2 Distance minimale et efficacité garantie d'un code

Définition. Soient C_1 et C_2 deux vecteurs de bits. La **distance de Hamming** entre C_1 et C_2 est le nombre de bits qu'il faut changer pour passer de l'un en l'autre. Pour la calculer, on considère le vecteur $D = C_1 \oplus C_2$ (addition des 2 vecteurs modulo 2), qui est le *vecteur des différences* (puisqu'il indique les positions distinctes). La *distance de Hamming* entre C_1 et C_2 est le poids (nombre de 1) de D, c'est-à-dire le nombre de bits dont ils diffèrent.

Exemple. Soient $C_1 = 01010110$ et $C_2 = 11010010$. On a $D = 10000100$ qui est de poids 2. Donc la distance de Hamming entre C_1 et C_2 est de 2.

Définition. La *distance minimale d d'un code* est la plus petite distance de Hamming entre deux mots de code différents. Autrement dit, c'est le plus petit nombre de bits dont diffèrent deux mots de code différents.

Exemple. Code de parité $\mathcal{C}_{4,1}$: tous les mots qui ont un nombre pair de 1 sont des mots de codes et seulement ceux là. Dès qu'on change deux bits on passe sur un autre mot de code. Conclusion : $d = 2$

Le code par répétition $\mathcal{C}_{3,1}$: les seuls mots de code sont 000 et 111. Il faut changer 3 bits pour passer de l'un à l'autre. Conclusion : $d = 3$

La distance minimale du code est une notion essentielle pour mesurer son efficacité

- Messages erronés avec **strictement** moins de d **erreurs** : TOUTES DÉTECTÉES
- Messages erronés avec **strictement** moins de $d/2$ **erreurs** : TOUTES BIEN CORRIGÉES

Pour plus de détails, voir Figures 6.3 et 6.4.

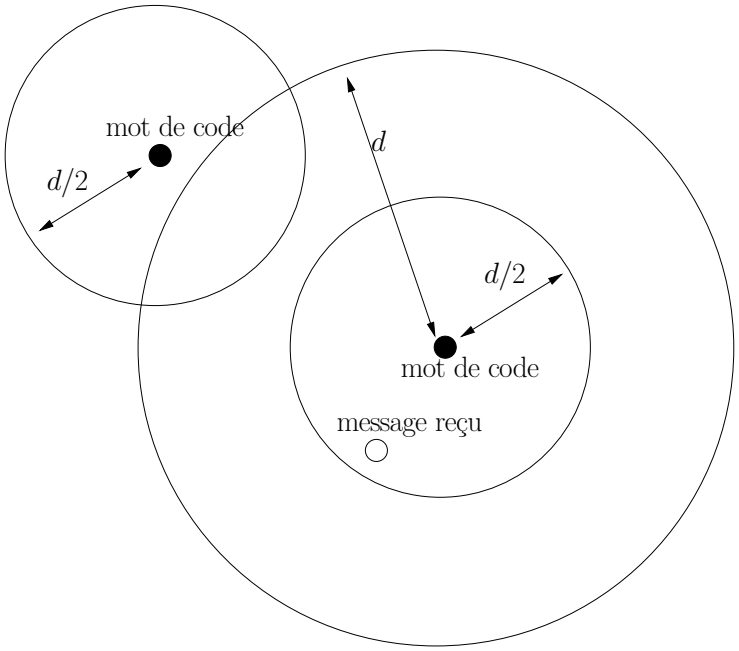


FIGURE 6.3 – Pas d'autre mot de code à distance $< d$; la notion de mot de code le plus proche existe toujours si on est à distance $< \frac{d}{2}$

