MITRO210 : Automates et donnés structurées

Notes de cours

Antoine Amarilli

Avertissement : ces notes de cours n'ont pas été beaucoup relues et contiennent certainement de nombreuses erreurs. Je vous invite à y prendre garde quand vous les utilisez, et vous encourage à me signaler tout problème ou toute question, par mail à l'adresse `a3nm@a3nm.net`*.*

1 Synopsis et contenus exigibles

- Toutes les notions récapitulées dans le synopsis d'INF105 [1] et couvertes dans le polycopié du cours INF105. *Attention, toutes ces notions n'ont pas été rappelées dans ce cours, et elles n'ont pas forcément été couvertes par votre groupe en INF105, mais je les suppose connues et maîtrisées.*
- Automates inambigus. Les automates inambigus peuvent être plus concis que les automates déterministes. Comptage efficace des mots d'une longueur donnée acceptés par un automate déterministe ou inambigu. NP-difficulté du comptage des mots d'une longueur donnée acceptés par un automate non-déterministe (sans démonstration). Test efficace de si un automate nondéterministe est inambigu (TD1).
- Automates alternants (définition). Les automates alternants se convertissent en automates nondéterministes (TD 2).
- Automates bidirectionnels (définition). Les automates bidirectionnels se convertissent en automates nondéterministes (TD 1 + cours 2).
- Automates à pile déterministes, inambigus, nondéterministes. Différences entre ces sortes d'automates et lien avec les classes de grammaires. Propriétés de clôture (sans démonstration).
- Automates à compteurs (non définis formellement) et relations avec les automates à pile.
- Machines de Turing (définition). Langages calculables. Grammaires sensibles au contexte (non définies formellement).
- Machines à plusieurs piles ou plusieurs compteurs : équivalence avec les machines de Turing (sans preuve).
- Recherche de facteurs. Algorithme de Knuth-Morris-Pratt (avec preuve complète). Construction du tableau des correspondances partielles (manuellement, et algorithmiquement) et utilisation. Algorithme d'Aho-Corasick (TD 3).
- Définition des tries, tries suffixes et tries suffixes compressés. Utilisation des tries suffixes notamment pour tester si un mot est facteur d'un autre. Construction manuelle d'un trie suffixe compressé pour un mot (le fait que ce calcul puisse être fait en temps linéaire est admis).

1. `https://perso.telecom-paristech.fr/madore/inf105/programme-inf105.pdf`

— Test efficace de si un facteur appartient à un langage, énumération efficace des facteurs (TD 4).
— Définition du monoïde de transition d'un automate et calcul de ce monoïde.
— Langages sans étoiles, automates sans compteurs, monoïdes apériodiques, et liens entre ces notions (admis).
— Utilisation du monoïde syntaxique pour la maintenance efficace de l'appartenance à un langage.
— Utilisation du monoïde syntaxique pour indexer efficacement un mot afin de déterminer si un facteur appartient à un langage (TD 5).
— Automates d'arbres de bas en haut sur les Σ-arbres : déterministes (DbTA), nondéterministes (NbTA), et inambigus (UbTA). Définition et construction pour des langages simples. Langages d'arbres reconnaissables. Les langages d'arbres finis sont reconnaissables (TD 6). Clôture sous opérateurs booléens (TD 6).
— Les automates d'arbres peuvent se déterminiser (admis) et admettent un résultat de pompage (seulement l'idée). Notion d'automates de haut en bas (seulement l'idée). Les automates de haut en bas déterministes sont moins expressifs que les automates de bas en haut (seulement l'idée).
— Liens entre automates d'arbres et automates réguliers : les arbres peignes correspondant à des langages de mots réguliers sont des langages d'arbres reconnaissables, et vice-versa.
— Problème de 3-coloriage avec contraintes. Résolution en temps linéaire sur les arbres avec un automate d'arbres (TD 6).
— Notion de décomposition arborescente et largeur arborescente : définition et calcul sur des exemples simples. Les graphes de largeur arborescente 1 sont précisément les forêts. Complexité du calcul de décompositions arborescentes (admis).
— Résolution du problème de 3-coloriage avec contraintes en temps linéaire sur des graphes de largeur arborescente constante.

2 Rappels de INF105 sur les automates finis

Les notions suivantes ont déjà été vues en INF105 et sont simplement rappelées brièvement ici ; il est recommandé de se reporter au polycopié d'INF105 [2] pour une présentation plus complète.

2.1 Alphabet, mot, facteur, langages

Un *alphabet* Σ est un ensemble fini de symboles appelés *lettres*. Un *mot* $w = a_1 \cdots a_n$ est une séquence finie de lettres ; sa *longueur* $|w|$ est n. Le *mot vide* est noté ϵ ; sa longueur $|\epsilon|$ est 0. On note Σ^* l'ensemble (infini dénombrable) de tous les mots.

On définit une opération de *concaténation* sur les mots : on note uv la concaténation de $u, v \in \Sigma^*$. On a toujours $|uv| = |u| + |v|$. On peut en particulier noter $u^2 = uu$ la concaténation de u avec lui-même. Plus généralement, on définit inductivement $u^0 = \epsilon$ (l'élément neutre de la concaténation), et pour $i \in \mathbb{N}$ on définit $u^{i+1} := uu^i$, en particulier $u^1 = u$.

Les mots sous l'opération de concaténation forment un *monoïde*, c'est-à-dire un ensemble muni d'une loi de composition associative dont l'élément neutre est ϵ. En revanche, ce monoïde n'est pas un groupe, car la concaténation n'a pas d'inverse. Ce monoïde n'est pas non plus commutatif (sauf dans le cas trivial où l'alphabet n'a qu'une seule lettre).

On dit que u est *préfixe* de v s'il existe $w \in \Sigma^*$ tel que $v = uw$. On dit que u est *suffixe* de v s'il existe $w \in \Sigma^*$ tel que $v = wu$. On dit que u est *facteur* de v s'il existe $s, t \in \Sigma^*$ tel que

2. `http://perso.enst.fr/madore/inf105/notes-inf105.pdf`

$v = sut$. Cette notion est à distinguer de celle de *sous-mot*, où u est sous-mot de v s'il apparaît comme une suite extraite (non nécessairement contiguë) de v.

Un *langage* L est un ensemble de mots, c'est-à-dire un sous-ensemble de Σ^*. Un langage peut être *fini*, mais en général un langage n'est pas nécessairement fini.

2.2 Langages réguliers, expressions régulières

On peut définir sur les langages les opérations booléennes habituelles : la *complémentation*, notée $\overline{L} := \Sigma^* \setminus L$; l'*union* $L_1 \cup L_2$ de deux langages L_1 et L_2, et l'*intersection* $L_1 \cap L_2$ de deux langages L_1 et L_2. On note qu'appliquées aux langages finis, les opérations d'union et d'intersection ne produisent que des langages finis ; si on ajoute l'opération de complémentation, alors on obtient les langages finis et les langages *cofinis* c'est-à-dire les langages dont le complémentaire est fini.

On définit également sur les langages l'opération de *concaténation* : étant donné deux langages L_1 et L_2, la *concaténation* de L_1 et L_2, notée L_1L_2, est le langage $\{u_1u_2 \mid u_1 \in L_1, u_2 \in L_2\}$ des mots qui peuvent s'obtenir comme la concaténation d'un mot de L_1 et d'un mot de L_2. Noter qu'on peut obtenir le même mot plusieurs fois. En particulier, pour un langage L, on note $L^2 = LL$ la concaténation de L avec lui-même. Attention, ce n'est pas le même langage que $\{u^2 \mid u \in L\}$. Plus généralement, on note $L^0 = \{\epsilon\}$ (l'élément neutre de la concaténation des langages), et pour $i \in \mathbb{N}$ on définit $L^{i+1} = LL^i$, en particulier $L^1 = L$.

On définit alors l'*étoile* comme l'opération suivante sur les langages : $L^* := \bigcup_{i\in\mathbb{N}} L^i$. En particulier, si $L = \emptyset$ alors $L^* = \{\epsilon\}$, de même si $L = \{\epsilon\}$ alors $L^* = \{\epsilon\}$, dans tous les autres cas L^* est un langage infini même si L est fini. Noter qu'on a toujours $\epsilon \in L^*$. En particulier, si on note Σ le langage formé de tous les mots d'une seule lettre de Σ, alors on a bien suivant cette définition que Σ^* est l'ensemble de tous les langages.

Les langages *réguliers* sont les langages que l'on peut obtenir à partir des langages finis avec les opérations d'union, de concaténation, de complémentation, et d'étoile. De manière équivalente, on peut définir les langages réguliers en n'autorisant que les opérations de concaténation, d'union, et d'étoile, à partir des cas de base suivants :

— Le langage vide $\emptyset$
— Le langage $\{\epsilon\}$ formé uniquement du mot vide
— Les langages singleton $\{a\}$ pour $a \in \Sigma$.

Cette définition plus restreinte donne la même classe de langages. Les *expressions régulières* sont une notation conventionnelle qu'on ne rappellera pas ici, qui permettent de décrire de façon concise la définition d'un langage à partir de ces cas de base et de ces opérations ; elles sont souvent étendues par du sucre syntaxique (qui abrège l'écriture sans permettre de définir davantage de langages), et parfois par d'autres extensions utilisées en pratique (qui augmentent leur pouvoir d'expressivité au-delà des langages réguliers ; c'est par exemple le cas des références arrière ou *backreferences*).

Les propriétés de clôture qui suivent peuvent être démontrées notamment via les automates finis (à savoir, par les constructions de l'automate produit, et par la complémentation des automates déterministes complets), en utilisant le théorème de Kleene que l'on rappellera plus bas :

Lemma 2.1. *Les langages réguliers sont clos par intersection, c'est-à-dire que l'intersection de deux langages réguliers est elle-même un langage régulier. De même, les langages réguliers sont clos par complémentation, c'est-à-dire que le complémentaire d'un langage régulier est lui-même un langage régulier.*

Les langages réguliers sont également clos sous l'opération dite de *miroir*. Le *miroir* d'un mot $w = a_1 \cdots a_n$ est le mot $w^{\mathrm{R}} = a_n \cdots a_1$. Le *miroir* d'un langage L est le langage $L^{\mathrm{R}} =$

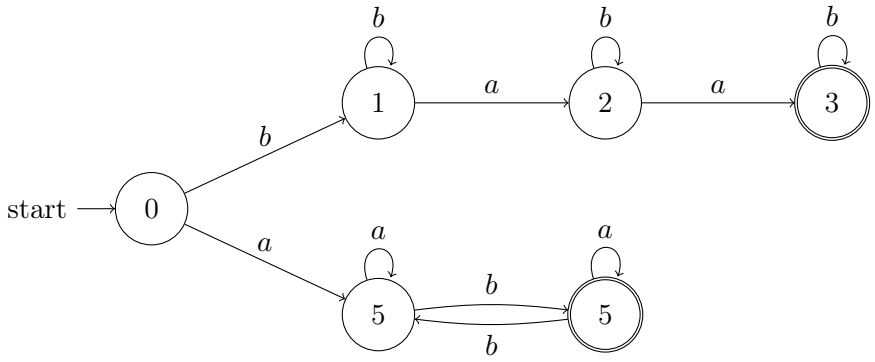


FIGURE 1 – Exemple de DFA, cf Exemple 2.3

$\{w^{\mathrm{R}} \mid w \in L\}$. La propriété de clôture correspondante peut se démontrer via les automates (à savoir, en inversant les transitions et en échangeant les états initiaux et finaux d'un automate nondéterministe) :

Lemma 2.2. *Les langages réguliers sont clos par miroir, c'est-à-dire que le miroir d'un langage régulier est lui-même un langage régulier.*

2.3 Automates finis déterministes et nondéterministes

Rappelons à présent la définition des automates finis déterministes. Un *automate fini déterministe* (à spécification incomplète), ou DFA, sur un alphabet Σ est un tuple $A = (Q, \Sigma, q_0, F, \delta)$ où Q est un ensemble fini d'*états*, Σ est l'alphabet, $q_0 \in Q$ est l'*état initial*, $F \subseteq Q$ est l'ensemble des *états finaux*, et $\delta \colon Q \times \Sigma \to Q$ est une fonction partielle appelée *fonction de transition*.

Exemple 2.3. *La Figure 1 montre un exemple de DFA reconnaissant le langage des mots qui soit commencent par a et contiennent un nombre impair de b, soit commencent par b et contiennent précisément deux a.*

Un automate est une façon opérationnelle de définir un langage. Formellement, pour un mot $w = a_1 \cdots a_n$ de Σ^*, un *run* de l'automate sur w est une séquence de la forme $q_0, \ldots, q_n$ tel que q_0 est l'état initial et, pour chaque $0 \leq i < n$, la fonction $\delta(q_i, a_{i+1})$ est définie et vaut q_{i+1}. Noter qu'un mot a au plus un run, mais qu'il peut ne pas y avoir de run dans les cas où la fonction de transition n'est pas totale. Le run est *acceptant* si son dernier état q_n est un état final. Le mot w est *accepté* par A si l'automate A a un run sur w et qu'il est acceptant ; sinon w est *rejeté*. Le *langage* de l'automate A est l'ensemble des mots acceptés par l'automate.

On peut vouloir imposer deux propriétés sur les automates qui sont en général incompatibles. D'un côté, on peut vouloir imposer que la fonction de transition soit une fonction totale ; auquel cas on dit que l'automate est *complet*, et tout mot admet alors exactement un run. La complétude peut-être imposée sur un automate arbitraire en ajoutant si nécessaire un état q supplémentaire, appelé *puits*, qui n'est ni initial ni final ; en ajoutant de transitions de q vers q vers toutes les lettres de l'alphabet, formellement en définissant $\delta(q, a) = q$ pour tout $a \in \Sigma$; et en complétant toutes les transitions manquantes pour aller vers le puits, formellement en définissant $\delta(q', a) := q$ pour toutes les paires $(q', a) \in Q \times \Sigma$ pour lesquelles la fonction δ n'était pas définie. On constate sans peine que l'automate ainsi modifié est complet et reconnaît le même langage que l'automate original.

D'un autre côté, on peut vouloir imposer que tous les états de l'automate soient *utiles*, c'est-à-dire qu'ils fassent partie d'un run acceptant. Formellement, un état de l'automate A est dit

accessible s'il existe un chemin de q_0 à lui dans le graphe orienté défini par les transitions de l'automate, formellement dans le graphe orienté ayant Q comme ensemble de sommets et ayant une arête (q, q') à chaque fois qu'il existe $a \in \Sigma$ tel que $\delta(q, a) = q'$. Un état de l'automate est dit *co-accessible* s'il existe un chemin de lui à un état final dans ce même graphe orienté. Un état est *utile* s'il est à la fois accessible et co-accessible ; on constate sans peine qu'un état est utile si et seulement s'il y a un mot accepté par l'automate dans le run acceptant duquel il apparaît. Un automate est dit *émondé* (en anglais *trimmed*) si tous ses états sont utiles. On peut émonder n'importe quel automate en supprimant tous ses états inutiles et les transitions qui mènent vers les états inutiles ; il est clair que le résultat est effectivement un automate émondé qui reconnaît le même langage que le langage original. En revanche ce n'est généralement pas un automate complet (et si on ajoute un puits alors le puits n'est pas coaccessible) [3].

Un modèle généralisant les automates déterministes est celui des automates *nondéterministes*. Un *automate fini nondéterministe* (NFA) est un tuple $(Q, \Sigma, I, F, \delta)$ où Q est un ensemble fini d'*états*, Σ est l'alphabet, $I \subseteq Q$ est l'ensemble des *états initiaux*, $F \subseteq Q$ est l'ensemble des *états finaux*, et $\delta \subseteq Q \times \Sigma \times Q$ est une *relation de transition*. La différence avec un DFA est que, dans un NFA, pour un état q et une lettre a donnée, il peut y avoir plusieurs transitions différentes à partir de q étiquetées par a, c'est-à-dire plusieurs q' tels que $(q, a, q') \in \delta$; en revanche dans un automate déterministe il y a au plus un état q' tel que $q' = \delta(q, a)$.

Un NFA définit un langage d'une manière analogue à celle des DFA. Formellement, pour un mot $w = a_1 \cdots a_n$ de Σ^*, un *run* du NFA A est une séquence de la forme $q_0, \ldots, q_n$ tel que q_0 est l'état initial et, pour chaque $0 \leq i < n$, on a $(q_i, a_{i+1}, q_{i+1}) \in \delta$. Noter qu'il n'y a plus unicité des runs. Comme précédemment, le run est *acceptant* si son dernier état q_n est un état final. Le mot w est *accepté* par A s'il existe un run acceptant de A sur w ; sinon il est *rejeté*. Le *langage* du NFA A est l'ensemble des mots acceptés par A.

2.4 Algorithme de déterminisation

Un résultat du cours de INF105 est que les NFA et les DFA ont le même pouvoir d'expressivité : étant donné un NFA, on peut construire un DFA reconnaissant le même langage, mais la construction est généralement exponentielle :

Proposition 2.4. *Étant donné un NFA A avec n états, on peut construire un DFA A' avec n états tels que A et A' reconnaissent le même langage.*

On rappelle la construction ici :

Démonstration. Notons $A = (Q, \Sigma, I, F, \delta)$. On définit l'ensemble des états de A' comme 2^Q, l'ensemble des parties de Q. Ceci satisfait bien la borne demandée. On choisit l'ensemble I comme état initial de Q' (c'est un ensemble d'états de Q, donc un état de Q'). On choisit comme état final de Q' l'ensemble des ensembles d'états de Q qui contiennent un état final, formellement $\{X \in 2^Q \mid X \cap F \neq \emptyset\}$. Quant à la fonction de transition $\delta' \colon 2^Q \times \Sigma \to 2^Q$ de A', qu'on choisira totale, elle est définie comme suit : pour $X \in 2^Q$ et $a \in \Sigma$, on pose :

$$\delta'(X, a) := \bigcup_{q_1 \in X} \{q_2 \in Q \mid (q_1, a, q_2) \in \delta\}$$

En d'autres termes, à partir de l'ensemble d'états X, quand on lit la lettre a, on considère chacun des états q_1 de X, et on regarde les états q_2 qu'on peut atteindre à partir de q_1 en lisant

3. On peut remarquer qu'un langage régulier L peut être reconnu par un automate à la fois complet et émondé si et seulement si le langage a la propriété d'être *continuable* au sens où pour tout mot $u \in \Sigma^*$, il existe $v \in \Sigma^*$ tel que $uv \in L$.

la lettre a ; l'ensemble de tous ces états q_2 est l'ensemble que l'on choisit pour $\delta'(X, a)$. À noter que cet ensemble n'est pas nécessairement de cardinalité supérieure à X : il peut même être vide, dans le cas où aucun des états q_1 de X n'a de transition sortante étiquetée par a.

On se convainc que A et A' reconnaissent le même langage en démontrant l'assertion suivante, par récurrence sur la longueur n du mot : pour tout mot w de longueur n, si on note X l'état atteint dans A' en lisant le mot w (formellement le dernier état de l'unique run de A' sur w), alors X est précisément l'ensemble des états q tels que A a un run sur w finissant en q. En effet, pour $n = 0$ et $w = \epsilon$, l'état atteint dans A' est l'état initial I, et de fait les runs de A sur le mot vide sont tous des séquences de longueur 1 dont le dernier état est un état initial. Si l'on suppose l'assertion vraie pour n et que l'on considère un mot $w = ua$ avec $a \in \Sigma$ et $|u| = n$, alors on sait en appliquant l'hypothèse d'induction sur u que la lecture de u dans A' nous conduit à un état X qui est précisément l'ensemble des états de A qui finissent un run de A sur u. Ensuite, quand on lit a dans A', on se retrouve à l'ensemble défini par l'équation plus haut ; or un run de A sur $w = ua$ se compose précisément d'un run de A sur u se finissant en un certain état q_1, puis d'un état q_2 qui termine le run et satisfait $(q_1, a, q_2) \in \delta$. Ainsi l'ensemble des états q_2 qui peuvent terminer un run de A sur w sont précisément les éléments de l'ensemble auquel on parvient en lisant w dans A'. Ceci conclut l'étape d'induction et termine la démonstration. □

Un formalisme d'automates que nous ne redéfinirons pas formellement mais que nous supposons connu suivant le cours INF105 est celui des automates à *transitions spontanées* ou *ϵ-transitions*. Il s'agit d'une extension des NFA où on autorise également des transitions étiquetées par le symbole ϵ, qui peuvent être franchies spontanément par l'automate sans lire de lettres dans le mot d'entrée. On suppose connue la construction du cours INF105 qui permet de démontrer qu'un automate nondéterministe avec transitions spontanées peut être récrit en un NFA (sans transitions spontanées) qui reconnaît le même langage. Ceci implique que les automates avec transitions spontanées ont le même pouvoir d'expressivité que les NFA, et donc que les DFA ; mais ils peuvent être plus concis que ces derniers.

2.5 Théorème de Kleene

Un résultat du cours INF105 que nous ne redémontrerons pas ici est que les langages réguliers (définis par les opérations d'union, concaténation et étoile à partir des langages finis) sont précisément les langages qui peuvent être reconnus par un automate fini (déterministes ou nondéterministes, cela ne change rien comme nous venons de le montrer).

Theorem 2.5 (Théorème de Kleene)**.** *Un langage est reconnaissable par un automate fini si et seulement s'il est régulier.*

En résumé, la preuve de ce théorème implique deux directions. La première est de montrer qu'un langage régulier peut être reconnu par un automate fini. Ceci se démontre en construisant explicitement un tel automate à partir d'une expression régulière pour le langage reconnu, par exemple avec la construction de Glushkov, ou avec celle de Thompson (qui est moins efficace, et introduit des ϵ-transitions, mais est conceptuellement plus simple). La seconde est de montrer que le langage reconnu par un automate peut s'exprimer avec les opérations régulières, ce qui peut se faire notamment par la méthode d'élimination des états.

2.6 Lemme de pompage

Un autre résultat du cours INF105 que nous supposerons connu est le *lemme de pompage* (ou lemme de l'étoile) : il s'agit d'une condition suffisante sur les langages réguliers, qu'on utilise souvent par contraposition pour montrer qu'un langage n'est pas régulier parce qu'il ne satisfait pas la condition. Son énoncé est le suivant :

Lemma 2.6 (Lemme de pompage)**.** *Pour tout langage régulier L, il existe un entier $k \in \mathbb{N}$ tel que tout mot w de L satisfaisant $|w| \geq k$ admet la propriété suivante : il existe $x, y, z \in \Sigma^*$ tels qu'on peut écrire $w = xyz$, et on a les propriétés suivantes :*

1. *Le mot y n'est pas vide, formellement $|y| \geq 1$.*
2. *On a $xy^iz \in L$ pour tout $i \in \mathbb{N}$.*
3. *On a $|xy| \leq k$.*

La démonstration de ce lemme est facile est passe par les automates : intuitivement k est le nombre d'états d'un DFA reconnaissant A, et y est défini en identifiant une boucle parcourue dans l'automate en lisant les k premières lettres de w (et en utilisant le principe des tiroirs).

Il est important de noter que les trois conditions du lemme ne remplissent pas les mêmes rôles : la condition qui nous intéresse vraiment est le point 2, et le point 1 est nécessaire pour que le point 2 soit intéressant. Le point 3, en revanche, peut permettre de simplifier certaines preuves mais n'est pas toujours nécessaire (le résultat aurait également un intérêt sans ce troisième point). Pour l'utilisation du point 2, on va généralement vouloir conclure à une contradiction en disant qu'un des itérés obtenus n'appartient pas aux langages : selon les cas, il pourra être plus simple de prendre $i = 0$ (c'est-à-dire retirer y), ou $i = 2$ (répéter y), ou bien prendre une plus grande valeur de i correspondant à répéter y un plus grand nombre de fois.

Exemple 2.7. *Le lemme de pompage permet facilement de montrer que le langage $\{a^nb^n \mid n \in \mathbb{N}\}$ n'est pas régulier ; cette démonstration est également supposée connue.*

3 Automates inambigus et nombres de mots acceptés

Après ces quelques rappels, nous présentons la notion d'un automate nondéterministe *inambigu* (ou UFA). Nous illustrons que les UFA peuvent être plus concis que les DFA, et cependant qu'ils peuvent mieux se comporter que les NFA : nous illustrons ceci en remarquant qu'on peut facilement *compter* les mots d'une certaine longueur acceptés par un UFA ou DFA, mais pas par un NFA.

3.1 Automates inambigus

Nous commençons par définir les automates inambigus :

Definition 3.1. *Un* automate inambigu *(ou UFA) est un NFA A satisfaisant la condition suivante : pour tout mot w, il y a au plus un run acceptant de A sur w.*

Noter que cette condition, contrairement au déterminisme des DFA, n'est pas une condition *syntaxique* (observable directement sur l'automate), mais *sémantique*. Nous verrons en TD comment vérifier, étant donné un NFA, s'il est inambigu ou non.

On remarque qu'un DFA (vu comme un NFA ayant au plus une cible par état et par lettre) est toujours un UFA. En effet, un DFA a toujours au plus un run sur un mot donné, en particulier un run acceptant. En revanche, un UFA peut avoir plusieurs runs, du moment qu'un seul est acceptant.

On remarque aussi que l'automate miroir d'un UFA (reconnaissant le langage miroir) est également un UFA ; la condition est symétrique en ce sens. En revanche le miroir d'un DFA n'est pas nécessairement un DFA ; mais c'est forcément un UFA, vu que son miroir est un DFA donc un UFA.

Quel est l'intérêt des UFA comparé aux DFA ? On peut constater que, pour certains langages, ils peuvent être plus concis :

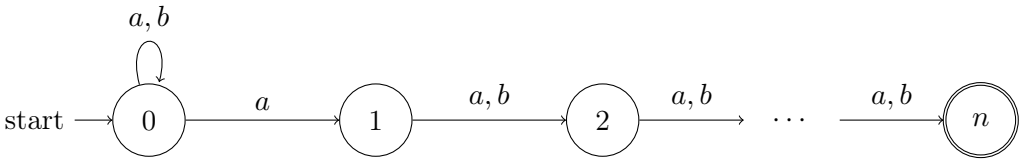


FIGURE 2 – Exemple d'un UFA pour L'_n (voir Exemple 3.2)

Exemple 3.2. *Fixons l'alphabet $\Sigma = \{a, b\}$. Pour $n \in \mathbb{N}$, on définit le langage L_n des mots dont la n-ième lettre en partant du début est un a, et L'_n le langage des mots dont la n-ième lettre en partant de la fin est un a.*

Le langage L_n peut facilement être reconnu par un DFA à $n+1$ états. En revanche, la situation est différente pour L'_n. On peut concevoir un NFA A_n à $n+1$ états pour le reconnaître, voir Figure 2. On remarque que ce NFA est en réalité un UFA : en effet, étant donné un mot w, l'unique run acceptant de A_n sur w, s'il existe, est celui qui suit la transition de 0 à 1 au moment de lire la n-ième lettre de w en partant de la fin (et ce doit être un a).

On peut démontrer qu'un DFA complet pour L'_n doit avoir au minimum 2^n états. En effet, posons un DFA complet A'_n reconnaissant L'_n, et considérons les 2^n mots de longueur n sur l'alphabet Σ, que l'on note $w_1, \ldots, w_{2^n}$. Comme A'_n est complet, on peut noter les états $q_1, \ldots, q_{2^n}$ que l'on atteint en lisant chacun de ces mots respectivement : comme A'_n est déterministe, ces états sont définis de manière unique. Procédons par l'absurde et supposons que A'_n a moins de 2^n états. Dans ce cas, par le principe des tiroirs, il y a deux mots w_i, w_j avec $i \neq j$ tels que $q_i = q_j$. Comme w_i et w_j sont différents, il y a une position où ils diffèrent ; quitte à échanger w_i et w_j on peut supposer que w_i contient un a à la p-ième position à partir de la droite et w_j contient un b. Mais alors, prenons w un mot quelconque de longueur $n - p + 1$, et considérons $w'_i = w_i w$ d'une part, et $w'_j = w_j w$ d'autre part. Le mot w'_i contient un a à la n-ième position à partir de la droite, et w'_j non, ainsi $w'_i \in L$ et $w'_j \in L$. Pourtant, les runs de A'_n sur w'_i et w'_j s'obtiennent en poursuivant le run de A'_n sur w_i et w_j respectivement, où on aboutit à $q_i = q_j$; et à lire la continuation commune w. Ainsi, les runs de A'_n sur w'_i et w'_j aboutissent-ils au même état. Mais cet état est à la fois final (puisque $w'_i \in L$) et non-final (puisque $w'_j \in L$), ce qui est impossible ; contradiction.

Cet exemple illustre ainsi que, pour certains langages, on peut obtenir un UFA concis même quand on ne peut pas obtenir de NFA concis.

3.2 Nombres de mots acceptés par un automate

Nous avons illustré l'intérêt des UFA relativement aux DFA : ils sont parfois plus concis. Toutefois, les UFA ont-ils un intérêt relativement aux NFA ? Pour illustrer que c'est le cas, nous considérons le problème de compter le nombre de mots d'une certaine longueur acceptés par un automate.

Definition 3.3. *Le problème de* comptage des mots acceptés *est le suivant : étant donné un entier $n \in \mathbb{N}$ et un automate A, compter combien de mots distincts de Σ^* de longueur n sont acceptés par l'automate A.*

On peut toujours résoudre ce problème en temps $O(2^n \times n \times |A|)$ sur un NFA A, où $|A|$ est le nombre de transitions du NFA : il suffit de considérer les 2^n mots candidats et de tester pour chacun d'entre eux si A l'accepte, en temps $O(n \times |A|)$. Cependant, on peut faire mieux dans le cas des DFA :

Proposition 3.4. *Étant donné un DFA A à m états sur un alphabet Σ de taille $|\Sigma| = k$, on peut déterminer en temps $O(mnk)$ le nombre de mots de longueur n acceptés par A.*

Démonstration. On procède par programmation dynamique. Supposons le DFA complet par souci de simplicité. On définit un tableau T, indexé par les états de l'automate et les entiers de 0 à n inclus, tel que $T[q, i]$ indique le nombre de mots de longueur i acceptés à partir de l'état q. On peut remplir ce tableau inductivement : pour $i = 0$ on définit $T[q, 0]$ comme valant 0 si q n'est pas final et 1 si q est final (correspondant à l'acceptation de ϵ depuis q) ; puis pour chaque $i > 0$ on définit :

$$T[q, i] := \sum_{a \in \Sigma} T[\delta(q, a), i - 1]$$

En effet il est clair que l'ensemble des mots de longueur i acceptés depuis q par A peut se partitionner suivant la première lettre du mot, et on utilise l'hypothèse d'induction pour justifier que les comptes précédents sont corrects. À la fin, le résultat que l'on souhaite renvoyer est $T[q_0, n]$, pour q_0 l'état initial. □

Qu'en est-il du cas des UFA ? On peut en réalité adapter le résultat précédent avec un algorithme très similaire :

Proposition 3.5. *Étant donné un UFA A à m états, on peut déterminer en temps $O(mn \times |A|)$ le nombre de mots de longueur n acceptés par A, où $|A|$ est le nombre de transitions de A.*

Démonstration. On suppose sans perte de généralité que A est émondé. On procède comme pour la Proposition 3.5, mais on remplit le tableau comme suit dans le cas inductif :

$$T[q, i] := \sum_{(q,a,q') \in \delta} T[q', i - 1]$$

Ceci garantit la borne sur le temps d'exécution, mais il faut se persuader que les comptes ainsi obtenus sont corrects. Le cas de base ne pose pas de problème. Pour le cas d'induction, c'est toujours vrai que les mots de longueur i acceptés depuis q se partitionnent suivant leur première lettre : en revanche, pour une lettre a donnée, il faut justifier que les mots de longueur i acceptés depuis q et commençant par a sont partitionnés par le choix du prochain état q' dans le run. La seule chose à justifier est que le même mot av de longueur i ne peut pas être acceptée depuis q en allant vers deux états différents q'_1 et q'_2 avec $(q, a, q'_1) \in \delta$ et $(q, a, q'_2) \in \delta$. Mais si c'était le cas, alors ceci contredirait le caractère inambigu de A. En effet, comme A est émondé, q est accessible : posons u un mot permettant d'atteindre q depuis un état initial. Le mot uav aurait alors (au moins) deux runs distincts : en rejoignant q en lisant u, on peut passer soit par q'_1, soit par q'_2. Ainsi, la somme dans la définition ci-dessus est correcte, ce qui conclut la preuve. □

Une autre façon de comprendre ce résultat est de se rendre compte que l'algorithme de ces deux propositions revient en réalité à compter les runs acceptants, qui pour les DFA et les UFA sont en bijection avec les mots du langages.

Peut-on avoir un algorithme efficace pour compter les mots acceptés par un NFA ? Le problème des NFA est que le même mot peut être accepté par plusieurs runs, ainsi compter les runs (chemins dans l'automate) n'est pas la même chose que compter les mots acceptés (et les algorithmes précédents surestimeraient donc le nombre de mots acceptés). Mais on peut en réalité montrer que le problème est computationnellement difficile. En effet :

Proposition 3.6 (admis)**.** *Le problème suivant est NP-difficile : étant donné un NFA A et une longueur n, déterminer s'il y a un mot de Σ^* de longueur n qui* n'est pas *accepté par A.*

Ce résultat implique que, sous l'hypothèse que P $\neq$ NP, il n'y a pas d'espoir de compter efficacement les mots acceptés par un NFA, puisque c'est plus difficile que de savoir si ce compte est égal à 2^n.

On a donc montré que les UFA conservent certaines bonnes propriétés des DFA, notamment pour ce problème de comptage du nombre de mots acceptés : on peut le résoudre en temps polynomial pour les DFA et les UFA, mais il est intractable pour les NFA.

Comptage approché. Remarquons pour conclure que l'intractabilité du problème pour les NFA peut également s'exprimer dans le contexte des *problèmes de comptage*, où on cherche à calculer une valeur numérique (le nombre de mots acceptés) et non pas juste à résoudre un problème de décision. Plus spécifiquement, le problème de compter le nombre de mots de longueur n acceptés par un NFA A, étant donné A et n, est #P-difficile : ceci se montre par une variante de la Proposition 3.6 en réduisant depuis le problème de compter le nombre de valuations satisfaisantes d'une formule en DNF, qui est également #P-difficile. En revanche, on peut efficacement calculer des *approximations* du nombre de mots d'une certaine longueur acceptés par un NFA, comme le démontre un article récent [2].

4 Automates plus concis pour des langages réguliers

Dans cette section, nous présentons d'autres généralisations du modèle des automates finis qui n'étendent pas leur pouvoir d'expressivité, c'est-à-dire qu'elles permettent de définir les langages réguliers ; mais l'automate résultant peut être plus concis.

4.1 Automates alternants

Les *automates alternants* (AFA) sont une généralisation des automates nondéterministes qui visent intuitivement à rendre la définition plus symétrique : là où un automate nondéterministe, confronté à un choix, va accepter si *l'un des choix* mène à l'acceptation, les automates alternants permettent de façon symétrique de spécifier que, depuis un état, on n'acceptera que si *tous les choix* mènent à l'acceptation.

Definition 4.1. *Un* automate alternant *(AFA) est défini comme* $A = (Q_\exists \sqcup Q_\forall, \Sigma, I, F, \delta)$ *où l'ensemble d'états* Q *est scindé entre les états de* $Q_\exists$ *(dits* existentiels*), et les états de* $Q_\forall$ *(dits* universels*). Le* langage reconnu par A *est le langage des mots acceptés par* A *à partir d'un état* q_0 *de* I*, où le* langage de A accepté à partir d'un état q *est défini inductivement :*

- *Le mot vide* ϵ *est accepté à partir de* q *si* q *est final ;*
- *Pour un mot non-vide de la forme* av *avec* $a \in \Sigma$ *et* $v \in \Sigma^*$*, pour un état* q*, le mot* av *est accepté à partir de* q *dans les cas suivants :*
  - *Si* $q \in Q_\exists$*, alors* av *est accepté à partir de* q *s'il existe un état* q' *avec* $(q, a, q') \in \delta$ *et tel que* v *soit accepté à partir de* q'*.*
  - *Si* $q \in Q_\forall$*, alors* av *est accepté à partir de* q *si, pour tout état* q' *avec* $(q, a, q') \in \delta$*, on a que* v *est accepté à partir de* q'*.*

Exemple 4.2. *Un exemple d'automate alternant est donné en Figure 3. Cet automate vérifie que le nombre de* a *dans le mot d'entrée est multiple de 3, et que le nombre de* b *est multiple de 4. On note que cet automate est a priori plus concis que le nombre d'états d'un automate nondéterministe pour vérifier la même condition.*

On verra en séance de TD que les automates alternants ne permettent pas de définir davantage de langages que les langages réguliers : on peut les transformer en des NFA reconnaissant le même langage. (En revanche, cette transformation est exponentielle en l'automate d'entrée.) Intuitivement, il n'est pas très surprenant que ces automates ne permettent pas de reconnaître d'autres langages que les langages réguliers : l'ensemble des états possibles au cours de la lecture

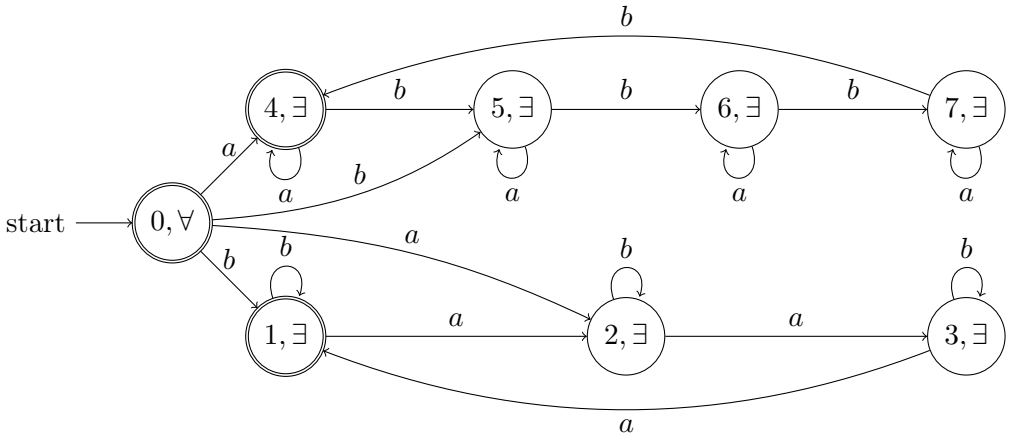


FIGURE 3 – Exemple d'automate alternant (cf Exemple 4.2). On annote chaque état avec $\forall$ et $\exists$ en fonction de s'il est universel ou existentiel.

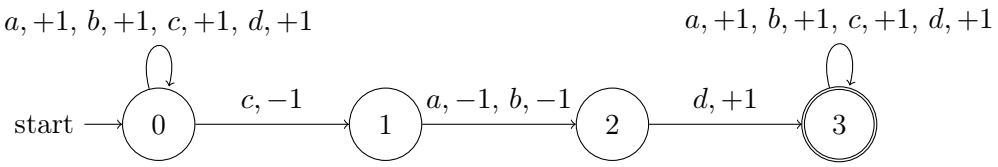


FIGURE 4 – Exemple de NFA bidirectionnel (cf Exemple 4.4). Par souci de lisibilité, on note -1 et $+1$ les déplacements dans les transitions.

du mot peut être résumé par une information finie qui dépend seulement de l'automate et non pas du contenu du mot (même si elle est plus complexe que pour les NFA).

4.2 Automates bidirectionnels

Les *automates bidirectionnels* sont une autre généralisation des automates finis qui autorise intuitivement l'automate à se déplacer dans les deux sens sur le mot d'entrée.

Definition 4.3. *Un* NFA bidirectionnel *(2NFA) est défini comme un quintuplet* $A = (Q, \Sigma, I, F, \delta)$ *avec* $\delta \subseteq Q \times \Sigma \times \{-1, 0, 1\} \times Q$. *Une* configuration *de* A *est une paire* (q, i) *avec* $i \in \mathbb{N}$. *Un* run *de* A *sur un mot* $w = a_1 \cdots a_n$ *de* Σ^* *est une séquence de configurations* $(q_0, i_0), \ldots, (q_m, i_m)$ *telle qu'on ait :*

— $q_0 \in I$ *et* $i_0 = 1$,
— $1 \leq i_j \leq n$ *pour chaque* $0 \leq j < m$
— *pour chaque* $0 \leq j < m$, *en écrivant* $d_j = i_{j+1} - i_j$, *on a* $(q_j, a_{i_j}, d_j, q_{j+1}) \in \delta$.

Le run est acceptant *si* $q_m \in F$ *et* $i_m = n + 1$.

Exemple 4.4. *Un exemple de 2NFA est donné en Figure 4. Cet automate fonctionne sur l'alphabet* $\Sigma = \{a, b, c, d\}$. *Il vérifie que le mot d'entrée contient un* c *tel qu'il y ait un* d *deux caractères avant et que le caractère entre les deux soit* a *ou* b.

On verra en séance de TD que les automates bidirectionnels, eux aussi, ne permettent pas de définir davantage de langages que les langages réguliers : on peut les transformer en des NFA reconnaissant le même langage.

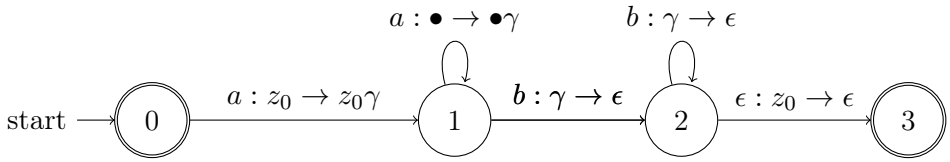


FIGURE 5 – Automate à pile déterministe qui reconnaît le langage $\{a^n b^n \mid n \in \mathbb{N}\}$. Les transitions portent des étiquettes de la forme $x : z \to \lambda$ pour indiquer qu'elles lisent la lettre x en trouvant le symbole z en haut de la pile et en le remplaçant par λ. On abrévie z par $\bullet$ quand une telle transition existe pour chaque z de l'alphabet de pile. Ici, l'alphabet de pile est $\{z_0, \gamma\}$.

Déterminisation des 2DFA. On note qu'on peut définir la notion d'un *automate bidirectionnel déterministe*, ou 2DFA, en imposant que pour chaque lettre il y ait une unique transition qui puisse être empruntée. Ainsi, pour tout mot, il y a un unique run (possiblement infini) du 2DFA sur le mot. On sait qu'on peut déterminiser tout 2NFA c'est-à-dire le transformer en un 2DFA reconnaissant le même langage : au minimum on peut le faire en transformant le 2DFA en un NFA, que l'on déterminise ensuite en un DFA, qui est en particulier un 2DFA. Cependant, on ne sait pas si un processus de déterminisation plus efficace existe, ou s'il faut au moins une complexité exponentielle pour déterminiser les 2NFA, de la même façon que les NFA. Ce problème est ouvert encore aujourd'hui ; il est lié à des questions ouvertes difficiles sur les classes de complexité, cf [10].

5 Automates pour des langages plus généraux

Dans cette section, on esquisse d'autres modèles d'automates, plus généraux que les automates finis, qui permettent cette fois de définir des langages plus généraux que les langages réguliers.

Automates à pile. Une classe particulièrement bien étudiée est celle des *automates à pile* ou en anglais *pushdown automaton* (PDA), dont on rappelle la définition (elle est également donnée dans le polycopié de INF105, mais on en dévie un peu) :

Definition 5.1. *Un* automate à pile nondéterministe *(NPDA) est un tuple* $(Q, \Sigma, \Gamma, z_0, I, F, \delta)$ *où* Q *est un ensemble fini d'états,* Σ *est l'alphabet,* Γ *est un ensemble fini appelé* alphabet de pile*,* $z_0 \in \Gamma$ *est un* symbole initial de pile*,* I *est un sous-ensemble de* Q *d'états dits* initiaux*,* F *est un sous-ensemble de* Q *d'états dits* finaux*, et* $\delta \subseteq Q \times (\Sigma \sqcup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$ *une* relation de transition*. Un tuple* (q, x, z, q', λ) *dans* δ *signifie que, si l'automate est à l'état* q *et que le symbole au sommet de la pile est* z*, alors l'automate peut transitionner en lisant la lettre* x *(et donc transitionner spontanément sans lire de lettre si* $x = \epsilon$*) pour arriver dans l'état* q' *et remplacer* z *sur le dessus de la pile par* λ *– donc retirer* z *si* $\lambda = \epsilon$*, ou le remplacer par un mot potentiellement plus long. L'automate accepte, informellement, s'il existe une succession de configurations partant d'un état initial avec la pile ne contenant qu'une copie de* z_0*, et parvenant à un état final en ayant consommé toutes les lettres du mot et avec un état quelconque de la pile.*

Un automate à pile déterministe *(DPDA) impose de plus qu'il y ait un unique état initial et que, informellement, en chaque état il y ait au plus une transition qui s'applique. Autrement dit, pour chaque état* q *et chaque symbole de sommet de pile* z *:*

- *Soit il y a une unique* ϵ*-transition depuis* q *pour le sommet* z *et aucune autre ;*
- *Soit il n'y a pas d'*ϵ*-transition, et pour chaque lettre il y a au plus une transition qui corresponde.*

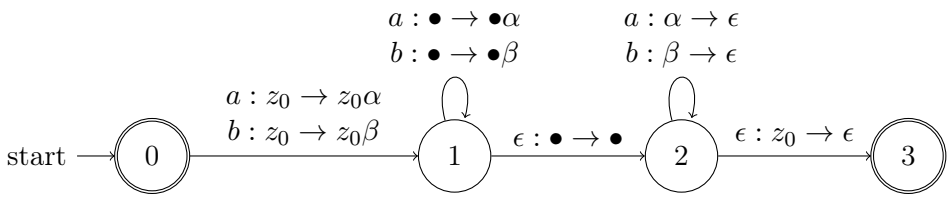


FIGURE 6 – Automate à pile qui reconnaît le langage des palindromes de longueur paire sur l'alphabet $\{a, b\}$. L'automate n'est pas déterministe à cause de la transition ϵ de 1 à 2 : intuitivement, l'automate doit deviner où se trouve le milieu du mot. Ici, l'alphabet de pile est $\{z_0, \alpha, \beta\}$.