Vocabulaire

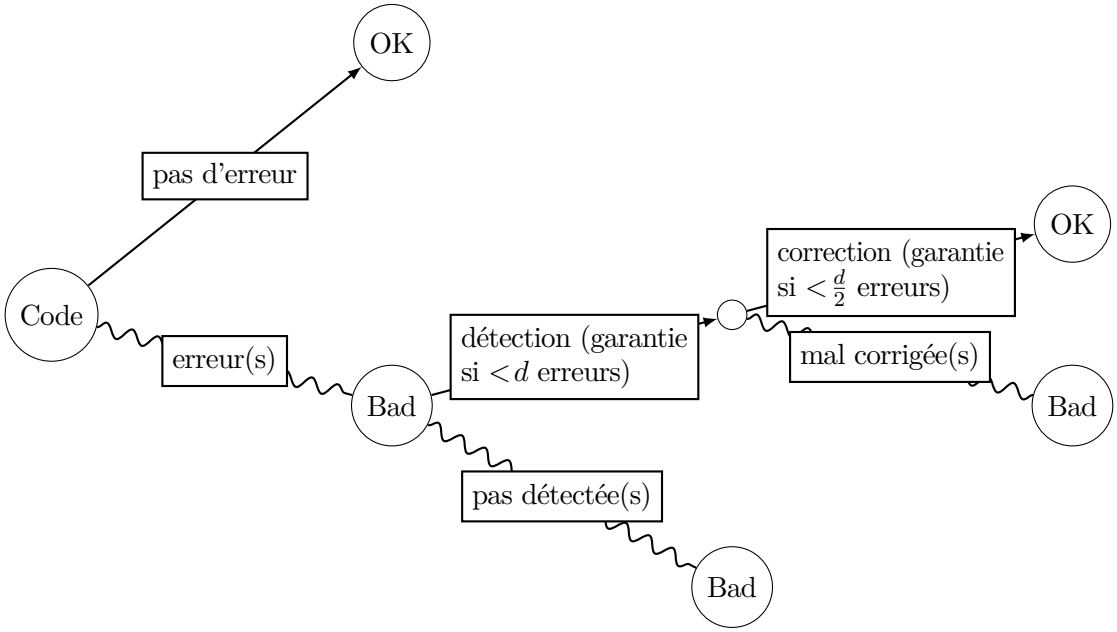


FIGURE 6.4 – la distance minimale permet d'offrir des garanties .

⋆ codes détecteurs et correcteurs d'erreurs
⋆ stratégie de surcodage
⋆ canal binaire symétrique
⋆ bits de contrôle
⋆ message émis
⋆ codage
⋆ détecte
⋆ corrige
⋆ décodage
⋆ redondance
⋆ rendement
⋆ code de parité
⋆ code par répétition
⋆ Hypothèse de travail
⋆ vecteur des différences
⋆ distance de Hamming
⋆ distance minimale d d'un code

6.3 Codes linéaires

Pourquoi choisir un code linéaire ? On peut coder, détecter, corriger en utilisant des matrices. Au niveau calcul, cela veut dire que le programme utilisé stocke peu

d'information et qu'il fonctionne rapidement.

Il y a 2 matrices importantes : La *matrice de codage* G, et la *matrice de contrôle* H.

- On fait $G \star M$ pour obtenir le mot de code C correspondant à M.
- On fait $H \star C'$ pour détecter si C' est un mot de code.

Le produit $\star$ d'une matrice par un vecteur (voir les transparents pour une définition et des exemples) est une opération qui semble un peu artificielle au premier abord. Il s'agit d'une opération de substitution.

Une matrice M_f représente dans notre contexte une **fonction linéaire** f (sur des variables x_1,x_2,...) qu'on applique à un vecteur v.

Le produit de la matrice M_f par un vecteur v correspond à une substitution (remplacer x_1 par v_1, x_2 par v_2 etc) c'à.d. au calcul de $f(v)$. C'est exactement ce qu'on souhaite faire ici.

Remarque. Notre vecteur M est un mot d'information contenant des 0 et des 1 quand on code en faisant $G \star M$. L'opération produit $\star$ correspond à faire la somme des colonnes de G pour lesquelles il y a un bit qui vaut 1 dans le mot M (une fois ces colonnes et 0 fois les autres).

Si par exemple M a un 1 aux deux premières positions et des 0 ailleurs, le mot de code sera la somme des deux première colonnes de la matrice G. On fait la somme ligne à ligne. Comme **on calcule modulo 2**, on aura un 0 dans le résultat à la ième ligne si les deux colonnes contiennent toutes les deux des 0 à la ième ligne ou toutes les deux un 1.

Si le message contient trois 1, on fera la somme des colonnes placées aux positions correspondantes dans la matrice G. Le mot de code aura en ième position, la parité du nombre de 1 de ces colonnes en ième position : 1 si il y a un nombre impair de 1 et 0 sinon.

Voir ci-dessous pour des exemples.