Exemple 5.2. *La Figure 5 montre un exemple d'automate à pile déterministe. La Figure 6 montre un exemple d'automate à pile nondéterministe.*

On admettra que les automates à pile permettent de reconnaître les mêmes langages que les grammaires hors-contexte étudiées en INF105 (en particulier ils permettent de reconnaître les langages réguliers, mais également certains langages non-réguliers comme nous venons de l'illustrer, à savoir les langages *algébriques* ou *hors-contexte*). Du coup, le lemme de pompage pour les langages hors-contexte vu en INF105 s'applique aussi aux langages reconnus par des automates à pile.

Les langages reconnus par les automates à pile sont les langages hors-contexte, dont on rappelle que l'ensemble est clos par union, concaténation, et étoile. On rappelle en revanche qu'ils ne sont pas clos par intersection : prendre $\{a^n b^n c^m \mid n, m \in \mathbb{N}\}$ et $\{a^n b^m c^m \mid n, m \in \mathbb{N}\}$ dont l'intersection $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ n'est pas reconnaissable par un automate à pile (ceci suit directement du lemme de pompage, cf également Proposition 4.6.2 du polycopié de INF105). Ils ne sont donc pas clos par complément par application de la loi de de Morgan. On sait en revanche du cours INF105 (polycopié Proposition 4.2.7) que l'intersection d'un langage hors-contexte et d'un langage régulier est toujours hors-contexte : les automates à pile permettent d'ailleurs de démontrer aisément ce fait, simplement en faisant la construction de l'automate produit entre un automate à pile reconnaissant le langage hors-contexte et un automate fini reconnaissant le langage régulier.

Les automates à pile déterministes reconnaissent pour leur part un sous-ensemble des langages hors-contexte, appelés *langages hors-contexte déterministes*. Il est à noter qu'en particulier, pour ces langages, étant donné un mot, on peut tester en temps linéaire si le mot est accepté ou non ; ces langages sont en particulier précisément ceux qui peuvent être analysés par un analyseur LR.

L'ensemble des langages hors-contexte reconnaissables par des DPDA est clos par complémentation. La démonstration de ce fait est un peu désagréable et donnée dans [1] : il faut d'abord compléter l'automate en un certain sens pour garantir qu'il parvient toujours à la fin du mot lors de la lecture et qu'il ne boucle jamais sur des transitions qui ne lisent pas de symbole d'entrée (c'est-à-dire que la pile croît indéfiniment ou passe deux fois par le même état sans qu'on lise de symbole d'entrée) : ceci nécessite d'ajouter un nombre exponentiel d'états qui identifient quand cela arrive et déplacent l'automate vers un puits. Il faut ensuite échanger les états finaux et non-finaux mais, à cause de la présence de transitions ϵ, il faut à nouveau modifier l'automate pour garder trace de si on a visité un état final ou non à une position donnée du mot d'entrée.

En revanche, l'ensemble des langages hors-contexte reconnaissables par des DPDA n'est pas clos par intersection (cf exemple ci-dessus), donc pas par union ; et pas non plus d'ailleurs par concaténation. Tout ceci implique que les langages reconnus par un automate à pile déterministe forment un sous-ensemble strict des langages reconnus par des automates à pile ; autrement dit, pour certains automates à pile nondéterministes, il n'y a pas d'automates à pile déterministes

qui leur soient équivalents. On remarque d'ailleurs que les langages hors-contexte reconnus par un automate à pile déterministe ne peuvent pas être inhéremment ambigus. En revanche il existe des langages qui ne sont pas inhéremment ambigus mais qui ne sont pas reconnaissables par des automates à pile déterministe : par exemple le langage des palindromes de longueur paire. En résumé :

- Les automates à pile reconnaissent les langages hors-contexte, qui sont clos par union, concaténation, étoile, mais pas complémentation ni intersection
- Les automates à pile non-ambigus reconnaissent les langages hors-contexte qui ne sont pas inhéremment ambigus [8] ; c'est une sous-classe stricte des langages hors-contexte qui n'est pas close par complémentation [7] ni par union (prendre L_2 ci-dessous), donc pas par intersection (de Morgan), ni par concaténation (cf [11])
- Les automates à pile déterministes reconnaissent un sous-ensemble strict des langages hors-contexte qui ne sont pas inhéremment ambigus : cette clase est close par complémentation, mais pas par union ni par intersection ni par concaténation. Les grammaires en question s'appellent des *grammaires hors-contexte déterministes* mais elles n'ont pas de caractérisation simple connue sans passer par les DPDA.

Automates à un compteur. Un cas particulier d'automates à pile est les *automates à un compteur* : c'est le cas particulier de PDA dont la pile ne peut intuitivement stocker qu'un compteur. Formellement, l'alphabet de pile consiste en deux symboles : le symbole initial z_0 qui peut être lu mais n'est jamais modifié ; et un autre symbole qui est utilisé par toutes les autres transitions. Les automates à un compteur reconnaissent un sous-ensemble strict des langages hors-contexte, et reconnaissent tous les langages réguliers (sans utiliser le compteur), et également certains langages hors-contexte non réguliers (par exemple $\{a^n b^n \mid n \in \mathbb{N}\}$).

Pour ces automates aussi, comme pour les automates à pile, la version déterministe est moins puissante que la version nondéterministe : cf [6], il y a des langages comme $L_2 = \{a^i b^j c^k \mid i, j, k \in \mathbb{N}, (i = j) \vee (j = k)\}$ qui sont reconnus par des automates nondéterministes à un compteur mais ne sont même pas reconnus par des automates à pile déterministes ou même inambigus (ce langage est en effet inhéremment ambigu). Il y a aussi des langages qui sont reconnus par des automates inambigus à un compteur mais ne sont même pas reconnus par des automates à pile déterministes, cf [9].

Les automates à un compteur, au demeurant, ne sont pas capables de reconnaître tous les langages hors-contexte, vu qu'ils ne peuvent pas reconnaître le langage des palindromes. Ils ne peuvent pas non plus reconnaître tous les langages hors-contexte déterministes, vu qu'ils ne peuvent pas reconnaître le langage des palindromes où le milieu de mot est explicitement marqué. Tout ceci suit de [12], Theorem 5 : l'ensemble de configurations accessibles à partir d'un mot pour un automate à un compteur ne peut croître que de manière polynomiale en le mot, alors qu'il faut un nombre exponentiel de configurations pour tester par exemple les palindromes.

Ainsi, les six types de machine (avec compteur ou avec pile, nondéterministe ou inambigu ou déterministes) reconnaissent à chaque fois des ensembles de langages différents. Les inclusions qui sont vraies sont seulement celles qui suivent des définitions : pour chaque formalisme (avec compteur ou avec pile), les formalismes nondéterministes sont plus puissants que les inambigus qui sont plus puissants que les déterministes ; et pour chaque type de déterminisme la version avec compteur est moins puissante que la version avec pile. Du reste toutes ces inclusions sont strictes.

Machines de Turing. Généralisant les automates à pile, les *machines de Turing* permettent de reconnaître tous les langages décidables (cf les notes de cours de INF105). En quelque sorte, les machines de Turing étendent les automates à pile avec la possibilité de se *déplacer dans la*

pile, de façon bidirectionnelle (ce qui rappelle aussi un peu les automates bidirectionnels). Pour simplifier la définition, on décrit plutôt les machines de Turing comme s'exécutant sur un ruban, infini dans une direction, qui contient initialement le mot d'entrée, et peut être utilisé comme mémoire de travail (comme une pile mais dans laquelle la machine peut naviguer librement). Formellement :

Definition 5.3. *Une* machine de Turing déterministe *est un tuple* $(Q, \Sigma, z_0, \delta, q_0, F)$ *où* Q *est un ensemble fini d'*états*,* $q_0 \in Q$ *est l'*état initial*,* $F \subseteq Q$ *est l'ensemble des états finaux,* Σ *est l'alphabet,* $z_0 \notin \Sigma$ *est le* symbole blanc*, et* $\delta \colon Q \times \Sigma \sqcup \{z_0\} \to Q \times \Sigma \sqcup \{z_0\} \times \{-1, 0, 1\}$ *est une fonction partielle de transition.*

La sémantique est la suivante. Initialement, le ruban contient le mot d'entrée de Σ^**, suivi d'un nombre infini de caractères* z_0 *; la machine est dans l'état* q_0 *; et une* tête de lecture *est positionnée sur la première case. Ensuite, à chaque étape, on applique la fonction de transition* δ *si c'est possible : la machine regarde son état actuel, et le symbole de la case où elle se trouve actuellement, ce qui définit la transition à appliquer. L'application d'une transition mène la machine à changer d'état, à changer la valeur de la case courante pour un nouveau symbole, et à se déplacer d'une case soit à gauche* (-1) *soit à droite* (1)*, le calcul s'arrêtant si c'est impossible (c'est-à-dire si on tente de se déplacer à gauche à partir de la première case). Le calcul est réussi si la machine atteint un état final. La machine accepte le langage des mots sur lequel le calcul est réussi.*

La notion de machine de Turing est intrinsèquement bidirectionnelle (comme les automates bidirectionnels), mais avec la possibilité en quelque sorte de modifier le mot courant. Une notion de *machine de Turing nondéterministe*, et même de *machine de Turing alternante*, peut être définie ; mais ces machines de Turing peuvent être déterminisées, c'est-à-dire que leur pouvoir d'expression est le même que celui des machines de Turing déterministes de la définition ci-dessus. Bien sûr, les machines de Turing peuvent reconnaître tous les langages hors-contexte.

Une classe intermédiaire entre les langages hors-contexte et les machines de Turing sont les machines de Turing *linéairement bornées*, où il y a une borne linéaire sur la taille de ruban utilisée en fonction du mot d'entrée. On ne sait pas si ces machines peuvent être déterminisées, c'est-à-dire si tout langage reconnu par une machine de Turing linéairement bornée non-déterministe peut être reconnu par une machine de Turing linéairement bornée déterministe (potentiellement beaucoup plus grande) : ce problème est ouvert depuis 1964 [15]. Un formalisme de grammaires équivalent existe : les grammaires *sensibles au contexte*, ou de façon équivalente les grammaires *essentiellement non-contractantes*. Les grammaires sensibles au contexte sont définies comme les grammaires hors-contexte mais avec des productions de la forme $\alpha A \beta \to \alpha \gamma \beta$ où A est un non-terminal, γ est un proto-mot non vide, et α et β sont également des proto-mots ; ou $A \to \epsilon$ pour A un non-terminal. Une grammaire *essentiellement non-contractante* a des productions arbitraires $\alpha \to \beta$ où α et β sont des proto-mots tels que $|\alpha| \leq |\beta|$, et également des productions $S \to \epsilon$. Les *langages sensibles au contexte* sont les langages reconnus par ces formalismes : ils incluent par exemple $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ ou $\{a^n b^n c^n d^n \mid n \in \mathbb{N}\}$. Un résultat majeur est que les langages sensibles au contexte sont clos par complémentation (c'est un cas particulier du théorème d'Immerman-Szelepcsényi) ; ils sont également clos par union, intersection, et concaténation. L'appartenance d'un mot à un langage sensible au contexte n'est pas forcément aisée à déterminer (ce problème est en effet potentiellement NP-complet), même si elle est faisable en temps linéaire pour certains langages (par exemple les exemples qui précèdent).

En tout cas, le formalisme des machines de Turing (déterministes ou non) est d'ailleurs utile, au-delà de la calculabilité, pour donner des définitions précises de classes de complexité, par exemple pour formaliser un modèle dans lequel définir la classe PTIME des problèmes qui peuvent être décidés en temps polynomial dans ce modèle. En revanche il n'est pas évident de savoir si

une machine de Turing nondéterministe s'exécutant en temps polynomial a toujours une machine déterministe équivalente qui satisfasse cette même condition : c'est la question P versus NP !

Automates à plusieurs piles ou plusieurs compteurs. On peut s'intéresser aux modèles des automates à pile si on les étend avec plusieurs piles, ou avec plusieurs compteurs. Cependant, il s'avère que ces modèles sont en fait déjà équivalents en pouvoir d'expressivité aux machines de Turing. En effet, un automate avec deux piles peut émuler une machine de Turing : l'intuition étant que la première pile contient ce qu'il y a à gauche de la tête de lecture, et la seconde pile ce qu'il y a à droite.

Il s'avère aussi que le modèle des machines à deux compteurs est déjà suffisant pour émuler des machines de Turing. Intuitivement, une machine à deux compteurs permet d'émuler une machine utilisant n'importe quel nombre constant fixé de compteurs, et en représentant les tuples de valeurs de compteurs par leur codage de Gödel dans l'un des compteurs, et en utilisant l'autre pour les opérations arithmétiques nécessaires pour modifier ou pour tester les valeurs des compteurs. Ensuite, une pile peut être représentée en binaire en utilisant un compteur auxiliaire pour modifier la pile (en doublant la valeur pour insérer zéro, et en doublant la valeur et en ajoutant 1 pour insérer 1).

Il y a cependant une variante des automates à compteurs qui ressemble à des automates à plusieurs compteurs mais qui n'est pas aussi expressive que les machines de Turing : les *vector addition systems* (VAS), qui sont proches d'un autre formalisme, les *réseaux de Petri*. Informellement, ce sont des automates à plusieurs compteurs où les transitions ajoutent un vecteur à la configuration courante de compteurs (d'où le nom) et où on impose la condition que les compteurs doivent rester positifs mais *sans* pouvoir la tester, c'est-à-dire qu'une transition qui aurait cet effet ne peut pas être franchie. Les langages définissables par ce formalisme sont incomparables avec les langages hors-contexte : autrement dit certains langages hors-contexte ne sont pas définissables par des VAS, mais certains langages définissables par des VAS ne sont pas hors-contexte [14]. Les langages définis par les VAS sont en revanche un sous-ensemble (strict) des langages sensibles au contexte. Remarquons aussi que la complexité précise du problème d'*accessibilité* dans les VAS, c'est-à-dire déterminer s'il existe une séquence de configurations d'une configuration à l'autre, n'a été identifiée qu'en 2022 [5] suivant une avancée majeure en 2020 [4].

6 Algorithmes de chaînes de caractères

Cette section présente des algorithmes pour résoudre des problèmes concrets sur des chaînes de caractères, qui peuvent parfois être vus en termes d'automates.

Un problème central est de rechercher un mot u comme facteur d'un mot v, c'est-à-dire en identifier toutes les occurrences. Nous proposons plusieurs algorithmes pour cela : l'algorithme de *Knuth-Morris-Pratt*, qui fonctionne intuitivement par indexation du mot u, et l'algorithme des *arbres suffixe*, qui fonctionne intuitivement par indexation du mot v. Les arbres suffixe ont également d'autres usages dont nous discuterons.

6.1 Recherche naïve de facteur

Le problème de recherche de facteur (ou recherche de sous-chaîne, ou *string-searching algorithm* en anglais) demande, étant donné un mot u et un mot v, si on peut trouver u comme facteur de v ; et, le cas échéant, de lister toutes les occurrences.

L'algorithme naïf pour cette tâche consiste à tester chaque position de départ dans v, et à tester l'égalité avec u. Dans le pire cas, la complexité de cet algorithme est de $O(|u| \times |v|)$. En

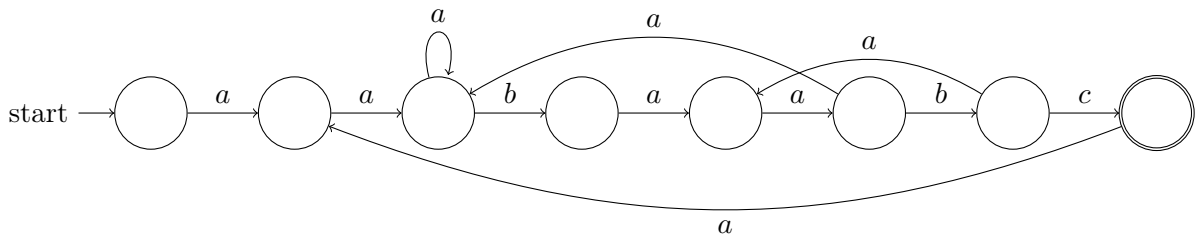


FIGURE 7 – Automate suffixe complet pour $u = aabaabc$, où on a omis toutes les transitions qui ramènent à l'état initial.

pratique, on peut abandonner les comparaisons dès lors qu'un caractère ne coïncide pas, mais cela ne change pas la complexité, par exemple si on cherche $a^n b$ dans a^m avec $n < m$.

Le problème de recherche de facteur admet de nombreux algorithmes plus efficaces que l'approche naïve, à la fois en théorie (optimisant la complexité dans le pire cas, la complexité moyenne, etc.), et en pratique (optimisation des facteurs constants et des cas d'utilisation typiques), dans de nombreux domaines. On voit dans ce cours deux de ces algorithmes efficaces : l'algorithme de Knuth-Morris-Pratt (que l'on verra entièrement), et l'approche par arbres suffixe (dont on admettra la construction).

6.2 Algorithme de Knuth-Morris-Pratt (KMP)

L'idée de l'algorithme KMP peut se formuler en termes d'automates. Rappelons-nous qu'on souhaite chercher un mot u comme facteur d'un mot v. L'idée de base est que l'on peut souvent s'épargner certaines comparaisons en ne revenant pas en arrière sur v.

Exemple 6.1. *Si l'on cherche $u = abcdefghijk$ dans un mot v, quand une position de v ne coïncide pas avec u, alors on peut reprendre directement à la position suivante. Il faut reprendre la recherche de u depuis le début si le caractère qui ne coïncide pas était autre chose qu'un a, et depuis la deuxième position sinon.*

En revanche, la recherche d'autres motifs peut se comporter différemment :

Exemple 6.2. *Si l'on cherche $u = aaaaab$ dans un mot v sur l'alphabet $\Sigma = \{a, b, c\}$, quand une position de v ne coïncide pas avec u, alors plusieurs cas sont possibles. Si on a lu un b au lieu d'un a, ou si on a lu un c, alors on peut reprendre du début. Mais si on a lu un a au lieu du b final de u, alors il faut rester à la même position dans u, tant que l'on lit des a : en effet le prochain b suffira à créer un résultat.*

On peut imaginer d'autres comportements encore :

Exemple 6.3. *Si l'on cherche $u = aabaabc$ dans un mot v, alors de nombreux cas se présentent quand on lit un caractère qui n'est pas celui attendu :*
- *Lire un a à la place du c final doit nous faire reprendre du 3e a*
- *Lire un a à la place d'un b doit nous faire reprendre du 2e a*
- *Sinon, on reprend du début (ou du 2e caractère si on a lu un a).*

Comment peut-on généraliser ces observations ? La notion d'*automate suffixe* du mot u est une façon de régler ces problèmes. Un *automate suffixe* pour u est un DFA qui accepte tous les mots qui se terminent par u.

Exemple 6.4. *Un automate suffixe pour $u = aabaabc$ est donné en Figure 6.2.*

Un automate suffixe existe bien sûr toujours (car le langage des mots se terminant par u est régulier), et il est intuitivement évident qu'il en existe forcément qui a $|u| + 1$ états et suit la structure illustrée en Figure 6.2, avec des états correspondant à des préfixes successifs de u. Il s'agit par ailleurs d'un automate complet émondé : il n'y a pas de transitions non-définies ou de puits non-acceptants, puisque tout mot peut se poursuivre en un mot accepté. La difficulté consiste à comprendre comment construire un tel automate de manière efficace.

Remarquons d'abord qu'une fois qu'on a un automate suffixe pour u, il est alors trivial de lire le mot v et de détecter toutes les fois où on passe par un état final de l'automate, ce qui détecte qu'un facteur u vient de se terminer. Ainsi, une fois l'automate suffixe construit, la recherche de u dans v s'effectue en $O(|v|)$. Noter que les transitions à partir de l'état final ne reviennent pas forcément à l'état initial : par exemple pour le motif $u = aaa$ il y a une boucle étiquetée par a sur l'état final.

Dans ce qui suit, nous introduisons d'abord une variante des automates, appelée *DFA avec transition d'échec*, que nous utiliserons pour construire les automates suffixe. Nous voyons ensuite comment utiliser de tels automates pour la recherche de facteurs avec la même complexité qu'un DFA normal. Enfin, nous voyons comment construire de tels automates, d'abord naïvement, puis de façon plus efficace, ce qui constitue le cœur de l'algorithme KMP.

Transitions d'échec. Un *DFA avec transitions d'échec* est un DFA non nécessairement complet où chaque état peut disposer d'une unique transition sortante sans étiquette, la *transition d'échec*. La sémantique est la suivante : lorsque l'automate se situe en un état et tente de lire une lettre pour laquelle il n'y a pas de transition sortante, alors, si la transition d'échec existe, l'automate l'emprunte, et tente à nouveau de lire une transition sortante à partir de l'état où il se trouve alors ; et l'automate procède ainsi de façon répétée jusqu'à effectivement suivre une transition qui n'est pas une transition d'échec. On impose que les transitions d'échec forment un graphe acyclique afin que ce processus se termine. Si enfin l'automate tente de lire une lettre depuis un état qui n'a pas de transition sortante pour cette lettre et pas de transition d'échec, alors la lettre en question est simplement ignorée et l'automate reste dans l'état où il se trouve à ce moment. Noter que ce dernier point change le comportement de l'automate même en l'absence de transitions d'échec, c'est-à-dire que sur un DFA avec transitions d'échec où il n'y a en fait aucune transition d'échec, le comportement est le même que sur un DFA non-nécessairement complet à ceci près que les tentatives d'emprunter des transitions inexistantes font que l'automate ignore la lettre et reste dans le même état (alors qu'il rejetterait dans la sémantique habituelle). Ce choix est motivé par le fait que les automates suffixe sont des automates qui ne doivent jamais définitivement rejeter.

La sémantique des transitions d'échec est un peu différente de celle des ϵ-transitions : en effet les ϵ-transitions peuvent être franchies ou non au choix de l'automate, alors que les transitions d'échec ne peuvent être franchies que quand il n'y a pas de transition normale qui puisse l'être. Ainsi, l'automate reste déterministe : il n'y a jamais deux transitions différentes qui peuvent être lues pour la même lettre depuis le même état. On se convainc sans difficulté qu'un DFA avec transitions d'échec A peut être transformé en un DFA au sens habituel de la manière suivante :

Proposition 6.5. *Étant donné un DFA avec transitions d'échec A sur l'alphabet Σ, on peut récrire A en un DFA au sens habituel en temps $O(|A| \times |\Sigma|)$.*

Démonstration. On traite le graphe acyclique des transitions d'échec suivant un tri topologique, et à chaque fois qu'on traite un état on assure qu'il n'a plus de transition d'échec sortante, qu'il a une transition sortante pour chaque lettre, et que sa sémantique est respectée au sens où le langage reconnu depuis cet état dans l'automate réécrit est le même que dans l'automate avec transitions d'échec originale.

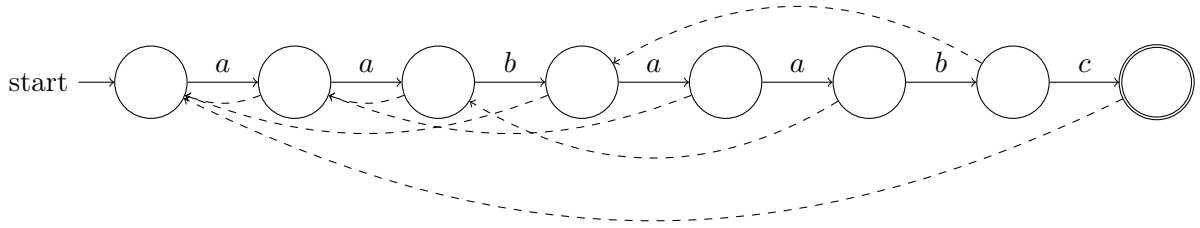


FIGURE 8 – Automate suffixe avec transitions d'échec pour $u = aabaabc$. Les transitions d'échec sont en pointillés, elles vont toutes de la droite vers la gauche.

Le cas de base est celui d'un état sans transition d'échec : on ajoute alors des boucles pour toutes les lettres pour lesquelles l'état n'a pas de transition. L'état a bien une transition sortante pour chaque lettre et la sémantique est correcte.

Le cas d'induction est celui d'un état q ayant une transition d'échec vers un état q' ; par hypothèse d'induction q' n'a plus de transition d'échec sortante. On élimine la transition d'échec et on fait alors l'opération suivante : pour chaque lettre a pour laquelle q n'a pas de transition sortante, on prend q'' la cible de la transition sortante de q' étiquetée a (qui existe suivant l'invariant), et on ajoute une transition étiquetée a de q à q''. Ceci garantit que q n'a plus de transition d'échec, a une transition sortante par lettre, et la sémantique est correcte en utilisant l'invariant sur q'. □

Ainsi, étant entendu que les transitions d'échec peuvent être éliminées, à quoi bon les définir ? Outre qu'elles allègent les représentations graphiques des automates, elles vont rendre l'exposition technique plus claire : il n'est pas beaucoup plus compliqué d'utiliser un automate suffixe avec transitions d'échec, et il sera plus facile de les calculer sous cette forme. Du reste, d'un point de vue pratique, le facteur $|\Sigma|$ gagné grâce aux transitions d'échec n'est pas anecdotique.

Exemple 6.6. *Un automate suffixe pour $u = aabaabc$ avec transitions d'échec est donné en Figure 6.2. Noter que sa structure est très simple (il s'agit de transitions représentant le mot u directement), la seule chose non évidente est la structure des transitions d'échec. On verra plus tard qu'en réalité on veut que les transitions d'échec pour un état correspondant à un préfixe u' de u pointent vers l'état correspondant au plus grand préfixe strict de u' qui soit aussi un suffixe de u'.*

Utilisation des automates suffixes avec transitions d'échec. Supposons pour l'instant qu'on a réussi à construire un DFA avec transitions d'échec qui est un automate suffixe pour u. Dans ce cas, on peut lire v dans cet automate et détecter les occurrences de u comme précédemment expliqué. Toutefois il y a une subtilité : les transitions d'échec pourraient nous faire perdre du temps. En effet naïvement il semble que la lecture d'une lettre de v peut nous faire faire jusqu'à $|u|$ opérations dans l'automate, pour suivre jusqu'à $|u|$ transitions d'échec.

Heureusement, en réalité, ce problème ne se pose pas, car les automates que l'on construit seront *bien ordonnés* en le sens suivant : il y a un ordre total $<$ sur les états qui satisfait les conditions suivantes :

— l'état initial est le premier état de $<$
— les transitions normales allant d'un état q à un état q' satisfont $q < q'$,
— les transitions d'échec de q à q' satisfont $q > q'$.

Cet ordre sera simplement obtenu en considérant les états de gauche à droite dans les illustrations.

On peut alors montrer que la complexité totale est quand même en $O(|v|)$ par une analyse de potentiel. En effet, à chaque fois que l'on emprunte une transition d'échec, notre position dans l'automate décroît strictement ; or elle ne croît que quand on lit une lettre. Ainsi, chaque lettre lue dans v nous coûte une opération (pour lire la lettre), et nous pouvons "mettre de côté" une autre opération (consistant en une transition d'échec qui sera potentiellement empruntée plus tard). Lorsque l'on suit une transition d'échec, on "dépense" les opérations mises de côté ; on a toujours un stock positif ou nul d'opérations de côté. Ainsi, à un facteur 2 près, la lecture d'un mot dans un DFA A avec transitions d'échec qui est bien ordonné est en $O(|v|)$, même s'il est possible qu'une lettre de v conduise ponctuellement à $O(|A|)$ opérations.

Ainsi, nous pouvons utiliser des automates bien ordonnés avec transitions d'échec, et il ne reste plus qu'à voir comment construire efficacement de tels automates.

Construction des DFA avec transition d'échec. Étant donné le mot u, on souhaite construire un automate suffixe pour u avec transitions d'échec. La structure de cet automate est claire : on a $|u|+1$ états correspondant aux préfixes de u, des transitions normales qui nous permettent de progresser dans les états, l'état initial à gauche, un unique état final à droite, et des transitions d'échec allant de droite à gauche. Tout le problème est de savoir quelles transitions d'échec adopter. Intuitivement, à partir d'un point du motif, une transition d'échec nous indique à ce que l'on doit faire quand on lit le mauvais caractère, c'est-à-dire une lettre qui n'est pas celle de l'unique transition allant vers la droite. Il faut alors reprendre la lecture du motif à partir d'une position antérieure, qu'il s'agit à présent de déterminer.

L'observation-clé est la suivante : si on a lu un certain préfixe u' du motif u, alors, si le prochain caractère n'est pas le bon, on souhaite reprendre la lecture à un préfixe strict u'' de u' qui correspond à ce qui a été lu jusque là. En d'autres termes : on veut un préfixe strict u'' de u' qui soit également un suffixe (strict) de u'. Lequel choisir ? Simplement le plus long possible. En effet, si la lecture n'est toujours pas possible à cet endroit, on pourra remonter à un préfixe plus court en répétant le processus.

Exemple 6.7. *Dans le motif $u = ababac$, pour le préfixe $u' = ababa$, le plus long préfixe strict qui est également un suffixe est aba. De fait, si on lit ababa dans v et que le prochain caractère est b, alors on sait qu'on vient de lire aba et il se trouve qu'on peut lire b à partir de là.*

Dans le motif $u = abacabad$, pour le préfixe $u' = abacaba$, le plus long préfixe strict qui est également un suffixe est à nouveau $u'' = aba$. Donc, si on lit abacaba dans v et qu'on lit autre chose que d, on sait qu'on vient de lire aba et on tente alors de lire le nouveau caractère. Si le nouveau caractère est c, alors on a lu abac et on continue ainsi. Si le nouveau caractère est b, alors on ne peut pas le lire depuis $u'' = aba$, et il faut donc reprendre à un préfixe strict de u''. Le plus long préfixe strict qui est également un suffixe est $u''' = a$. Depuis u''', on peut lire b, et on vient de lire ab. Sinon, on soit prendre ϵ comme préfixe strict de u''', à partir duquel on peut lire a (pour arriver à a). Si on lit encore un autre caractère, on reste sur la lecture de ϵ.

Dans le motif $u = babbabc$, pour le préfixe $u' = babbab$, le plus long préfixe strict qui est également un suffixe est bab. Il y en a un deuxième, plus court, qui est b. Ceci illustre ce qu'il faut faire après avoir lu babbab :

- *Soit on lit un c et on arrive à l'état final (transition normale)*
- *Soit on lit un b et on poursuit en ayant lu babb (transition d'échec vers bab, puis transition normale b)*
- *Soit on lit un a et on poursuit en ayant lu ba (transition d'échec vers bab suivie d'une transition d'échec vers b, puis transition normale a).*

Les transitions d'échec de notre DFA correspondent donc à l'indication, pour chaque préfixe u' du motif u, de la longueur du plus long préfixe strict de u' qui est également un suffixe de u'. Cette

information peut être stockée dans un tableau T, appelé tableau des correspondances partielles. On peut manifestement la calculer de manière naïve en temps cubique : pour chaque préfixe u' de u, pour chaque préfixe strict de u' par taille décroissante, vérifier si c'est un suffixe. Mais on peut la calculer de manière plus intelligente, de proche en proche.

(Remarque : dans certaines implémentations de KMP, par exemple celle de la Wikipedia anglophone, le tableau des correspondances partielles est définie d'une manière un peu différente pour tenir compte d'une optimisation : pour un mot u, un préfixe u', il ne sert à rien de considérer un préfixe strict u'' de u' qui soit suffixe de u' mais où le caractère a qui suit u'' est le même que celui qui suit u'. En effet, si on lit a on n'utilisera pas la fonction d'échec, et si on lit autre chose que a ça ne sera jamais utile de tenter de lire a depuis u''. Or il se peut que le plus grand préfixe strict u'' qui soit également un suffixe ait cette propriété. Pour cette raison on peut vouloir écrire dans la table des suffixes plus petits mais où le prochain caractère est différent, ce qui accélère la lecture en pratique (mais ne change pas la complexité asymptotique). En revanche cette optimisation ne concerne que ce qui est écrit au final dans le tableau ; il faut bien garder trace du préfixe strict maximal au cours de la lecture, sans tenir compte de cette optimisation, car c'est lui qu'il faudra tenter d'étendre dans la suite de l'algorithme.)

Calcul du tableau des correspondances partielles. On procède par induction sur le motif u. Pour le premier caractère de u, la longueur du plus long préfixe strict de u est de zéro (le seul préfixe strict). Ensuite, si l'on suppose que l'on a calculé le tableau T jusqu'à des préfixes de longueur i et qu'on veut calculer la case $T[i+1]$ pour le préfixe u_{i+1} de longueur $i+1$ de u, deux cas sont possibles.

— Soit le $i+1$-ème caractère de u est le caractère a qui suit dans u le préfixe strict maximal p_i qui avait été trouvé pour u_i et stocké dans $T[i]$, i.e., $T[i] = |p_i|$. Dans ce cas, on étend p_i avec a et on écrit $T[i+1] = T[i] + 1$. C'est manifestement optimal : il ne peut pas y avoir de préfixe strict plus long de u_{i+1} qui soit également un suffixe, sinon ceci contredirait la maximalité du préfixe strict p_i.

— Soit c'est un autre caractère b. Dans ce cas, il faut considérer un autre préfixe plus court. Mais la réponse recherchée pour la case $i+1$, à savoir le préfixe strict maximal de u_{i+1} qui est également un suffixe, est forcément un préfixe strict (non nécessairement maximal) de u_i qui est également un suffixe de u_i et qui peut s'étendre avec b. Ce qu'on veut naturellement faire est utiliser $T[|p_i|]$ pour trouver un préfixe strict, et ainsi de suite itérativement. Cette construction est correcte grâce au résultat suivant :

Lemma 6.8. *Soit u un mot, et considérons l'ensemble U des préfixes stricts de u qui sont également des suffixes de u. Notons $f\colon \Sigma^* \setminus \{\epsilon\} \to \Sigma^*$ la fonction qui associe à un mot v le plus grand préfixe strict de v qui est également un suffixe (strict) de v. Considérons la séquence obtenue itérativement en posant $u = u_1$, $u_2 = f(u_1)$, et ainsi de suite jusqu'à atteindre $u_n = \epsilon$ (auquel cas $f(u_n)$ n'est pas défini car ϵ n'a aucun suffixe strict). Alors on a $U = \{u_1, \dots, u_n\}$.*

Ainsi, grâce au lemme, comme on sait que le préfixe strict maximal à trouver pour u_{i+1} se compose d'un préfixe de u_i étendu avec le caractère b aussi long que possible, on sait que trouver le bon préfixe dans l'ensemble U peut se faire en appliquant itérativement la fonction f, qui est donnée par les valeurs de T calculées jusque là.

Démonstration. Une inclusion est claire : $\{u_1, \dots, u_n\} \subseteq U$. En effet, u_2 est un préfixe strict de u qui est également un suffixe de u. Du reste, la propriété suivante de transitivité est vraie : si x est un préfixe strict de y qui est un préfixe strict de z, et x est un suffixe de y et y est un suffixe de z, alors x est un préfixe strict de z et un suffixe de z.

Pour l'autre inclusion, prenons v le plus petit préfixe strict de u qui est un suffixe de u et qui n'est pas de la forme u_i. Soit i_0 le dernier index tel que u_{i_0} soit défini et soit plus long que v.

TABLE 1 – Tableau des correspondances partielles pour le mot u de l'Exercice 6.9. Sous la dernière lettre de chaque préfixe non-vide, la colonne $1 \leq i \leq |u|$ correspondant au préfixe de longueur i on indique la valeur de $T[i]$

a	b	a	b	c	a	b	a	b	a	a	b	a	b	c
0	0	1	2	0	1	2	3	4	3	1	2	3	4	5

Ainsi on sait que v est un préfixe strict de u_{i_0}, et que v est un suffixe de u, donc de u_{i_0} puisque u_{i_0} est un suffixe de u. Ainsi v est un préfixe strict de u_{i_0} qui en est également un suffixe. Ainsi $f(v)$ devrait être défini, et être égal à u_{i_0} ou plus long. Mais on a supposé qu'il n'était pas défini ou qu'il était moins long, contradiction. □

La complexité de cet algorithme est linéaire, avec là encore une analyse de potentiel pour le justifier. En effet, la case $T[i+1]$ contient au plus $T[i]+1$, et potentiellement on la définit en consultant $T[T[i]] < T[i]$, $T[T[T[i]]] < T[T[i]]$, etc. Donc, le nombre d'étapes nécessaires au calcul de $T[i+1]$ est d'au plus $T[i]$. Donc là encore à chaque fois qu'on définit $T[i+1] = T[i]$ on peut incrémenter le potentiel de 1 et consommer cette valeur plus tard quand on déréférence le tableau.

Une illustration de l'algorithme est disponible en ligne [4]

Exercice 6.9. *Calculer le tableau des correspondances partielles T pour le mot $u = ababcababaababc$.*

Solution 6.10. *La table est donnée en Table 1.*

Conclusion. L'algorithme KMP fonctionne avec un prétraitement du motif en temps linéaire ($O(|u|)$), et ensuite s'exécute en temps $O(|v|)$, soit un temps total de $O(|u| + |v|)$. C'est bien mieux que les $O(|u| \times |v|)$ de l'algorithme naïf !

6.3 Arbres suffixe

On présente maintenant un autre algorithme : les *arbres suffixe*. On cherche intuitivement à indexer un long texte u pour construire une représentation efficace de tous ses suffixes. On utilisera pour cela la structure de *trie*, plus spécifiquement de *trie compressé*.

Definition 6.11. *Un* trie *est un arbre enraciné où les arêtes sortantes de chaque nœud portent des étiquettes de l'alphabet Σ ; il y a au plus une arête sortante pour chaque nœud et pour chaque étiquette. De plus, on distingue certains nœuds internes, et toutes les feuilles sont considérées comme distinguées.*

Un trie représente *un ensemble de mots en prenant les chemins allant de la racine jusqu'à chaque nœud distingué (en particulier chaque feuille).*

Exemple 6.12. *Un trie d'exemple pour $U = \{banal, banana, band, bandana, bandit, can, candle, candy, cat, catalog\}$ est donné en Figure 9.*

On peut aussi voir un trie comme une forme restreinte d'automate : chaque nœud de l'arbre est un état, la racine est l'état initial, les nœuds distingués sont les états finaux, et les arêtes représentent les transitions. L'automate est déterministe, et son langage est l'ensemble de mots représenté par le trie. L'automate est émondé, car on a imposé dans la définition que toutes les feuilles soient distinguées. L'automate reconnaît un automate fini, car il n'a pas de cycle. Mais

4. `https://cmps-people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html`

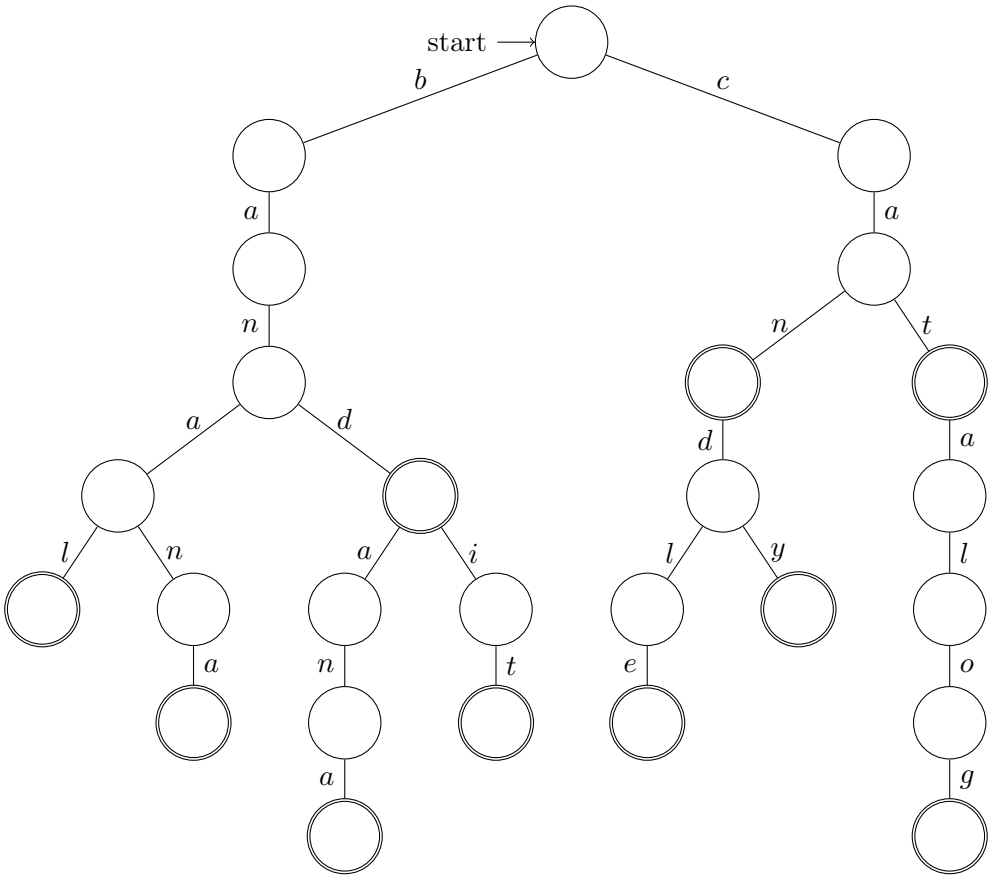


FIGURE 9 – Trie d'exemple pour l'Exemple 6.12

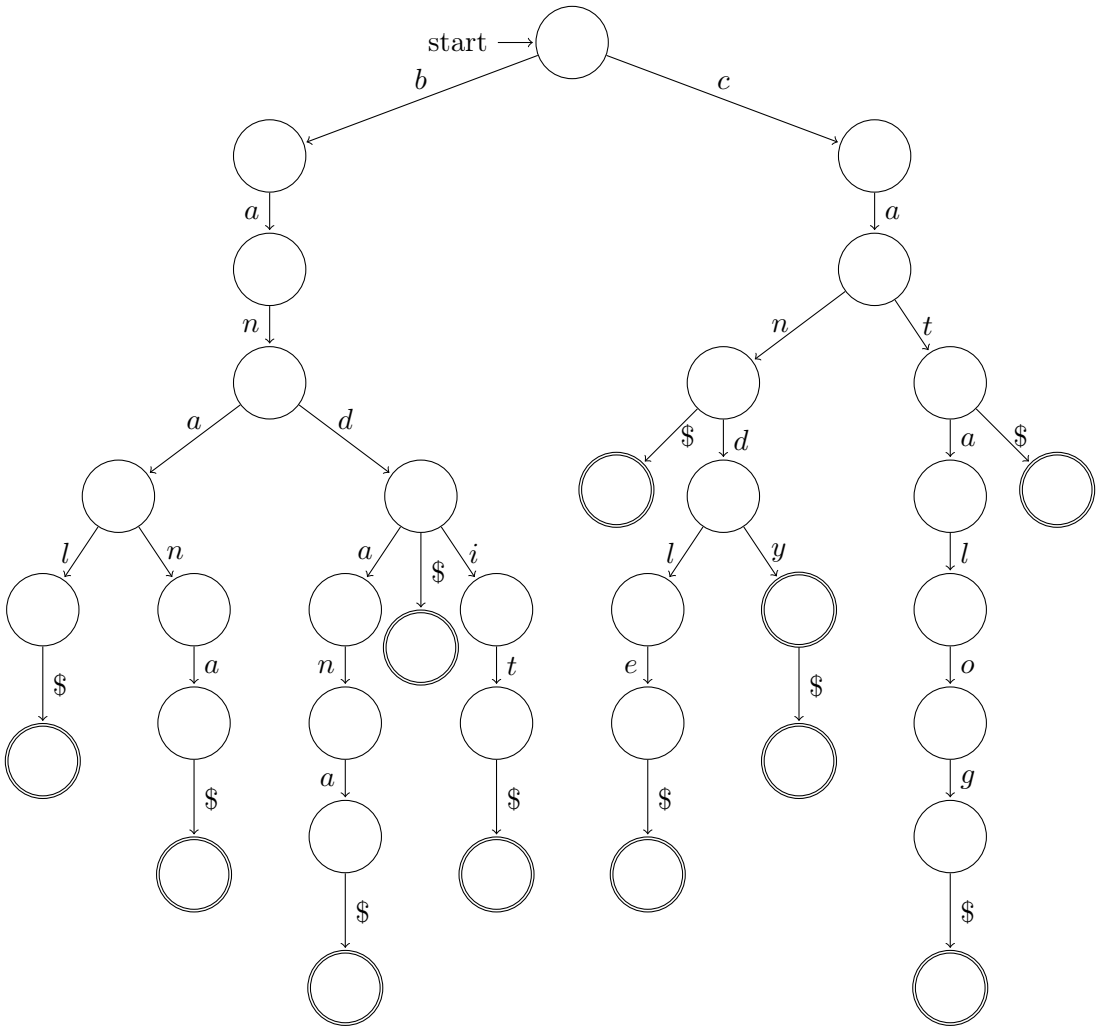


FIGURE 10 – Trie d'exemple pour l'Exemple 6.12, mais en ajoutant $ à la fin de chaque mot, de sorte que les nœuds distingués sont précisément les feuilles.

on a également imposé que chaque état ait au plus une transition entrante, i.e., il n'y a pas de "partage".