Définition. On appelle *vecteurs élémentaires* e_1,e_2,...,e_m de $\mathbb{R}^m$ les vecteurs formés par les colonnes de la matrice identité I_m. Il s'agit des vecteurs ayant un seul 1, le vecteur e_i ayant un 1 en $i$ème position et des 0 ailleurs.

Remarque. Tout vecteur de $\mathbb{R}^n$ s'écrit comme combinaison linéaire de ces vecteurs. C'est ce que nous faisons sans le savoir en écrivant un vecteur de $\mathbb{R}^n$ sous forme de tuple.

Par exemple c'est ce que vous faites quand vous écrivez les 2 coordonnées d'un point dans l'espace à 2 dimensions quand vous avez étudié et représenté graphiquement des fonctions au lycée.

Dans notre contexte nous avons souvent plus que 2 coordonnées, une coordonné par bit du mot binaire que nous manipulons.

Comment déterminer la matrice de codage d'un code linéaire ?

- Pour une fonction de codage f d'un code linéaire
  - Il suffit de connaître les mots de code des vecteurs élémentaires : $f(e_1), f(e_2) \ldots, f(e_m)$
  - qui forment les colonnes de la matrice G.

C'est une propriété de toutes les applications linéaires, c'est-à-dire de toutes les fonctions qu'on peut représenter par une matrice. Nous verrons plus de détails à ce sujet ultérieurement.

Exemple.

mot de la base		mot de code
1000	$\longrightarrow$	1000101
0100	$\longrightarrow$	0100111
0010	$\longrightarrow$	0010110
0001	$\longrightarrow$	0001011

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

Structure de la matrice génératrice G

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

- Code de type $\mathcal{C}_{7,4}$
  - Longueur des mots de code $n = 7$ bits
  - Longueur des mots d'info $m = 4$ bits
- Matrice génératrice G
  - Nombre de lignes $n = 7$
  - Nombre de colonnes $m = 4$

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Matrice identité I_m
  - Nombre de lignes $m=4$
  - Nombre de colonnes $m=4$

$$Q = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

- Matrice de surcodage Q
  - Nombre de lignes $r=n-m=3$
  - Nombre de colonnes $m=4$

Rôle des matrices I_m et Q

- La matrice identité I_m sert à recopier les bits d'information

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

- La matrice de surcodage Q sert à calculer les bits de contrôle

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

La matrice de contrôle H

Pour un code systématique, on prend la *matrice de surcodage* Q qu'on augmente en lui accolant une matrice identité (avec autant de lignes que la matrice Q, soit r).

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \qquad Q = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \qquad I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \qquad I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad Q = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

Utilisation de la matrice de contrôle

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

matrice de contrôle message reçu *syndrome*

$$H \qquad M \qquad = \qquad S$$

Syndrome d'un message $HM = S$

- Le message reçu est un mot de code : syndrome nul

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

- Le message reçu n'est pas un mot de code : syndrome non nul

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Structure de la matrice de contrôle H

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

- Code de type $\mathcal{C}_{7,4}$
  - Longueur des mots de code $n = 7$ bits
  - Longueur des mots d'info $m = 4$ bits
- Matrice de contrôle H
  - Nombre de lignes $r = n - m = 3$
  - Nombre de colonnes $n = 7$

$$Q = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

- Matrice de surcodage Q
  - Nombre de lignes $r = 3$
  - Nombre de colonnes $m = 4$

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Matrice identité I_r
  - Nombre de lignes $r = 3$
  - Nombre de colonnes $r = 3$

Décomposition du produit $HM = S$

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Rôle des matrices Q et I_r

- La matrice de surcodage Q agit sur les bits d'info reçus et sert à recalculer les bits de contrôle

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

- La matrice identité I_r agit sur les bits de contrôle reçus et sert à les recopier

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}=\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}=\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

- La somme $\oplus$ permet de tester si les deux colonnes obtenues sont identiques

$$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\oplus\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}=\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}\oplus\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}=\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Détection des erreurs

Propriété. M est un mot de code ssi son syndrome $S=HM=\begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$

Message reçu	Syndrome	Contrôle
M	$HM=\begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$	OK
M'	$HM'\neq\begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$	Anomalie

Vocabulaire

⋆ matrice de codage ou matrice génératrice G
⋆ matrice de contrôle H
⋆ vecteurs élémentaires
⋆ matrice de surcodage Q
⋆ syndrome

6.4 Correction automatique

Question Un message reconnu erroné est corrigé en le remplaçant par le mot de code le plus proche Comment calculer ce mot de code ?