Étant donné un ensemble de mots, on peut toujours construire un trie qui les représente : on considère chaque mot et on l'insère en temps linéaire en la longueur du mot. La complexité est linéaire en la longueur totale des mots. Du reste, le trie est unique à isomorphisme près. Remarquer que, si aucun mot n'est préfixe d'un autre, alors il n'y a aucun autre nœud distingué que les feuilles. On peut facilement garantir cette condition en ajoutant à chaque mot de U un terminateur unique, conventionnellement noté \$: comparer la Figure 9 et 10. Pour notre application, on voudra calculer un *trie suffixe* du mot u, c'est-à-dire un trie représentant la collection des suffixes de u. On rappelle qu'on peut voir ce trie comme un automate (déterministe, acyclique et sans "partage") qui accepte précisément le langage des suffixes du texte u – à distinguer de KMP où on construisait un automate reconnaissant les mots qui se terminent par le motif recherché. Conventionnellement, dans un trie suffixe, on ajoutera le symbole spécial \$ à la fin de u pour garantir qu'aucun suffixe n'est préfixe d'un autre.

On va voir comment calculer efficacement un trie suffixe en fonction du mot d'entrée. Malheureusement, la définition actuelle ne va pas pouvoir convenir à un calcul efficace : la taille du trie suffixe est potentiellement quadratique (et elle est clairement au plus quadratique).

Exemple 6.13. *Sur le mot $a^nba^n\$$, les suffixes seront $a^iba^n\$$ pour $0 \leq i \leq n$ et $a^i\$$ pour $0 \leq i \leq n$ et l'arbre est de taille quadratique, cf Figure 11.*

Pour cette raison, on a besoin d'une notion un peu différente, celle d'un *trie compressé* :

Definition 6.14. *Un* trie compressé *(ou* arbre radix*, ou* arbre Patricia*) est un arbre défini comme un trie sauf que les arêtes portent pour étiquette un mot non-vide. On impose que, pour chaque nœud n, pour chaque lettre de l'alphabet a, il y a au plus une seule arête sortante de n étiquetée par un mot qui commence par a.*

Les tries compressés généralisent les tries ordinaires, qui sont le cas où chaque arête est étiquetée par un mot formé d'une seule lettre. On imposera souvent sur les tries compressés qu'ils soient "aussi compressés que possible" c'est-à-dire qu'il n'y a pas de nœud non-distingué ayant un unique enfant : en effet ce nœud peut alors se fusionner avec son unique enfant en concaténant l'étiquette de l'arête entrante et de l'arête sortante.

Un exemple de trie compressé correspondant à l'Exemple 6.12 est donné en Figure 12 : c'est la version compressée du trie de la Figure 9. Un exemple de trie compressé correspondant à la Figure 11 est donné en Figure 13.

On peut convertir n'importe quel trie en trie compressé, en temps linéaire, en supprimant itérativement les nœuds ayant un unique enfant en concaténant l'étiquette de l'arête parent et celle de l'arête enfant. À nouveau, pour cette raison, le trie compressé représentant un ensemble de mots est unique à isomorphisme près.

On peut toujours insérer un mot dans un trie compressé en temps linéaire, simplement en navigant dans le trie. La complication est qu'il peut être nécessaire de scinder une arête étiquetée par un mot $w = uv$ en une arête u puis une arête v, avec un embranchement au nœud intermédiaire auquel on insère le nouvel enfant.

La représentation d'un trie compressé n'est en réalité pas vraiment plus compacte que celle d'un trie. En effet, là où on avait un chemin de n nœuds de degré 1, on a maintenant une arête étiquetée par un mot de longueur n. C'est plus concis en pratique, mais pas en termes d'occupation mémoire. Toutefois, pour la construction d'un *trie suffixe compressé*, on remarque que les étiquettes des arêtes du trie compressé correspondent toujours à des facteurs du mot, qu'on peut représenter de manière concise par la position de début et la position de fin. Ceci prend un espace constant (deux entiers) quelle que soit la longueur du facteur.

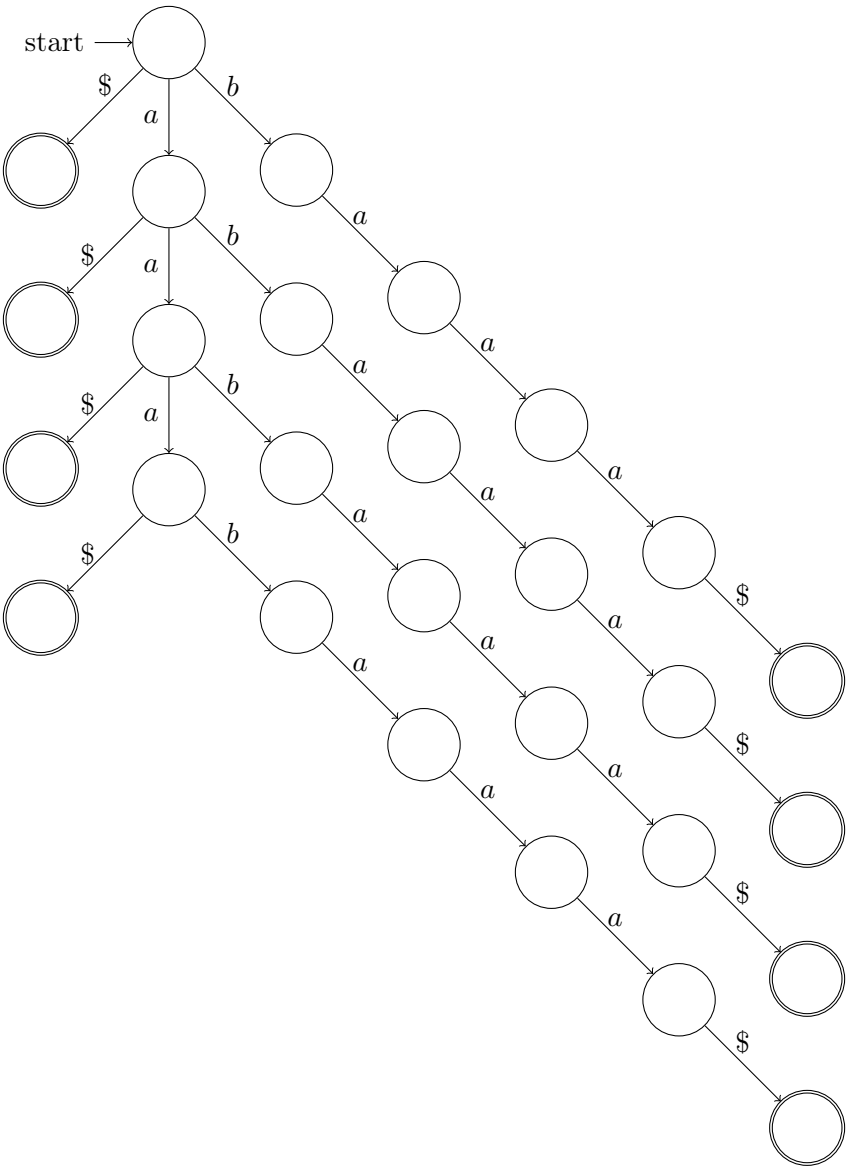


FIGURE 11 – Illustration d'un trie suffixe non compressé pour *aaabaaa*, de taille quadratique, cf Example 6.13

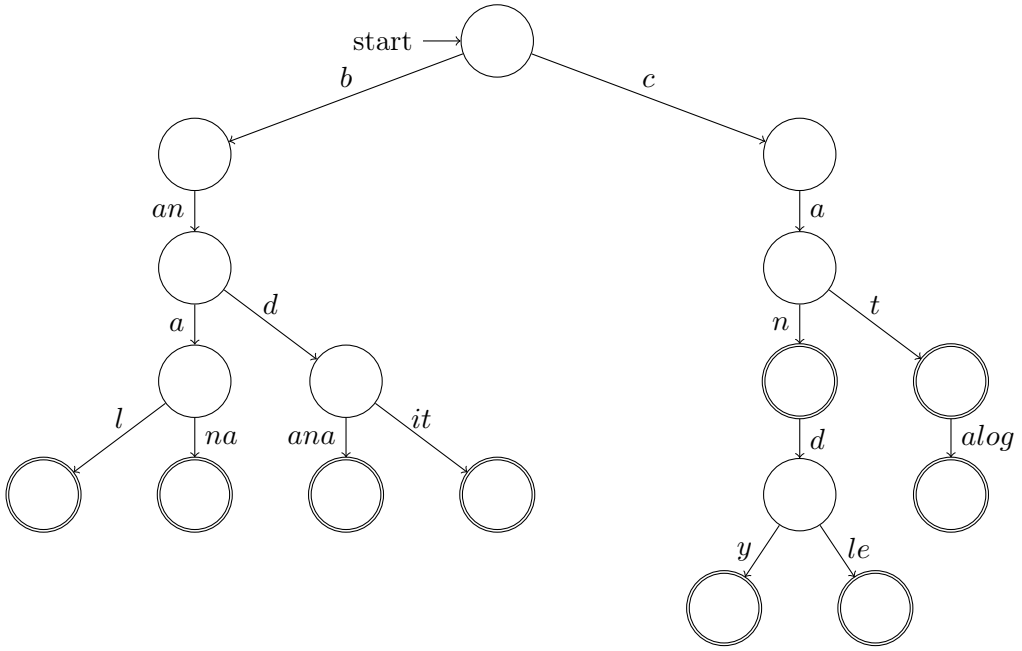


FIGURE 12 – Trie compressé correspondant à l'Exemple 6.12

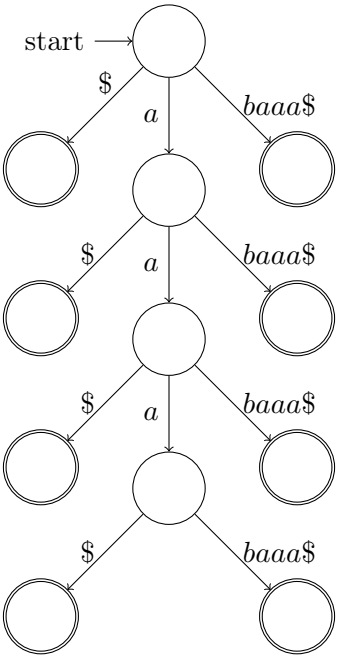


FIGURE 13 – Trie suffixe compressé pour *aaabaaa*, dont la version non compressée était en Figure 11. Les facteurs *baaa*\$ ne seraient pas écrits explicitement mais écrits avec une paire d'indices $3, 7$ correspondant à ce sous mot dans *aaabaaa*\$

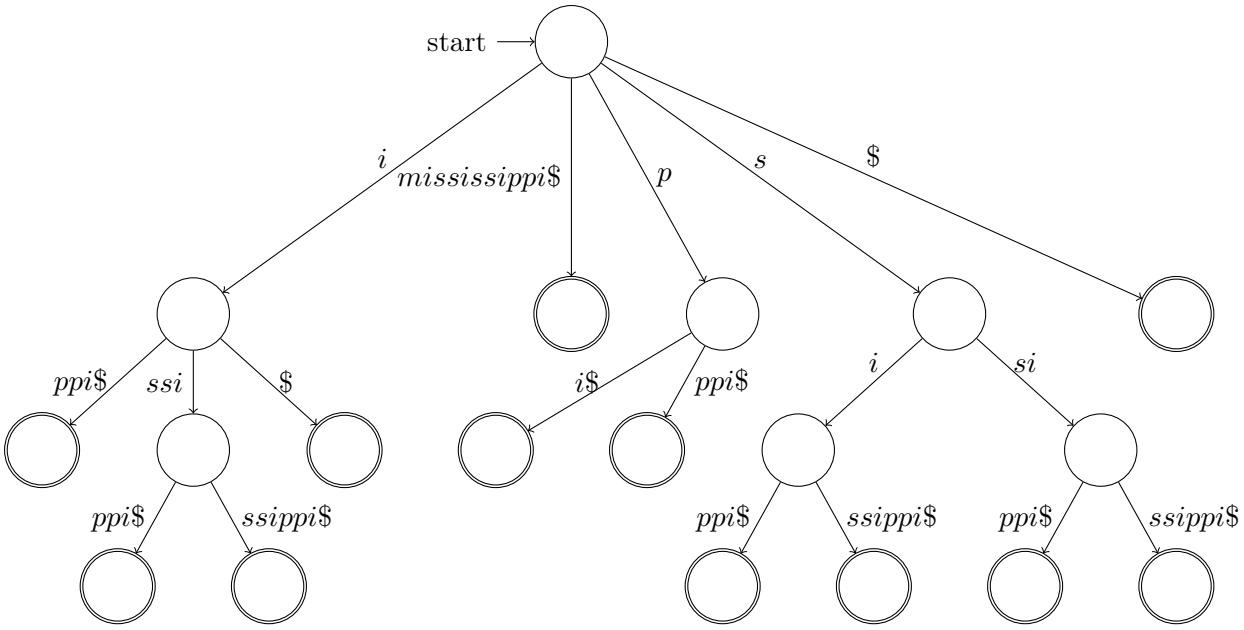


FIGURE 14 – Trie suffixe compressé pour *mississippi*\$, cf Exercice 6.15

Exercice 6.15. *Construire le trie suffixe compressé pour la chaîne misssissippi*\$.

Solution 6.16. *La solution est donnée en Figure 14.*

De manière générale, l'implémentation en ligne de Visualgo [5] permet de jouer avec des exemples et comprendre l'algorithme.

On peut maintenant décrire les deux questions qu'on va étudier : comment peut-on calculer un trie suffixe compressé ? et comment peut-on l'utiliser pour diverses tâches ? dans la suite on parlera simplement de *trie suffixe* en supposant implicitement que les tries suffixes sont compressés.

Construction d'un trie suffixe. La méthode naïve pour construire un trie suffixe compressé consiste à simplement insérer tous les suffixes les uns après les autres. On a expliqué pourquoi cette méthode fonctionnait en temps linéaire en chaque suffixe inséré, donc elle fonctionne en temps quadratique en le mot u, ce qui est plutôt défavorable.

Il se trouve que des algorithmes astucieux (par exemple, l'algorithme d'Ukkonen) permettent de construire un trie suffixe compressé en temps linéaire en le mot u. Cette construction est technique et sera donc admise :

Theorem 6.17 (admis)**.** *Étant donné un mot u, on peut construire en $O(|u|)$ un trie suffixe compressé pour u, c'est-à-dire un trie compressé qui représente tous les suffixes de u\$.*

Noter que ceci implique également que la taille d'un trie suffixe compressé est toujours linéaire en la taille du mot d'entrée (ce qui est clairement impossible à améliorer) : ce point n'a rien d'évident.

On remarque que, dans un trie suffixe (compressé ou non), les feuilles sont en bijection avec les suffixes (non stricts, potentiellement vides) du mot. On supposera toujours que chaque feuille est étiquetée par un entier indiquant à quel suffixe elle correspond : soit la longueur de ce suffixe, soit sa position de départ, ces informations étant équivalentes. Noter qu'on peut assurer cela en

5. `https://visualgo.net/en/suffixtree`

temps linéaire en parcourant le trie suffixe en retenant, pour chaque nœud, la longueur totale du facteur auquel il correspond (c'est la longueur totale des mots étiquetant les arêtes parcourues) : ceci nous donne, à chaque fois qu'on atteint une feuille, la longueur du suffixe à laquelle cette feuille correspond.

Utilisation d'un trie suffixe. Le principal intérêt d'un trie suffixe est qu'il permet de tester de façon efficace, en temps linéaire en un motif v, si v apparaît comme facteur de u. En effet, c'est le cas si et seulement si v est un préfixe d'un suffixe de u. Ainsi, on peut simplement explorer le trie suffixe compressé en cherchant v, et voir s'il apparaît ou non comme un nœud n de l'arbre (potentiellement comme un nœud interne implicite, c'est-à-dire au milieu d'une arête étiquetée par un mot de longueur supérieure à 1). Ensuite, pour trouver toutes les occurrences du facteur v, il suffit d'énumérer tous les suffixes de u qui commencent par v : les feuilles représentant ces suffixes sont précisément les feuilles descendantes du nœud n (ou, si n est un nœud implicite au milieu d'une arête, les feuilles descendantes du nœud cible de cet arête). On peut obtenir ces nœuds par un parcours depuis n, et ainsi identifier toutes les occurrences de v dans u (produites dans un ordre arbitraire).

À noter que la complexité de cette exploration est linéaire en le nombre de résultats à produire : on identifie en effet toutes les feuilles, et on visite également des nœuds internes, mais comme le trie suffixe est compressé on sait que tous les nœuds internes ont au moins deux enfants, ainsi pour N feuilles il y a au plus $N - 1$ nœuds internes et la complexité du parcours est bien en $O(N)$. Ainsi, pour trouver toutes les occurrences d'un motif v comme facteur d'un mot u à l'aide d'un trie suffixe, la complexité est bien $O(|u|)$ (pour la construction admise du trie suffixe compressé à partir de u, qui est indépendante de v) plus $O(|v|)$ (pour la recherche du nœud du trie, potentiellement implicite, qui correspond à v) plus $O(N)$ (pour la production des résultats). En cumulé comme $N = O(|u|)$ on retrouve une complexité $O(|u| + |v|)$ comme pour KMP ; mais on note bien que pour KMP on indexe d'abord le motif v pour le chercher ensuite dans un texte u arbitraire, alors qu'avec les tries suffixes on indexe d'abord le texte u pour y chercher ensuite un motif v arbitraire. L'une et l'autre de ces tâches peut être plus ou moins pertinente en fonction des applications.

On peut également résoudre d'autres problèmes avec un trie suffixe. Par exemple, le problème de trouver la *sous-chaîne répétée la plus longue* : trouver le plus long mot v (non nécessairement unique) qui apparaît comme facteur à deux endroits différents de u. Il s'agit simplement de parcourir le trie suffixe compressé pour trouver le nœud le plus profond (en termes de longueur totale des étiquettes du chemin depuis la racine) qui a au moins deux feuilles (or un précalcul linéaire parcourant l'arbre de bas en haut nous permet de savoir, pour chaque nœud du trie, le nombre de feuilles accessibles depuis ce nœud).

Si on ne souhaite pas autoriser les chevauchements, c'est-à-dire qu'on cherche un mot v de longueur maximale (là encore non nécessairement unique) de sorte qu'on peut écrire $u = rvsvt$ pour un certain choix de $r, s, t \in \Sigma^*$, on peut procéder comme suit. On précalcule d'abord de bas en haut, pour chaque nœud interne de l'arbre, la longueur minimale et maximale du suffixe accessible depuis ce nœud. Ainsi, étant donné un nœud n, pour u l'étiquette du chemin de la racine jusqu'à n, on sait quelle est la première et dernière occurrence de n. On peut ainsi savoir, pour chaque nœud, s'il y a deux occurrences qui ne se chevauchent pas, vu qu'il suffit de vérifier cela sur la première et dernière occurrence et que le fait qu'il y ait ou non chevauchement ne dépend que de la position des occurrences et de la longueur du facteur actuellement considéré. Ainsi, trouver le tel nœud le plus profond répond au problème.

On peut également énumérer efficacement les mots les plus courts qui ne sont *pas* facteurs de u, en faisant une exploration en largeur du trie suffixe compressé pour identifier les nœuds "manquants" les moins profonds.

On peut aussi résoudre efficacement des requêtes de type "longest common extension" (LCE) : étant donné deux positions dans u, quel est le mot le plus long permettant de poursuivre ces positions ? Là encore, on peut autoriser ou non les chevauchements. La tâche se formalise ainsi : si on écrit $u = xyz$, quel est le w le plus long tel qu'on a $yz = wy'$ et $z = wz'$ (version avec chevauchements), ou tel qu'on a $y = wy'$ et $z = wz'$ (version sans chevauchements). On répond à cela en calculant le plus petit ancêtre commun dans le trie suffixe compressé des deux nœuds qui correspondent aux suffixes y et yz interrogés : or il se trouve qu'on peut identifier le plus petit ancêtre commun en temps constant dans un arbre après un prétraitement linéaire. D'autres solutions pour le problème LCE sont possibles.

6.4 Conclusion

Nous avons présenté comment, étant donné deux mots u et v, on pouvait efficacement tester si u est un facteur de v : soit par indexation de u (algorithme KMP), soit par indexation de v (trie suffixe).

On peut, dans les deux cas, produire efficacement toutes les occurrences de u comme facteur de v.

On a généralisé ceci en TD 3 à l'algorithme d'Aho-Corasick (pour le cas où le motif n'est pas une chaîne u mais un ensemble fini de chaînes), et en TD 4 à l'énumération efficace des facteurs d'une chaîne qui sont acceptées par un automate donné (au prix d'une complexité plus élevée).

7 Monoïde de transition et applications

Dans cet section, nous définissons un objet mathématique à partir des automates déterministes : le *monoïde de transition*. Nous énonçons certaines de ses propriétés. Nous montrons enfin comment le monoïde de transition peut être utilisé pour donner des algorithmes efficaces pour deux tâches : la maintenance incrémentale de l'appartenance d'un mot à un langage, et le test rapide de si un facteur d'un mot appartient à un langage ou non. Ces deux tâches sur des chaînes de caractères peuvent être vues comme une variante de tâches étudiées à la section précédente. La deuxième tâche sera étudiée en TD ; on verra également un algorithme plus efficace pour cette tâche spécifiquement, qui ne repose pas sur le monoïde de transition.

7.1 Définitions

On rappelle qu'un *monoïde* $(S, \cdot, e)$ est un ensemble S qui est muni d'une loi de composition $\cdot : S \times S \to S$ supposée associative (c'est-à-dire que $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ pour tous $x, y, z \in S$) et d'un élément neutre $e \in S$ (c'est-à-dire que $e \cdot x = x \cdot e = e$ pour tout $x \in S$). Autrement dit un monoïde est comme un groupe mais où les éléments n'ont pas nécessairement d'inverses. Par exemple, tout groupe est en particulier un monoïde : mais on peut par exemple considérer $(\mathbb{N}, +, 0)$ ou $(\mathbb{N}, \times, 1)$ qui sont des exemples de monoïdes (en effet l'addition et la multiplication sont associatives et ont 0 et 1 respectivement comme éléments neutres) mais qui ne sont pas des groupes (il n'y a pas d'inverse à l'addition ou à la multiplication dans $\mathbb{N}$).

Suivant l'associativité on ignorera les parenthèses et on notera la composition multiplicativement c'est-à-dire xy dénote $x \cdot y$. Un monoïde peut être *commutatif*, c'est-à-dire $xy = yx$ pour tous $x, y \in S$, mais en général les monoïdes ne sont pas commutatifs.

Soit $A = (Q, \Sigma, q_0, F, \delta)$ un automate déterministe supposé complet. On rappelle qu'on peut étendre la fonction de transition $\delta : Q \times \Sigma \to Q$ en une fonction $\delta^* : Q \times \Sigma^* \to Q$ telle que, pour chaque $q \in Q$ et $w \in \Sigma^*$, la valeur de $\delta^*(q, w)$ est l'état auquel on parvient en lisant w depuis q. Formellement, on définit δ^* inductivement en posant $\delta^*(q, \epsilon) := q$ et $\delta^*(q, au) = \delta^*(\delta(q, a), u)$ pour chaque $a \in \Sigma$ et $u \in \Sigma^*$.

Le *monoïde de transition* se compose d'objets qui sont des fonctions de Q dans Q et qui sont associés à des mots de Σ^*. Plus précisément :

— Au mot vide ϵ, on associe la fonction identité sur Q ;
— À chaque lettre $a \in \Sigma$, on associe la fonction $\delta_a : Q \to Q$ définie comme suit : pour chaque $q \in Q$, on a $\delta_a(q) = \delta(q, a)$.
— À chaque mot $w \in \Sigma^*$, on associe la fonction $\delta_w : Q \to Q$ définie par $\delta_w(q) = \delta^*(q, w)$ pour chaque $q \in Q$.

En d'autres termes, pour chaque mot $w \in \Sigma^*$, l'élément du monoïde des transitions associé à w envoie chaque état $q \in Q$ à l'unique état auquel on aboutit en lisant w depuis q. On peut le voir graphiquement en regardant les chemins étiquetés par w dans l'automate, en notant qu'il y en a précisément un pour chaque état de départ car l'automate est déterministe complet.

Formellement, le monoïde de transition de A est défini comme suit :

Definition 7.1. *Le monoïde de transition de A est l'ensemble des fonctions f de Q dans Q telles que $f = \delta_w$ pour un certain $w \in \Sigma^*$. La loi de composition du monoïde de transition est la composition de fonctions. L'élément neutre de ce monoïde est la fonction identité.*

On explique ci-dessous comment calculer le monoïde de transition ; cette construction sera revue en TD. La présentation suit [13], p 83-84. On considère les mots dans l'ordre radix $<$: d'abord par longueur, puis par ordre lexicographique. Au fur et à mesure du processus on va maintenir deux choses :

— Une table stockant des éléments du monoïde de transition, et pour chaque élément un mot (le plus petit dans l'ordre radix) permettant d'obtenir cet élément. Ceci est commode à représenter sous la forme d'un tableau, dont les colonnes représentent les états de l'automate, et où chaque ligne correspond à un élément du monoïde. Pour chaque ligne, on inscrit dans la case de la colonne d'un état q l'image de q dans l'élément du monoïde de transition représenté par la ligne. On écrit également à gauche un mot permettant d'obtenir l'élément du monoïde.
— Des règles de récriture $u \to v$ où $v < u$, dans les cas où on constate que les mots u et v sont envoyés vers le même élément du monoïde de transition. On garantira que, dans ce cas, le mot v figure dans la table du point précédent.

Lorsque l'on considère un mot w, on vérifie d'abord s'il y a un facteur de w qui apparaît comme membre droit d'une règle de récriture $u \to v$. Si c'est le cas, alors on ignore w. En effet, on peut alors écrire $w = sut$, et on sait que $w = svt$ a déjà été considéré précédemment : en effet on vérifie aisément que l'ordre $<$ est compositionnel en cela que $u < v$ implique $sut < svt$. Du reste, comme u et v sont envoyés vers le même élément du monoïde de transition, on sait pas compositionnalité que sut est envoyé vers le même élément que svt, qui a déjà été considéré. Ainsi, il est inutile de considérer w. (Noter qu'il est également inutile de stocker la règle $w \to svt$: cela ne fait pas de mal, mais ce n'est pas nécessaire car tout mot ayant w comme facteur aura également v comme facteur.)

Si l'on considère un mot w qui ne comporte aucun facteur qui est membre gauche d'une règle de récriture (en particulier au début où aucune règle n'est stockée), alors il faut déterminer à quel élément du monoïde de transition il correspond. On peut le faire de diverses manières : soit directement sur l'automate, soit en écrivant $w = av$ et en composant l'image de a et l'image de v qui a été précédemment calculée (elle doit figurer dans la table, sinon v a un facteur qui est membre gauche d'une règle de récriture et on aurait ignoré w). On considère l'élément du monoïde de transition ainsi obtenu. De deux choses l'une : soit l'élément figure déjà dans la table, soit pas.

S'il figure déjà dans la table, soit u un mot précédent ayant permis de l'obtenir (ce mot est alors unique) : on stocke alors la règle de récriture $w \to u$ (et il est inutile de stocker l'élément du monoïde associé à w dans la table).

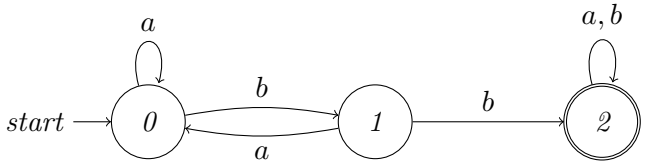
S'il est nouveau, alors on stocke le nouvel élément du monoïde dans la table et on l'associe à w (qui est le premier mot permettant de l'obtenir).

On arrête le calcul lorsqu'il est clair que tous les prochains mots auront un facteur qui soit membre droit d'une règle de transition. En particulier, si pour une certaine longueur aucun mot de cette longueur n'a conduit à l'ajout d'une ligne dans la table, alors on sait qu'on peut s'arrêter : les mots plus longs auront forcément un mot de la longueur précédente comme facteur et ils seront ignorés aussi (car ils contiennent comme facteur un mot de la longueur précédente, or tous ces mots sont soit membres gauches d'une règle de récriture soit contiennent un facteur qui l'est).

Cet algorithme est correct : il est clair que tout élément qu'il découvre est bien un élément du monoïde de transition, et à l'inverse on peut montrer il découvre tous les éléments. En effet, quand l'algorithme s'arrête, on sait alors que tout prochain mot w s'écrira $w = sut$ avec u un membre gauche de règle de récriture ainsi il donnera la même image qu'un mot précédemment considéré, comme on l'a déjà expliqué. Du reste, on est sûr que l'algorithme termine. En effet, le nombre d'éléments possibles du monoïde de transition est fini, donc le nombre de lignes ajoutées à la table est fini : au bout d'un moment on ne fait plus qu'ignorer des mots et ajouter des règles de récriture, donc il y a forcément une longueur pour laquelle aucun mot ne conduit à l'ajout d'une nouvelle ligne dans la table.

Illustrons ce calcul sur un exemple :

Exemple 7.2. *On considère le langage $\Sigma^* bb \Sigma^*$ sur $\Sigma = \{a, b\}$, et l'automate déterministe minimal pour ce langage :*



On commence par le mot vide ϵ. Celui-ci correspond toujours à la fonction identité. On obtient :

	0	1	2
ϵ	0	1	2

(Noter que l'état 2 est un puits, ainsi l'image de 2 sera toujours 2.)

On travaille avec l'ordre radix induit par l'ordre suivant sur Σ : $a < b$. On considère donc les mots de longueur 1 : d'abord a, puis b.

Pour a, on obtient l'image suivante :

	0	1	2
ϵ	0	1	2
a	0	0	2

Pour b, on obtient :

	0	1	2
ϵ	0	1	2
a	0	0	2
b	1	2	2

Passons aux mots de longueur 2. *On considère aa. (Bien sûr aa contient un facteur qui apparaît déjà dans la table, c'est-à-dire a, mais on le considère tout de même : on n'a pas encore trouvé de règle de récriture donc on doit tout considérer pour le moment.) On détermine sur l'automate que aa correspond au même élément que a : il envoie* 0 *vers* 0 *et* 1 *vers* 0 *et* 2 *vers* 2. *Ainsi on stocke la règle :* $aa \to a$.

On considère ab, on calcule l'image sur l'automate et on obtient un nouvel élément. On considère ensuite ba, on obtient aussi un nouvel élément. On considère enfin bb, on obtient aussi un nouvel élément. Ces trois étapes donnent :

	0	*1*	*2*
ϵ	*0*	*1*	*2*
a	*0*	*0*	*2*
b	*1*	*2*	*2*
ab	*1*	*1*	*2*
ba	*0*	*2*	*2*
bb	*2*	*2*	*2*

On remarque que bb est un zéro. Ainsi, on pourra accélérer le calcul par la suite en sachant que tout mot qui contient bb va se récrire à bb. Mais attention : il faut quand même considérer ces mots et ajouter une règle de récriture pour eux ! (Autrement dit, remarquer que bb est un zéro ne revient pas à considérer que bb est membre gauche d'une règle de récriture.)

On passe aux mots de longueur 3 :

- *Les mots aaa et aab contiennent le facteur aa qui est membre gauche de la règle* $aa \to a$ *donc on les ignore.*
- *Pour aba, on obtient la même image que a, donc* $aba \to a$.
- *Pour abb, on obtient la même image que bb (on le sait car bb est un zéro), donc* $abb \to bb$.
- *Pour baa on l'ignore à cause de la règle* $aa \to a$.
- *Pour bab on obtient la même image que b, donc* $bab \to b$.
- *Pour bba on obtient la même image que bb (là encore car c'est un zéro) donc* $bba \to bb$. *De même* $bbb \to bb$.

Aucun mot de longueur 3 n'a conduit à l'ajout d'une nouvelle ligne dans la table. Ainsi, tous les mots de longueur 4 ou plus contiennent un facteur qui est membre droit d'une règle de récriture. Ainsi, on s'arrête là. Les éléments du monoïde de transition sont donc ceux associés à ϵ, *a, b, ab, ba, et bb (ce dernier étant un zéro). On peut si on le souhaite construire la table de multiplication du monoïde pour matérialiser sa loi de composition.*

On dit que deux mots u et v sont *équivalents* pour A si on a $\delta_u = \delta_v$. Noter que le nombre de telles classes d'équivalence est manifestement fini, vu qu'il y en a au plus $|Q|^{|Q|}$.

On remarque également que le monoïde de transition n'est pas une propriété intrinsèque du langage, vu qu'il dépend de l'automate utilisé pour représenter le langage. Ainsi, si on a le choix, on préférera généralement prendre un automate aussi petit que possible pour le langage, en particulier on pourra vouloir prendre l'automate minimal du langage (cette notion est couverte en INF105).

On montrera certaines propriétés simples du monoïde de transition en séance de TD. Le monoïde de transition de l'automate minimal d'un langage est isomorphe à un autre monoïde pour le langage, dont la définition ne dépend plus cette fois de l'automate : on l'appelle alors le *monoïde syntaxique* du langage. L'étude des propriétés du monoïde syntaxique, et de ses liens avec le langage, est un sujet fécond de recherche depuis les années 50, qui est l'objet de la *théorie algébrique des automates*. Pour en illustrer la puissance, nous énoncerons sans preuve une caractérisation frappante d'une sous-classe des langages réguliers, les *langages sans étoile* :

Definition 7.3. *Une* expression régulière généralisée *est une expression régulière où on autorise également l'opérateur de complémentation, noté* $\overline{\cdot}$*. Par exemple* $\overline{\emptyset}$ *est une expression régulière généralisée qui dénote* Σ^**. Une* expression régulière généralisée sans étoile *est une expression régulière généralisée où on autorise les opérateurs d'union, de concaténation, et de complémentation, mais pas l'étoile de Kleene. Un* langage sans étoile *est un langage qui peut être décrit par une expression régulière généralisée sans étoile.*

Noter que les expressions régulières généralisées, et les expressions régulières généralisées sans étoile, peuvent librement utiliser le symbole d'intersection, puisque $e \cap e'$ est équivalent à $\overline{\overline{e} \cup \overline{e'}}$ par la loi de De Morgan.

Exemple 7.4. *Les langages finis sont sans étoile. Sur l'alphabet* $\Sigma = \{a, b\}$*, le langage* a^* *est sans étoile puisqu'on peut l'écrire comme le langage des mots qui ne contiennent pas de* b*, c'est-à-dire* $\overline{\Sigma^* b \Sigma^*}$*, ou plus précisément* $\overline{\overline{\emptyset} b \overline{\emptyset}}$ *pour éliminer toutes les étoiles.*

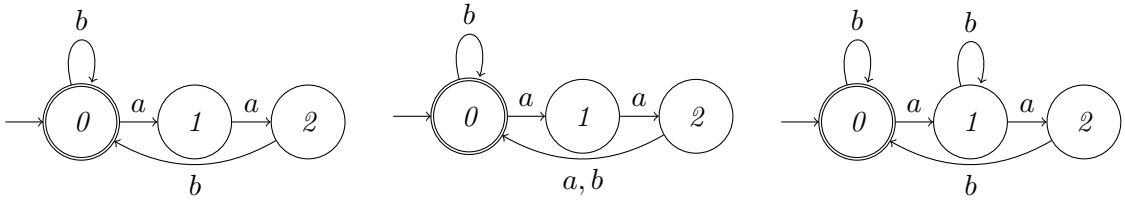
Le langage $(ab)^*$ *est sans étoile : c'est l'intersection des langages qui ne contiennent pas* aa*, qui ne contiennent pas* bb*, qui commencent par* a *et qui finissent par* b*. Autrement dit c'est l'intersection de* $\overline{\Sigma^* aa \Sigma^*}$*, de* $\overline{\Sigma^* bb \Sigma^*}$*, de* $a\Sigma^*$ *et de* $\Sigma^* b$*, où les* Σ^* *peuvent à nouveau être remplacés par des* $\overline{\emptyset}$*.*

On mentionne en passant le *problème de la hauteur d'étoile généralisée* : on ne sait pas si les expressions régulières généralisées sans étoiles imbriquées (ceci s'appelle de *hauteur d'étoile généralisée* égale à 1) suffisent à décrire tous les langages réguliers. Les expressions régulières généralisée de hauteur d'étoile généralisée zéro définissent précisément les langages sans étoile d'après ce qu'on vient de dire. (On note que, pour les expressions régulières non généralisées c'est-à-dire sans complémentation, il y a une hiérarchie stricte entre les langages réguliers qui peuvent être décrits par un certain nombre d'imbrications maximales d'étoile : ceci est une conséquence du théorème d'Eggan [3])

La notion de langage sans étoile semble assez ad hoc. Pourtant, c'est une notion très robuste : elle admet plusieurs caractérisations indépendantes. Voici des définitions a priori sans rapport.

Definition 7.5. *Soit* $A = (Q, \Sigma, q_0, F, \delta)$ *un automate fini déterministe complet. Pour* $q \in Q$ *et* $w \in \Sigma^*$*, on dit que* w *définit un* cycle *sur* q *si* $\delta^*(q, w) = q$*. On dit que* A *est* sans compteur *si, pour tout* $m \geq 1$*, tout* $w \in \Sigma^*$ *et tout* $q \in Q$*, si* w^m *définit un cycle sur* q *alors* w *définit aussi un cycle sur* q*.*

Exemple 7.6. *Considérons les trois automates suivants :*



Remarquer que les deux automates de droite sont une modification de l'automate de gauche obtenus à chaque fois en ajoutant une nouvelle transition.

L'automate de gauche est sans compteur. En effet, les mots qui définissent des cycles sur 0 *sont de la forme* $(aab + b)^*$ *: si un tel mot s'écrit comme* w^m *pour un certain* $m > 1$*, alors on peut montrer que* w *est également de la forme* $(aab+b)^*$*, en effet* aab *et* b *forment un code préfixe donc tout mot de* $(aab + b)^*$ *admet une unique décomposition de cette forme et si* w *finissait au milieu d'un mot* aab *alors il n'est pas possible de continuer avec une autre occurrence de* w*. Les mots qui définissent des cycles sur* 1 *sont de la forme* $ab(b + aab)^*a$ *: si un tel mot s'écrit comme* w^m *avec* $m > 1$ *alors comme* w *commence par* ab *la seule possibilité est de s'arrêter après le*

premier a d'un facteur aab et à ce moment l'automate est sur 1 donc w est aussi l'étiquette d'un cycle sur 1. Pour des cycles sur 2 c'est le de la forme $b(b+aab)^*aa$, si un tel mot s'écrit w^m avec $m > 1$ alors comme w termine par aa la seule possibilité est que l'avant-dernier w termine avant le dernier b d'un facteur aab et à ce moment l'automate est sur 2 et on conclut comme précédemment.

Le langage de l'automate peut s'écrire comme le langage des mots qui ne contiennent pas trois a consécutifs, qui ne contiennent pas le facteur bab, qui ne contiennent pas de facteur bab et qui ne finissent pas par un a : c'est un langage sans étoile.

L'automate du milieu n'est pas sans compteur. En effet, aaa définit un cycle sur 0 et $aaa = a^3$ mais a ne définit pas de tel cycle. Le langage correspondant, $(b + aab + aaa)^*$, n'est donc pas un langage sans étoile d'après le résultat admis. De fait son intersection avec le langage sans-étoile a^* est $(aaa)^*$, et les langages sans-étoile sont clos par intersection, donc le fait important est que le langage $(aaa)^*$ n'est pas sans étoile, ce que l'on peut démontrer. (Noter que son monoïde syntaxique n'est clairement pas apériodique.)

L'automate de droite n'est pas sans compteur. En effet, $abab$ définit un cycle sur 0 (noter que ce n'est pas un cycle simple) alors que ab ne définit pas de tel cycle. Ceci implique que le langage correspondant, $(b + (ab^*ab))^*$, n'est pas sans étoile. De fait son intersection avec le langage sans-étoile $(ab)^*$ est $(abab)^*$ dont on peut montrer qu'il n'est pas sans étoile et dont le monoïde syntaxique n'est manifestement pas apériodique.

Definition 7.7. *Un monoïde fini $(S, \cdot, e)$ est* apériodique *si, pour chaque $x \in S$, il y a un entier $n > 0$ tel que $x^n = x^{n+1}$.*

Le lien entre ces trois définitions est fourni par un théorème de Schützenberger et de McNaughton et Papert :

Theorem 7.8 (admis)**.** *Soit L un langage régulier. Les conditions suivantes sont équivalentes :*
- *L est un langage sans étoile, c'est-à-dire représentable par une expression régulière généralisée sans étoile ;*
- *L est accepté par un automate fini déterministe complet qui est sans compteur (et dans ce cas l'automate minimal convient toujours) ;*
- *Le monoïde syntaxique de L (c'est-à-dire le monoïde de transition de l'automate minimal reconnaissant L) est apériodique.*

D'autres caractérisations équivalentes sont possibles et ne sont pas définies ici, par exemple un langage est sans étoile si et seulement si tout facteur est itérable à partir d'un certain rang, ou bien si et seulement s'il est définissable par une formule de la logique du premier ordre.

7.2 Application : Maintenance incrémentale de l'appartenance

Nous allons illustrer l'utilité du monoïde de transition d'un langage pour un problème appelé la *maintenance incrémentale de l'appartenance d'un mot à un langage régulier*. On fixe un langage régulier L dans cette section, dont la taille sera supposé constante dans l'étude de la complexité asymptotique. Étant donné un mot w, on souhaite savoir si le mot w appartient au langage L, et maintenir cette information efficacement sous des *mises à jour* qui modifient w, sans relire w en entier à chaque fois. On s'intéressera ici au cas des mises à jour de *substitution* : étant donné une position $1 \leq i \leq |w|$ et une lettre $a \in \Sigma$, l'effet de la mise à jour de substitution (i, a) est de remplacer la i-ème lettre du mot w par a.

Exemple 7.9. *Pour le mot $w = abaa$, l'application de la mise à jour de substitution $(2, a)$ donne le mot $w = aaaa$. L'application de la mise à jour de substitution $(4, a)$ est sans effet. On remarque que la longueur du mot n'est jamais modifiée par les mises à jour de substitution.*

L'algorithme naïf pour le problème est le suivant : pour chaque mise à jour de substitution, on l'applique au mot (stocké par exemple dans un tableau), puis on lit le mot actuel et on vérifie s'il est accepté ou non par l'automate, en temps $O(|w|)$ à chaque mise à jour. On démontre dans cette section le fait suivant : après un précalcul en temps $O(|w|)$ sur le mot initial, on peut supporter chaque mise à jour en temps $O(\log |w|)$ et savoir si le mot actuel appartient ou non au langage. En fait, on va même maintenir l'information de l'élément du monoïde syntaxique réalisé par le mot courant. Pour être précis, on se fixe l'automate minimal $A = (Q, \Sigma, q_0, F, \delta)$ reconnaissant L, et on maintient l'élément δ_w du monoïde de transition de A réalisé par le mot courant w. Ceci nous indique en particulier si $w \in L$ ou $w \notin L$, en testant si $\delta_w(q_0)$ est final ou non.

Pour ce faire, on va d'abord chercher à simplifier l'exposition en supposant que le mot d'entrée a pour longueur une puissance de 2, mettons 2^l. C'est-à-dire qu'étant donné le mot de départ w, on le remplace par $w' := w\bullet^i$ pour un certain i de sorte que $|w\bullet^i|$ soit une puissance de 2, où $\bullet$ est un nouveau caractère. Cette opération peut être réalisée en temps linéaire, vu qu'au plus elle double la taille du mot.

Ensuite, on bâtit un arbre binaire complet T de hauteur l, et on met en correspondance les 2^l feuilles de T avec les lettres de w' : c'est-à-dire que la i-ème feuille de T dans l'ordre de parcours préfixe correspond à la i-ème lettre de w'. On identifie plus généralement chaque nœud n de T avec l'intervalle ι_n formé des feuilles accessibles depuis n. Formellement les intervalles ι_n sont un intervalle de positions de w, c'est-à-dire un couple $[i, j[$ avec $1 \leq i \leq |w'|+1$. Pour la i-ème feuille n_i dans le parcours préfixe, son intervalle est $\iota_{n_i} = [i, i+1[$. On définit ensuite inductivement de bas en haut les intervalles de chaque nœud : pour un nœud interne n ayant pour enfants n_1 et n_2, un invariant inductif montrera que les intervalles ι_{n_1} et ι_{n_2} sont contigus, c'est-à-dire $\iota_{n_1} = [l, m[$ et $\iota_{n_2} = [m, r[$, et on a alors $\iota_n = [l, r[$ c'est-à-dire la concaténation de ι_{n_1} et ι_{n_2}. Noter que la racine a pour intervalle $[1, |w'|+1[$ qui représente le mot entier. Du reste, si n est une feuille alors ι_n est un intervalle de longueur 1, et si n est à profondeur $1 \leq i \leq l$ alors ι_n est un intervalle de longueur 2^{l-i+1}.