Le principe

On prend comme *vecteur de correction* V un vecteur qui a

- Même *syndrome* que le message reçu M
- Poids le plus faible possible

On *corrige* en calculant $C = M \oplus V$ Le mot de code C obtenu est le mot de code le plus proche du message reçu M

Tableau standard

Pour gagner du temps, on précalcule UNE SEULE FOIS, pour chaque syndrome S, un vecteur V de poids le plus faible possible qui a pour syndrome S

- Ce vecteur V servira de vecteur de correction pour TOUS les message reçus M de syndrome S
- On obtient le emphtableau standard du code

Après calcul du syndrome S du message reçu M, le choix du vecteur de correction V s'effectue par simple lecture du tableau standard

Exemple $G = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}$ $H = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$

mot d'info		mot de code
00	$\longrightarrow$	0000
01	$\longrightarrow$	0111
10	$\longrightarrow$	1010
11	$\longrightarrow$	1101

- Longueur des mots d'info $m = 2$ bits
  - $\rightsquigarrow$ Nombre de mots de code = Nombre de mots d'info = $2^m = 4$
- Longueur des mots de code $n = 4$ bits
  - $\rightsquigarrow$ Nombre messages $= 2^n = 16$
- Longueur des syndromes $r = n - m = 2$ bits
  - $\rightsquigarrow$ Nombre de syndromes $= 2^r = 4$

Syndrome	00	01	10	11
Vecteurs	0000	0001	1000	0100
	1010	1011	0010	1110
	0111	0110	1111	0011
	1101	1100	0101	1001
Vecteur(s) de plus faible poids	0000	0001	1000 ou 0010	0100
Vecteur de correction	0000	0001	NC_1	0100

- Syndrome S avec plusieurs vecteurs de plus faible poids k
  - Tous la même probabilité d' être le vecteur d'erreur
- Message reçu M avec syndrome S
  - Plusieurs mots de code à distance k de M
  - Impossible de corriger M par le mot de code le plus proche
- Dans le tableau standard, on écrit NC_k
  - Le syndrome S correspond à des messages *Non Corrigeables* avec k erreurs

Tableau Standard

Syndrome	00	01	10	11
Vecteur de correction	0000	0001	NC_1	0100

Utilisation du tableau standard

- Message reçu : $M = 1100$
- Calcul du syndrome $S = HM = 01$

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

- Lecture du tableau standard

Syndrome	00	01	10	11
Vecteur de correction	0000	0001	NC_1	0100

- Correction

$$C = M \oplus V = 1100 \oplus 0001 = 1101$$

- Message reçu : $M = 1010$
- Calcul du syndrome $S = HM = 00$

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

- Lecture du tableau standard

Syndrome	00	01	10	11
Vecteur de correction	0000	0001	NC_1	0100

- Correction

$$C = M \oplus V = 1010 \oplus 0000 = 1010$$

- Message reçu : $M = 0101$
- Calcul du syndrome $S = HM = 10$

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

- Lecture du tableau standard

Syndrome	00	01	10	11
Vecteur de correction	0000	0001	NC_1	0100

- Correction

$M = 0101$ message erroné non corrigeable (1 erreur)

Vocabulaire

⋆ vecteur de correction
⋆ Tableau standard

6.5 Construction du tableau standard

Méthode pratique :
1. Calculer les syndromes des vecteurs par *poids croissant*
2. Quand un nouveau syndrome apparaît, noter le vecteur correspondant
3. S'arrêter dès que le tableau est terminé (attention à NC_k)

Exemple
- Ligne 1 : Liste des 2^r syndromes
- Ligne 2 : Liste des vecteurs de correction correspondants
  - Vecteur nul : Syndrome nul
  - Vecteurs de poids 1 : Syndromes qui sont des colonnes de H
  - En cas d'ambiguïté : Non Corrigeable d'ordre k
  - Poids 2,3,... si nécessaire

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Syndrome	0	0	0	0		1	1	1	1
	0	0	1	1		0	0	1	1
	0	1	0	1		0	1	0	1
Vecteur de correction	0	0	0	0	1				
	0	0	0	0	0				
	0	0	0	0	ou 1				
	0	0	1	1	0				
	0	1	0	1	0				

Vocabulaire

⋆ Construction du tableau standard
⋆ Utilisation du tableau standard

6.6 Retour sur la distance minimale du code

Il s'agit d'une notion essentielle pour mesurer l'efficacité du code puisqu'elle permet de garantir la capacité à détecter ou corriger du code.