On calcule ensuite de bas en haut l'information suivante : pour chaque nœud n, en prenant ι_n son intervalle et w'_{ι_n} le facteur de w' aux positions de l'intervalle ι_n, on veut stocker dans n un élément noté τ_n du monoïde de transition de A, à savoir l'élément $\delta_{w'_{\iota_n}}$ que réalise w. On calcule les τ_n pour tous les nœuds de T de bas en haut, en temps linéaire en T à langage L et automate A fixé, donc en temps $O(|w|)$, avec la définition suivante :

- Base 1 : pour une feuille n étiquetée $a \in \Sigma$, on a $\tau_n := \delta_a$, où δ_a est le cas de base pour la lettre a dans la définition des éléments du monoïde de transition. On rappelle que δ_a est la fonction définie par $\delta_a := \delta(q, a)$ pour chaque $q \in Q$.
- Base 2 : pour une feuille n étiquetée $\bullet$, on a $\tau_n := \mathrm{id}$, où id est la fonction identité sur Q. Noter que c'est également δ_ϵ, l'élément du monoïde de transition obtenu pour le mot vide ϵ.
- Induction : pour un nœud interne n ayant pour enfants n_1 et n_2, avec τ_{n_1} et τ_{n_2} calculés par induction, on définit $\tau_n = \tau_{n_2} \circ \tau_{n_1}$, c'est-à-dire la fonction qui envoie chaque état $q \in Q$ vers $\tau_{n_2}(\tau_{n_1}(q))$.

Noter bien que, si l'on peut stocker pour chaque nœud n son intervalle ι_n explicitement et que l'on stocke également l'élément τ_n, il ne faut surtout pas écrire explicitement les facteurs $w_{\iota(n)}$: ce serait trop coûteux.

Ce calcul est en temps linéaire en T à langage L et automate A fixé. On remarque par une induction immédiate que, pour chaque nœud n de T, l'élément τ_n du monoïde de transition est bien celui que réalise le facteur $w_{\iota(n)'}$. En particulier on sait que τ_r pour r la racine de T est l'élément réalisé par w' entier (en interprétant les symboles $\bullet$ comme n'ayant aucun effet), c'est-à-dire par w entier. On a donc $w \in L$ si et seulement si $\tau_r(q_0) \in F$.

L'intérêt de cette construction est que, quand w subit une mise à jour par substitution (i, a),

on peut la refléter facilement dans T en temps proportionnel à la hauteur de T. Noter que la structure de T, ainsi que les valeurs de ϕ, ne changent pas. La mise à jour change en revanche les valeurs de w'_n précisément pour les nœuds sur le chemin C de la racine r à la i-ème feuille. (Noter que les feuilles numérotées j pour $j > |w|$, correspondant aux $\bullet$ rajoutés dans w' relativement à w, ne subiront jamais de mise à jour.) Ainsi, la mise à jour invalide les τ_n précisément pour n sur le chemin C. Mais on peut recalculer ces valeurs de bas en haut en suivant ce chemin, et obtenir ainsi l'arbre T avec les valeurs de τ correctes suite à cette mise à jour. En particulier, τ_r nous indique si le mot après mise à jour appartient ou non au langage. À langage L et automate A fixé, ce recalcul est linéaire en C dont la cardinalité est la hauteur l de T, or $l = \log_2(|w'|)$ donc est $O(|w|)$. On a donc bien la complexité attendue.

7.3 Application : Requêtes d'appartenance de facteurs

Une autre application du monoïde de transition est de permettre d'indexer rapidement un mot, par une variante de la construction précédente, afin de répondre en temps logarithmique aux requêtes d'appartenance de facteurs : étant donné les coordonnées d'un facteur du mot, déterminer si le facteur correspondant est dans le langage ou non. Cette construction sera vue en TD. On note qu'elle supporte également les mises à jour de substitution d'une manière analogue à la section précédente. En fait, elle généralise la section précédente, vu que l'appartenance du mot entier au langage correspond à un cas particulier de requête d'appartenance de facteur, pour le facteur englobant la totalité du mot.

8 Automates d'arbre

Dans cette section, on commence à présent à généraliser notre étude de la théorie des langages pour s'éloigner du cas des mots et aller vers le cadre plus général de l'étude des *données structurées*. Ce terme vague désigne des données dont la forme est très contrainte, mais qui va au-delà des simples mots. On parlera dans cette section de données *arborescentes*, c'est-à-dire d'arbres et d'automates lisant des arbres : ceci permet de définir une classe de langages appelés *langages réguliers d'arbres*.

L'intérêt des automates d'arbres, d'un point de vue théorique, est qu'il s'agit d'une généralisation naturelle des langages réguliers. D'un point de vue plus appliqué, les automates d'arbres permettent de formaliser certaines notions concernant les arbres en informatique. Ils permettent par exemple de spécifier des contraintes structurelles que doivent satisfaire des documents, ce qui pourrait s'avérer utile pour spécifier des formats en XML ou en JSON. Ils permettent aussi d'abstraire certains types de calculs que l'on peut vouloir mener sur des arbres, notamment ceux qui se prêtent bien à un calcul récursif (correspondant aussi à de la programmation dynamique). Enfin, ils ont des liens avec les grammaires hors-contexte qui ont été étudiées en INF105, mais en s'intéressant aux arbres (correspondant peu ou prou aux arbres de dérivation) pour eux-mêmes.

L'objectif pour MITRO210 est simplement de présenter la définition des automates d'arbres (et le modèle d'arbre considéré), dans différentes variantes. On verra ensuite dans la section suivante comment la notion de *largeur arborescente* permet de généraliser notre étude en s'appliquant à des graphes généraux qui peuvent être représentés sous forme arborescente.

8.1 Définition des arbres

On fixe un alphabet fini Σ qui correspond cette fois aux étiquettes admises sur les nœuds des arbres.

Definition 8.1. *Un* Σ-arbre *est une structure définie récursivement comme suit :*

— *Un arbre singleton, consistant d'un seul nœud portant une étiquette de* Σ*, est un* Σ*-arbre*
— *Un arbre non-singleton, consistant d'un nœud dit* racine *portant une étiquette de* Σ*, et de deux enfants* T_1 *et* T_2 *appelés* enfant gauche *et* enfant droit *qui sont eux-mêmes des* Σ*-arbres*

Les Σ-arbres correspondent manifestement à des arbres dont les nœuds portent tous une étiquette de Σ. Ce sont des arbres enracinés et où chaque nœud est soit une feuille soit a exactement deux enfants ordonnés. On peut discuter de l'impact de ces choix :

— Le choix d'enraciner les arbres n'est pas très important : si on souhaite travailler sur des arbres non enracinés (autrement dit des graphes connexes acycliques), on peut généralement choisir arbitrairement une racine et se ramener au cas des arbres ordonnés.
— On impose que les arbres soient *binaires*, c'est-à-dire que chaque nœud a au plus deux enfants. On aurait pu à la place considérer des arbres d'arité non-bornée, où le nombre d'enfants de chaque nœud n'est pas limité. On peut ramener ce formalisme au formalisme des arbres binaires par la représentation *enfant-gauche-adelphe-droit*, ou (moins inclusivement) *fils-gauche-frère-droit*, ou en anglais *left-child-right-sibling* : chaque nœud admet pour enfant gauche son premier enfant (s'il existe) et pour enfant droit le nœud qui est successeur parmi les enfants de son parent (s'il existe). Cette transformation n'aura pas trop d'impact au sens où des notions d'automates d'arbres sur des arbres d'arité non bornée peuvent se ramener au cas des arbres binaires sans effet notable sur la classe des langages reconnaissables obtenus.
— On travaille ici sur des arbres, mais à condition de ne pas se restreindre au cas binaire on pourrait travailler aussi sur des forêts (ordonnés), là encore ce choix n'est pas très important.
— On impose aussi que les nœuds aient soit deux enfants soit aucun. Ce choix est essentiellement sans perte de généralité : on peut l'impose quitte à compléter les enfants manquants avec des nœuds annotés par un symbole spécial.
— On distingue les enfants gauche et droit. Dans le cas d'arbres binaires, c'est sans perte de généralité : quitte à étendre l'alphabet, on peut annoter chaque nœud par l'information de s'il est l'enfant gauche ou l'enfant droit de son parent. Pour les arbres d'arité non-bornée, le choix d'ordonner les enfants est plus engageant, mais c'est également le choix le plus courant.
— Bien sûr les arbres ne sont pas forcément complets et pas forcément équilibrés.

Un *langage d'arbres sur l'alphabet* Σ est simplement un ensemble de Σ-arbres. On va définir les *automates d'arbres* comme un formalisme permettant de définir des sous-ensembles des Σ-arbres satisfaisant certaines conditions d'intérêt.

8.2 Automates d'arbres

Les automates d'arbres peuvent se définir en deux variantes : de *bas en haut*, et de *haut en bas*. Ils peuvent également se définir comme des automates déterministes, nondéterministes, ou inambigus. Nous nous concentrons sur la variante de bas en haut qui est la plus commune, et mentionnons brièvement à la fin le cas des automates de haut en bas.

Definition 8.2. *Soit* Σ *un alphabet. Un* automate d'arbres déterministe complet de bas en haut *(DbTA) est un quintuplet* $A = (Q, \Sigma, \iota, F, \delta)$ *où* Q *est un ensemble fini d'états,* Σ *est l'alphabet,* $\iota \colon \Sigma \to Q$ *est une fonction appelée* fonction initiale*, et* $\delta \colon Q \times Q \times \Sigma \to Q$ *est la* fonction de transition*.*

Pour expliquer brièvement les différences avec les automates de mots (DFA) :

— On a une *fonction initiale* plutôt qu'un état initial, car le choix de l'état initial va s'effectuer en chaque feuille en fonction de l'étiquette de cette feuille. Dans le cas d'un automate de mots, cela reviendrait à dire que l'état initial est choisi en fonction de la première lettre.
— La fonction de transition prend maintenant un argument de $Q \times Q \times \Sigma$ au lieu de $Q \times \Sigma$. Là où la fonction de transition d'un automate de mots indique quel état est obtenu en lisant une certaine lettre à partir d'un certain état, celle d'un automate d'arbres indique quel état est obtenu sur un nœud interne portant une certaine étiquette, en fonction des *deux* états atteints sur les deux enfants.

Un exemple s'impose :

Exemple 8.3. *Considérons* $\Sigma = \{a, b\}$.

Posons $A = (Q, \Sigma, \iota, F, \delta)$ *l'automate défini comme suit :*

— $Q = \{0, 1\}$
— ι *envoie* a *vers* 1 *et* b *vers* 0
— $F = \{0\}$, *c'est-à-dire que seul l'état* 0 *est final*
— δ *est définie comme suit :*
  — *pour chaque* $q, q' \in Q$, *on a* $\delta(q, q', a) = q + q' + 1 \mod 2$
  — *pour chaque* $q, q' \in Q$, *on a* $\delta(q, q', b) = q + q' \mod 2$

Le rôle de cet automate est simplement de compter les nœuds étiquetés a modulo 2, et d'accepter si le nombre de tels nœuds est pair. On explique à présent informellement comment se déroule l'évaluation. L'automate associe à chaque nœud n *une valeur de* $\{0, 1\}$, *représentée par l'état. Intuitivement, il s'agit du compte modulo 2 du nombre de nœuds* a *accessibles depuis* n. *Le cas de base de la preuve inductive de cet invariant correspond aux valeurs de* ι, *et l'étape d'induction correspond aux valeurs de* δ.

À noter que, contrairement aux automates de mots, il n'est pas très courant d'utiliser une représentation graphique des automates d'arbre : en effet chaque transition ne va pas d'un état à un autre mais d'un couple d'états vers un état, ce qui rend le dessin difficile.

Pour définir formellement l'acceptation des automates d'arbres DbTA, on définit la notion de *run* :

Definition 8.4. *Soit* Σ *un alphabet,* $A = (Q, \Sigma, \iota, F, \delta)$ *un* Σ*-DbTA, et* T *un* Σ*-arbre. Le* run *de* A *sur* T *est une fonction* $\rho \colon T \to Q$ *est une fonction qui associe à chaque nœud* n *de* T *un état* $\rho(n)$ *de* Q. *Le run est défini inductivement de la manière suivante :*

— *Pour chaque feuille* n, *en notant* $a \in \Sigma$ *l'étiquette portée par* n, *la valeur du run en* n *est donnée par la fonction initiale : on a* $\rho(n) := \iota(a)$
— *Pour chaque nœud interne* n, *en notant* n_1 *et* n_2 *ses enfants gauche et droit respectivement et* a *son étiquette, la valeur du run en* n *est donnée par la fonction de transition en fonction de l'étiquette* a *et de la valeur du run sur les deux enfants : On a* $\rho(n) := \delta(\rho(n_1), \rho(n_2), a)$.

L'arbre T *est dit* accepté *par* A *si, en notant* ρ *le run et* n_0 *la racine de* T, *on a* $\rho(n_0) = F$; *l'arbre est* rejeté *dans le cas contraire.*

Si l'on contraste avec la Section 2.3, un run n'est plus défini simplement comme une séquence d'états mais comme un étiquetage de l'arbre d'entrée avec des états.

Un langage d'arbre est dit *régulier* s'il y a un automate qui le reconnaît. Par analogie avec les langages réguliers, on peut définir une syntaxe d'expressions rationnelles qui soit équivalente aux automates d'arbres c'est-à-dire qui définisse précisément les langages réguliers d'arbres, mais on n'en verra pas le détail ici.

On peut bien sûr changer légèrement la définition des DbTA en n'imposant plus que la fonction de transition ou la fonction d'initialisation soient une fonction totale, et en interprétant les cas où l'une de ces fonctions n'est pas définie comme conduisant immédiatement au rejet de l'arbre ;

comme pour un automate de mots. Si l'on a un automate de ce type, on peut également le *compléter* en ajoutant un état puits pour les transitions non définies afin d'obtenir un DbTA correspondant aux définitions données ci-dessus : dans ce cas il faut compléter ι de la façon attendue et compléter δ par $\delta(q, q', a) := q_p$, où q_p est le puits, pour tous $q, q' \in Q$ et $a \in \Sigma$, dès lors que *l'un* des états q et q' vaut q_p.

On donne un autre exemple d'automate d'arbre, cette fois en ne définissant pas toutes les transitions :

Exemple 8.5. *On pose $\Sigma = \{a\}$, c'est-à-dire que les Σ-arbres ont tous leurs nœuds étiquetés par a et ainsi l'étiquette n'est pas importante. On considère le Σ-DbTA $A = (Q, \Sigma, \iota, F, \delta)$ défini comme suit :*

- *$Q = \{f, i\}$ correspondant intuitivement aux feuilles et aux nœuds internes*
- *$\iota(a) := f$*
- *$F = Q$: tous les états sont finaux (ceci n'a d'intérêt que si certaines transitions ne sont pas définies)*
- *δ est défini de la façon suivante : pour tout triplet (q, q', a) avec $q, q' \in Q$, la valeur $\delta(q, q', a)$ n'est définie que si $q = f$, auquel cas on pose $\delta(q, q', a) = i$*

L'automate A accepte les arbres qui sont des peignes droits, c'est-à-dire que l'enfant gauche de chaque nœud interne est une feuille. En effet, sur un Σ-arbre T, le run de A sur T donnera l'état f à chaque feuille, et sur chaque nœud interne de bas en haut la transition ne sera définie que si l'enfant gauche du nœud a l'état f (c'est une feuille), auquel cas le nœud interne aura l'état i (et donc pas f). On accepte si et seulement si le run peut être complètement défini c'est-à-dire qu'il n'y a aucun nœud interne où le premier enfant ne soit pas une feuille.

Il y a bien sûr une correspondance entre les automates de mots et les automates finis. La façon la plus simple de le faire est de prendre des arbres peigne : on va prendre des arbres peignes droits, mais évidemment on pourrait faire le choix symétrique.

Definition 8.6. *Soit Σ un alphabet, et fixons un symbole $\perp$ n'appartenant pas à Σ. On pose $\Sigma' := \Sigma \sqcup \{\perp\}$. À tout mot $w = a_1 \cdots a_n$ de Σ^*, on associe le Σ'-arbre peigne droit T_w obtenu de la façon suivante : les nœuds internes de bas en haut sont étiquetés par les lettres $a_1, \ldots, a_n$, et les feuilles sont étiquetées par $\perp$. En particulier si $w = \epsilon$ on obtient l'arbre singleton avec un unique nœud étiqueté $\perp$.*

On prétend ensuite qu'une conversion entre les automates de mots et les automates d'arbre est possible, de manière assez simple, en ne s'intéressant qu'au comportement des automates d'arbre sur les arbres peigne.

Proposition 8.7. *Posons Σ un alphabet et $\Sigma' := \Sigma \sqcup \{\perp\}$ défini comme en Définition 8.6. Pour tout DFA A sur Σ, on peut obtenir un DbTA A' sur Σ' avec la propriété suivante : pour tout mot $w \in \Sigma^*$, si on pose T_w le Σ'-arbre correspondant, alors w est accepté par A si et seulement si T_w est accepté par A'.*

À l'inverse, pour tout DbTA A' sur Σ', on peut calculer un DFA A sur Σ avec la propriété suivante : pour tout Σ'-arbre T qui est un peigne droit dont les feuilles sont étiquetées par $\perp$ et les nœuds internes par Σ, si l'on définit par w_T l'unique mot tel que $T_{w_T} = T$, alors T est accepté par A' si et seulement si w_T est accepté par A.

Démonstration. On prouve d'abord la première direction. Soit $A = (Q, \Sigma, q_0, F, \delta)$ un DFA. On prend $q_\perp$ un nouvel état n'appartenant pas à Q. On pose $A' = (Q \cup \{q_\perp\}, \Sigma, \iota, F', \delta')$ où :

- $\iota(\perp) := q_\perp$ (et les autres valeurs de ι ne sont pas définies),
- $F' := F$ si A n'accepte pas le mot vide (c'est-à-dire $q_0 \notin F$) et $F' := F \sqcup \{q_\perp\}$ sinon,
- $\delta'(q, q_\perp, a) := \delta(q_0, a)$ si $a \in \Sigma$ et $q = q_\perp$

— $\delta'(q, q', a) := \delta(q', a)$ pour $q' \in Q$ si $a \in \Sigma$ et $q = q_\perp$
— $\delta'(q, q', a)$ n'est pas défini pour $q \neq q_\perp$ ou pour $a = \perp$

On montre alors par une induction immédiate que l'automate obtenu est correct. Le cas du mot vide est correctement traité par définition : A' accepte l'arbre singleton $\perp$ si et seulement si A accepte le mot vide. Il est manifeste que, comme dans l'Exemple 8.5, l'automate A' a un run défini sur un Σ'-arbre si et seulement si c'est un peigne droit où les feuilles toutes sont étiquetées par $\perp$ et les nœuds internes sont tous étiquetés par des lettres de Σ. Ensuite, pour tout mot non-vide $w = a_1 \cdots a_m$ avec $m \geq 1$, si l'on considère T_w et les nœuds internes $n_1, \ldots, n_m$ de bas en haut et ρ le run de A' sur T_w (qui est défini d'après ce qu'on vient de dire), on prétend que, pour chaque $1 \leq i \leq n$, l'état atteint par l'automate A après avoir lu le préfixe $a_1 \cdots a_i$ est précisément $\rho(n_i)$. Le cas de base pour $i = 1$ revient à constater que l'automate A atteint l'état $\delta(q_0, a_1)$ après avoir lu a_1, or $\rho(n_1) = \delta'(q_\perp, q_\perp, a_1) = \delta(q_0, a)$. Pour l'induction, après la lecture de $a_1 \cdots a_{i+1}$, l'automate A atteint l'état $\delta(q_i, a_{i+1})$ où q_i est l'état atteint après la lecture de $a_1 \cdots a_i$: par induction on a $q_i = \rho(n_i)$, et on a $\rho(n_{i+1}) = \delta'(q_\perp, \rho(n_i), a_{i+1}) = \delta(\rho(n_i), a_{i+1})$ comme attendu.

Pour la seconde direction, prenons $A' = (Q', \Sigma, \iota, F', \delta')$ un DbTA. Soit $q_\perp := \iota(\perp)$. On définit un DTA $A = (Q, \Sigma, q_\perp, F, \delta)$, en posant $\delta(q, a) := \delta'(q_\perp, q, a)$ pour tout $q \in Q$. On montre que A' est correct à nouveau par induction. Appelons *peigne bien formé* un Σ'-arbre qui est un peigne droit où les nœuds internes portent tous des étiquettes de Σ et où les feuilles portent toutes l'étiquette $\perp$. Pour le peigne bien formé T constitué d'un unique nœud, on a $w_T = \epsilon$, or A' accepte T si et seulement si $q_\perp \in F$, donc si et seulement si A accepte ϵ. Considérons à présent les peignes bien formés qui ne sont pas des singletons, et raisonnons à nouveau par induction. Sur un peigne bien formé T, si on considère les nœuds internes de bas en haut $n_1, \ldots, n_m$, et qu'on pose $a_1, \ldots, a_m$ leurs étiquettes respectives, alors on a $w_T = a_1 \cdots a_m$. Or, les états successifs $q_1, \ldots, q_m$ associés aux nœuds $n_1, \ldots, n_m$ par le run de A' sur T (tant qu'il est défini) sont manifestement $q_1 = \delta'(n_\perp, n_\perp, a_1)$, puis $q_2 = \delta'(n_\perp, q_1, a_2)$, et ainsi de suite. Il est alors immédiat que la sémantique de A est correcte. □

On définit à présent les variantes non-déterministes des automates d'arbre de bas en haut de la façon attendue :

Definition 8.8. *Soit Σ un alphabet. Un* automate d'arbres non-déterministe de bas en haut *(NbTA) est un quintuplet $A = (Q, \Sigma, \iota, F, \delta)$ où Q est un ensemble fini d'états, Σ est l'alphabet, $\iota \subseteq \Sigma \times Q$ est une relation appelée* relation initiale*, et $\delta \subseteq Q \times Q \times \Sigma \times Q$ est la* relation de transition*.*

Comme d'habitude, un automate d'arbres déterministe (DbTA) $A = (Q, \Sigma, \iota, F, \delta)$ est un cas particulier d'automate d'arbres nondéterministe en posant les relations $\iota' = \{(a, \iota(a)) \mid a \in \Sigma\}$ et $\delta' = \{(q_1, q_2, a, \delta(q_1, q_2, a)) \mid q_1, q_2 \in Q, a \in \Sigma\}$. Ces définitions supposent que A est complet, mais des définitions analogues sont possibles si A n'est pas complet simplement en excluant les tuples pour lesquels ι et δ ne sont pas définis.