- Messages erronés avec strictement moins de d erreurs : TOUS DÉTECTÉS
- Messages erronés avec strictement moins de $d/2$ erreurs : TOUS BIEN CORRIGÉS

Notons au passage, que ceci n'interdit pas au code de bien détecter ou de bien corriger certains messages avec plus d'erreurs, seulement ce n'est pas possible en général.

Dans le cas des codes linéaires,

Propriété. La distance minimale d'un *code linéaire* est égale au poids d'un mot de code non nul de plus faible poids

6.6.1 Colonnes de la matrice de contrôle

- Les colonnes de H sont les syndromes des messages de poids 1

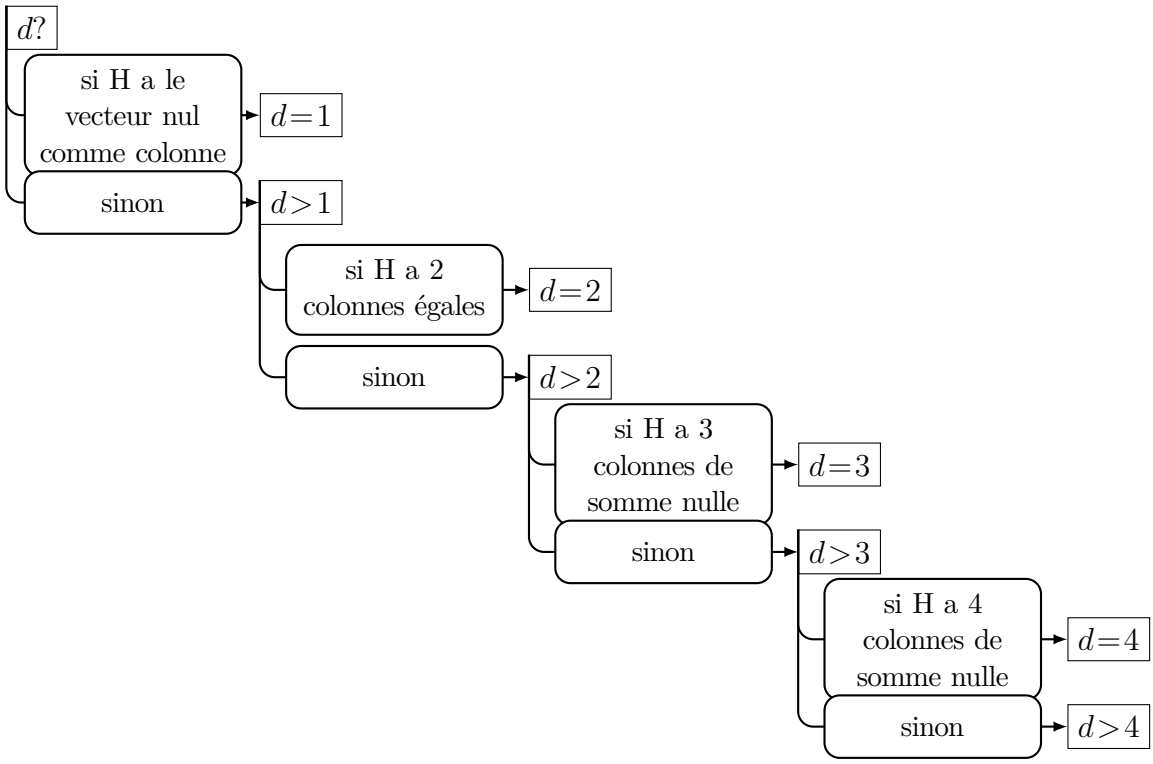
$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

- Plus précisément, la colonne C_i est le syndrome du message $m_i = 0...010...0$ avec un 1 en position i
- Les mots de code ont le syndrome nul $\begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$ et eux seuls

6.6.2 Calcul de la distance minimale du code

Propriété. La distance minimale du code d est le plus petit nombre de colonnes de la matrice de contrôle H qu'il faut additionner $\oplus$ pour obtenir la colonne nulle.

Mise en œuvre



Vocabulaire

⋆ Propriété de la distance minimale dans un code linéaire
⋆ Calcul de la distance minimale à partir de la matrice de contrôle