Le run d'un automate sur un arbre n'est plus défini de façon unique lorsque l'automate n'est pas déterministe. Voici l'analogue de la Définition 8.4 :

Definition 8.9. *Soit Σ un automate, T un Σ-arbre, et $A = (Q, \Sigma, \iota, F, \delta)$ un Σ-NbTA. Un* run *de A sur T est une fonction ρ des nœuds de T dans Q qui satisfasse les conditions suivantes :*

— *Pour chaque feuille n de T, en posant $q := \rho(n)$ et $a \in \Sigma$ l'étiquette de n, on a $(q, a) \in \iota$*
— *Pour chaque nœud interne n de T ayant pour enfants gauche et droit n_1 et n_2 respectivement, en posant $q := \rho(n)$, $q_1 := \rho(n_1)$ et $q_2 := \rho(n_2)$, et en posant $a \in \Sigma$ l'étiquette de n, on a $(q_1, q_2, a, q) \in \delta$.*

Le run est acceptant *si la racine de T est envoyée vers un état de F, et* rejetant *dans le cas contraire. L'automate A* accepte *l'arbre T s'il existe un run acceptant de A sur T, sinon on dit que A* rejette *T. Le* langage *de A est le langage des arbres acceptés par A.*

À noter qu'il est possible qu'aucun run d'un automate A n'existe sur un certain arbre T, auquel cas la définition ci-dessus implique que A rejette T.

Exemple 8.10. *On s'intéresse pour chaque $k \in \mathbb{N}$ au langage L_k sur l'alphabet $\Sigma = \{a, b\}$ défini de la façon suivante : l'arbre a au moins k feuilles, et la k-ième feuille de l'arbre dans l'ordre de parcours préfixe (ou postfixe, peu importe) est un a. On souhaite démontrer que ce langage est reconnaissable.*

On peut d'abord se donner un DbTA A avec un nombre d'états exponentiel en k. Posons Q l'ensemble des mots de Σ de longueur au plus k. Notons $\odot$ la concaténation tronquée à k sur l'ensemble de ces mots, c'est-à-dire que pour deux mots $u, v \in \Sigma^*$ de longueur au plus k, on a $u \odot v$ qui vaut uv si $|uv| \leq k$ et vaut le préfixe de uv de longueur k sinon. On va maintenir l'invariant que le run de A sur un arbre de T associe à tout nœud n le mot formé de la séquence des étiquettes des feuilles accessibles depuis n, dans l'ordre, en conservant au plus les k premières feuilles. On définit $\iota(x)$ pour chaque $x \in \Sigma$ comme le mot singleton x, ce qui satisfait manifestement l'invariant. Pour toute paire d'états $q_1, q_2 \in Q$, pour toute étiquette $x \in \Sigma$ (qu'on ignore), on définit $\delta(q_1, q_2, x)$ comme $q_1 \odot q_2$, ce qui préserve manifestement l'invariant. Enfin, l'ensemble des états finaux sont tous les mots $u \in \Sigma$ qui sont de longueur exactement k et finissent par a. L'invariant garantit ainsi que les arbres T pour lesquels le run de A sur T mène à un état final sont précisément les arbres où la racine a accès à au moins k feuilles et où la k-ième est un a, comme demandé.

On peut aussi se définir un NbTA A où le nombre d'états sera linéaire en k plutôt qu'exponentiel. L'ensemble Q des états se compose :

- *des entiers de 0 inclus à 42 exclu, qui inductivement seront accessibles à la racine d'arbres avec exactement le nombre de feuilles indiqué*
- *d'une valeur ≥ 42 qui inductivement seront accessibles à la racine d'arbres ayant au moins 42 feuilles*
- *des valeurs $i, *$ pour chaque i de 0 inclus à 42 exclu, qui inductivement seront accessibles en haut d'arbres où la $(i+1)$-ème feuille existe et est un a*

L'état final est $41, *$, qui par l'invariant indique que la 42e feuille existe et que c'est un a.

La relation initiale comprend $(a, 1)$, $(b, 1)$, et $(a, (0, *))$. Intuitivement, pour une feuille on peut soit deviner que c'est une feuille qui n'est pas celle qui sera la 42e, soit si c'est un a deviner que ce sera la 42e feuille.

La relation de transition contient, pour chaque étiquette de nœud interne $x \in \Sigma$ (ignorée) :

- *Des transitions permettant de compter le nombre de feuilles d'arbres sans état impliquant un $*$:*
  - *Pour $0 \leq i, j < 42$, si $i + j < 42$, une transition $(i, j, x, i + j)$*
  - *Pour $0 \leq i, j < 42$, si $i + j \geq 42$, une transition $(i, j, x, \geq 42)$*
  - *Pour $0 \leq i < 42$ et $j = \geq 42$, les transitions $(i, j, x, \geq 42)$ et $(j, i, x, \geq 42)$*
  - *La transition $(\geq 42, \geq 42, x, \geq 42)$*
- *Des transitions permettant de combiner les états de la forme $i, *$ avec les autres :*
  - *Pour chaque $0 \leq i < 42$, pour tout $0 \leq j < 42$ tel que $i + j < 42$, la transition $(i, (j, *), x, (i + j, *))$*
  - *Pour chaque $0 \leq i < 42$ et $0 \leq j < 42$, la transition $((i, *), j, x, (i, *))$*
  - *Pour chaque $0 \leq i < 42$, la transition $((i, *), \geq 42, x, (i, *))$*
- *Dans les autres cas aucune transition n'est définie. En particulier il n'y a pas de façon de combiner deux états impliquant un $*$, ni de combiner un état $j, *$ à droite avec un état i à*

gauche si $i = \geq 42$ *ou si* $i + j \geq 42$ *(ceci voudrait dire que la feuille devinée avec* $*$ *n'est pas la 42e)*

Pour constater que cet automate est correct, si on a un arbre où la 42e feuille n existe et est un a alors on peut lui attribuer l'état $0, *$ *et attribuer l'état* 1 *à toutes les autres feuilles. On verra alors que les ancêtres* n' *de n seront étiquetés par* $i, *$ *avec i le nombre de feuilles strictement à gauche de i dans le sous-arbre enraciné en* n'*, et les autres nœuds porteront une valeur de* $\{0, \ldots, 41, \geq 42\}$ *représentant leur nombre de feuilles. Sur la racine en particulier on aura l'état* $41, *$ *et l'automate aura donc un run acceptant.*

À l'inverse, sur un arbre où l'automate a un run acceptant, ce run étiquette la racine par $41, *$*. Par définition des transitions impliquant* $*$*, on peut descendre dans l'arbre en suivant à chaque fois le nœud qui portait un état impliquant* $*$ *dans ce run, et on arrive à une feuille n étiquetée a. On se persuade sans peine que, dans le reste du run, il n'y a aucun état impliquant* $*$ *et que l'état pour chaque nœud* n' *indique le nombre de feuilles accessibles depuis* n'*. En s'intéressant au chemin de la racine à n et aux cas où le chemin suit l'enfant droit, on voit que la somme des états des enfants gauches vaut 41, ainsi n est la 42e feuille et l'arbre était dans le langage. Ceci établit que l'automate est correct.*

On peut constater par ailleurs que l'automate ainsi défini est inambigu, au sens où sur chaque arbre du langage il n'y a qu'un seul run acceptant, à savoir celui où on devine $(0, *)$ *pour la 42e feuille et* 1 *sur toutes les autres et où on suit ensuite les transitions de la relation de transition (qui est en réalité déterministe).*

Comme nous le verrons en TD, les automates non-déterministes de bas en haut (NbTA) ne sont pas plus expressifs que les automates déterministes de bas en haut (DbTA) définis plus haut : on peut appliquer une construction de déterminisation qui ressemble à l'automate des parties. On peut aussi définir une notion d'automates *inambigus* de bas en haut, avec des propriétés analogues à celles des automates de mots :

Definition 8.11. *Un automate* inambigu *de bas en haut (UbTA) est un NbTA A avec la propriété que, sur tout arbre T, il existe au plus un run acceptant de A sur T.*

En particulier, l'automate de l'Exemple 8.10 est inambigu.

Pour ce qui concerne les automates de *haut en bas*, leur définition est intuitivement la suivante : il y a un état initial attribué à la racine, et pour chaque nœud interne l'étiquette du nœud et l'état attribué permet de connaître le couple d'états attribué au couple des enfants. Pour savoir si l'arbre est accepté, on vérifie si, pour chaque feuille, l'état attribué et l'étiquette de la feuille sont dans une certaine liste de couples autorisés.

Les automates de haut en bas peuvent être définis comme des automates déterministes ou nondéterministes. La version nondéterministe du formalisme a le même pouvoir d'expressivité que les automates nondéterministes de bas en haut. En fait, les deux modèles d'automates sont équivalents à renommage près :

— les états finaux de l'automate de bas en haut correspond aux états initiaux de l'automate de haut en bas
— la relation initiale de l'automate de bas en haut correspond aux couples d'étiquette et d'états autorisés aux feuilles pour l'automate de haut en bas
— la relation de transition restreint juste les quadruplets "étiquette de nœud interne, état de nœud interne, états des deux enfants" qui sont autorisés, on peut donc la voir indifféremment de bas en haut ou de haut en bas.

Pour le formalisme déterministe d'automates de haut en bas, en revanche, leur pouvoir expressif est plus faible que la version nondéterministe des automates de haut en bas, ou que les automates de bas en haut (déterministes ou non). En effet, les automates déterministes de haut en bas ne peuvent même pas reconnaître tous les langages finis : en particulier le langage formé de deux

arbres ayant racine a, l'un ayant pour enfants b et c, l'autre ayant pour enfants c et b. (On peut en fait montrer qu'un automate déterministe de haut en bas qui accepte ces deux arbres doit aussi accepter un l'arbre où la racine est un nœud a qui a deux enfants b.)

9 Décompositions arborescentes de graphes arbitraires

Dans cette dernière partie du cours, en guise d'ouverture, on s'intéresse à comment des méthodes inspirées par les automates peuvent s'appliquer à des données plus générales que les arbres. On cherche en particulier à les appliquer à des graphes généraux, et à définir une mesure décrivant à quel point un graphe est "proche" d'un arbre : la *largeur arborescente*.

9.1 Le problème de 3-coloriage avec contraintes

Comme exemple dans cette section, on va s'intéresser à un problème spécifique : le problème de *3-coloriage avec contraintes*. Nous avons vu en TD la définition de ce problème sur les arbres. En voici la définition sur les graphes généraux.

Definition 9.1. *On pose $\Sigma_{RGB} = \{R, G, B\}$ un ensemble formé de trois couleurs (rouge, vert, bleu). Soit $G = (V, E, \lambda)$ un* graphe étiqueté*, c'est-à-dire un graphe non-orienté (V, E) muni d'une fonction partielle $\lambda\colon V \to \Sigma_{RGB}$ associant une couleur de Σ_{RGB} à certains nœuds. Un* coloriage *de G est une fonction associant une couleur de Σ_{RGB} à tous les nœuds, c'est-à-dire une fonction totale c de V dans Σ_{RGB}. Un coloriage c est* valide *s'il satisfait les deux conditions suivantes :*

- *Le coloriage respecte les couleurs initiales données par λ, c'est-à-dire que pour tout nœud $v \in V$ où $\lambda(v)$ est définie on a $c(v) = \lambda(v)$*
- *Deux sommets adjacents n'ont jamais la même couleur, c'est-à-dire que pour toute arête $\{u, v\} \in E$ on a $c(u) \neq c(v)$*

Le problème de 3-coloriage avec contraintes *(3CC) demande de savoir, étant donné un graphe étiqueté, s'il admet un coloriage valide. Dans ce cas, on dit que le graphe étiqueté est* coloriable.

Exemple 9.2. *Le graphe 4-clique, formé de 4 sommets avec une arête reliant toute paire de deux sommets distincts, pour la fonction partielle λ définie nulle part, n'est pas un graphe coloriable : par le principe des tiroirs deux nœuds devront recevoir la même couleur et ces deux nœuds seront adjacents ainsi la condition ne sera pas respectée.*

Le graphe étoile à 3 branches, formé de 3 sommets u_1 u_2 u_3 tous reliés à un sommet central, muni d'une fonction λ définie uniquement pour u_1 et u_2, est toujours coloriable. Si λ est définie sur u_1, u_2, et u_3, il n'est coloriable que dans le cas où λ donne la même couleur à deux sommets distincts.

Le problème 3CC est facile dans certains cas, par exemple :

- Si λ donne la même couleur à deux sommets adjacents alors il n'y a jamais de solution, donc la réponse est toujours négative
- Si les sommets sont de degré maximal 2 (c'est-à-dire que le graphe est une union disjointe de chemins et de cycles) alors on peut donner itérativement à chaque sommet non colorié une couleur qui soit différente des couleurs déjà assignées à ses voisins, donc la réponse est toujours positive (sauf dans le cas du point précédent)
- Si le graphe d'entrée contient une 4-clique, c'est-à-dire 4 sommets u_1, u_2, u_3, u_4 deux à deux distincts et deux à deux adjacents, alors il n'y a pas de solution : dans tout coloriage, par le principe des tiroirs, deux sommets devront recevoir la même couleur et on aura ainsi deux sommets adjacents de même couleur.

Toutefois, dans le cas général, ce n'est pas facile de voir comment résoudre 3CC. En fait, il s'avère que le problème 3CC sur des graphes généraux est NP-difficile, et cela même si on impose que la fonction initiale λ ne soit définie nulle part. Ainsi, on ne peut pas espérer avoir d'algorithme efficace pour résoudre 3CC sur des graphes d'entrée quelconques.

On a toutefois vu en TD que, dans le cas où le graphe d'entrée est un arbre enraciné où chaque nœud interne a degré 2, le problème peut être résolu en temps linéaire, en passant par les automates d'arbres. Cette observation se généralise manifestement à des graphes acycliques satisfaisant la contrainte de degré, sans imposer qu'ils soient connexes. En effet, le problème 3CC peut manifestement se résoudre en considérant chaque composante connexe du graphe de départ séparément, ainsi on suppose que le graphe d'entrée est connexe.

On aimerait généraliser cette observation : y a-t-il d'autres classes de graphes sur lesquels on puisse résoudre le problème 3CC en temps linéaire ? On sait déjà que, si P est différent de NP, alors on ne pourra pas l'étendre à des graphes quelconques. En revanche il est raisonnable de penser que la condition sur le degré devrait pouvoir être relâchée ; et que cette approche devrait marcher pour des graphes qui ne sont pas des arbres, mais en sont proches. Par exemple, si on a un graphe G qui est un arbre avec seulement une arête supplémentaire $\{u, v\}$, un algorithme serait de tester les choix de couleurs possibles pour u et v (il y en a au plus 6), retirer l'arête entre u et v maintenant que leurs valeurs sont fixées, et appliquer l'algorithme précédent. Plus généralement, si on a un graphe G qui peut être transformé en une forêt en supprimant un petit nombre k de sommets, alors on a un algorithme en $O(3^k \times |G|)$ en testant de fixer tous les choix de couleur possibles pour ces sommets et de se ramener au point précédent.

On va donc voir une notion sur les graphes, la *largeur arborescente*, permettant de caractériser d'autres classes de graphes sur lesquels l'algorithme peut être appliqué.

9.2 Largeur arborescente

La *largeur arborescente* est une mesure sur les graphes qui sert à généraliser cette notion de graphes qui sont "presque" des arbres. On définit à présent cette notion, à travers les *décompositions arborescentes* :

Definition 9.3. *Soit $G = (V, E)$ un graphe non-orienté. Une* décomposition arborescente *de G est un arbre (non-enraciné) T où chaque nœud n est étiqueté par un sous-ensemble $\beta(n)$ de V. On impose les conditions suivantes sur l'étiquetage :*

- *Représentation des sommets : pour chaque $v \in V$, il y a au moins un nœud n tel que $v \in \beta(n)$*
- *Représentation des arêtes : pour chaque arête $e = \{u, v\}$ de E, il y a au moins un nœud n tel que $e \subseteq \beta(n)$*
- *Connexité : pour chaque $v \in V$, le sous-ensemble des nœuds de T où le sommet v apparaît est connexe dans T, c'est-à-dire que c'est un sous-arbre connexe (non nécessairement enraciné) de T. De façon équivalente, pour tous nœuds n et n' tels que $v \in \beta(n)$ et $v \in \beta(n')$, si on prend $n = n_0, \ldots, n_m = n'$ l'unique chemin simple dans T reliant n et n', alors pour chaque $0 \leq i \leq m$, on a $v \in \beta(n_i)$.*

La largeur *d'une décomposition arborescente est la plus grande cardinalité d'un sous-ensemble $\beta(n)$, moins un : autrement dit c'est* $\max_{n \in T} |\beta(n)| - 1$. *La* largeur arborescente *de G est la plus petite largeur d'une décomposition arborescente de G.*

Quelques exemples et remarques sont utiles pour comprendre cette définition. Tout graphe (V, E) admet une décomposition arborescente triviale formée d'un seul nœud n avec $\beta(n) = V$; mais la largeur arborescente ainsi obtenue est mauvaise, elle est de $|V| - 1$. On cherche donc des décompositions arborescentes où les ensembles $\beta(n)$ sont petits.

Les conditions sur une décomposition arborescente sont intuitivement conçues pour qu'on puisse résoudre des problèmes comme 3CC en suivant une décomposition arborescente T. Plus précisément :

- La condition de représentation des sommets impose que chaque sommet apparaisse dans un nœud de T, et donc qu'on choisisse une couleur pour lui.
- La condition de représentation des arêtes impose que chaque arête soit contenue dans un nœud de T, et donc qu'on vérifie à un moment que la couleur devinée pour les deux sommets de l'arête soit effectivement différente.
- La condition de connexité nous impose que, quand on voit un sommet v et qu'on choisit une couleur pour lui, alors on ne "perd pas de vue" le sommet v en circulant dans l'arbre. Si on atteint des nœuds où v n'est plus présent, alors la connexité nous garantit que v ne réapparaîtra plus jamais. Une réapparition de v serait un problème car on aurait alors oublié la couleur qu'on avait devinée pour v, ainsi on risquerait de deviner deux couleurs différentes pour v à deux endroits de la décomposition arborescente, ce qui pourrait donner l'impression que les contraintes sont satisfaites alors qu'en réalité le coloriage obtenu serait mal défini.

De plus, le fait que les images de β soient petites est prévu pour aider un algorithme qui suit la décomposition arborescente. En effet, intuitivement, le cardinal k de $\beta(n)$ représente le nombre de sommets dont on doit deviner la couleur au nœud n de T. On va en réalité plus précisément considérer chaque façon possible de colorier cet ensemble de sommets (et retenir cette valeur pour vérifier qu'elle est cohérente avec les nœuds voisins de T), ainsi la complexité sera exponentielle en k.

Graphes de faible largeur arborescente. Il est immédiat qu'un graphe a une décomposition arborescente de largeur 0 si et seulement s'il n'admet aucune arête : un tel graphe peut être décomposé par un arbre quelconque où chaque nœud apparaît seul dans un sommet, à l'inverse si un graphe admet une décomposition arborescente de largeur 0 alors la condition de représentation des arêtes interdit que le graphe ait une arête.

Pour la largeur arborescente 1, on a le résultat suivant :

Proposition 9.4. *Un graphe G a une décomposition arborescente de largeur 1 si et seulement si G est une forêt.*

Démonstration. On montre d'abord la première direction : une forêt admet une décomposition arborescente de largeur 1. Il suffit de montrer cela pour chaque arbre de la forêt, en effet il suffira ensuite de connecter successivement les décompositions de chaque arbre et on obtiendra une décomposition de la forêt (elle respectera manifestement les conditions de représentation, et les sous-arbres d'occurrence de chaque élément seront à l'intérieur de chaque décomposition et donc toujours des sous-arbres).

Étant donné un arbre $G = (V, E)$, s'il n'a pas d'arête la décomposition est triviale. Sinon, on prend une feuille quelconque $r \in V$ de G, et on enracine G en r, pour obtenir un arbre enraciné T. On va construire récursivement une décomposition arborescente T' de T, qui suivra la même structure que T, de sorte que chaque sommet n de T correspond à un nœud b_n de T'. On va garantir plus précisément l'invariant suivant : pour chaque nœud $n \neq r$ de T, en notant p le parent de n, en considérant le sous-arbre T_n^+ de T enraciné en n augmenté du sommet p et de l'arête $\{n, p\}$, alors le sous-arbre de T' enraciné en b_n est une décomposition arborescente de T_n^+ de largeur 1 où le nœud racine b_n est envoyé par β vers $\{n, p\}$.

Le cas de base de la construction est celui des feuilles de T, c'est-à-dire des sommets $n \in V$ qui n'apparaissent que dans une seule arête avec leur parent p. Pour ces sommets, on crée une

décomposition arborescente formée d'un unique nœud b_n avec $\beta(b_n) = \{n, p\}$. Cette décomposition satisfait manifestement les conditions de représentation de sommets et des arêtes de T_u, ainsi que la condition de connexité ; elle est manifestement de largeur 1.

Pour le cas d'induction, étant donné un nœud interne $n \neq r$ de T, p son parent, et $n_1, \ldots, n_m$ ses enfants, on obtient par induction des décompositions arborescentes $T'_1, \ldots, T'_m$ des sous-arbres augmentés $T_1^+, \ldots, T_m^+$ de T respectivement enracinés aux sommets $n_1, \ldots, n_m$, avec des racines respectivement $b_1, \ldots, b_m$ et $\beta(b_i) = \{n_i, n\}$ pour chaque $1 \leq i \leq m$ suivant l'invariant. On crée alors un nouveau nœud racine b_n pour la décomposition arborescente T'_n de T_n^+, que l'on relie aux racines $b_1, \ldots, b_m$; et on définit $\beta(b_n) := \{n, p\}$. Ceci satisfait manifestement la condition d'occurrence des sommets, car les sommets de T_n^+ sont l'union des sommets des T_i^+ pour $1 \leq i \leq m$ qui sont correctement représentés dans les décompositions arborescentes T'_i par hypothèse d'induction, plus p qui est représenté dans b_n. Pour la condition d'occurrence des arêtes, l'arête de T_n^+ reliant n à son parent p est reflétée dans le nœud racine b_n, et les autres arêtes impliquent un nœud qui n'est pas n et qui existe dans un T_i^+ pour un certain $1 \leq i \leq m$: par hypothèse d'induction ces arêtes sont reflétées dans la décomposition arborescente correspondante T'_i.

Enfin, pour la condition de connexité, elle s'obtient immédiatement par hypothèse d'induction pour tous les sommets W qui ne sont pas n, p, ni l'un des n_i, en effet les occurrences de chaque sommet de W sont limitées à l'un des T_i^+ qui satisfait la condition par hypothèse d'induction. Il suffit ainsi de vérifier les autres sommets. Pour p il n'apparaît que dans b_n donc la condition est vérifiée. Pour n il apparaît dans b_n, et dans chacun des b_i (et potentiellement dans un sous-arbre enraciné en b_i dans chaque T'_i), donc la condition est vérifiée. Pour les n_i, ils apparaissent seulement dans leur T'_i dans la construction, donc la condition est vérifiée aussi. Finalement on constate aussi immédiatement que T'_n est de largeur 1.

En conclusion, si on prend r' l'unique enfant de la racine r de T (cet enfant est unique car on a choisi r comme une feuille de T au moment d'enraciner T), on obtient une décomposition arborescente de largeur 1 de $T_{r'}^+$, et c'est en réalité une décomposition arborescente de T car $T_{r'}^+ = T$ puisque r' est l'unique enfant de r. Ceci conclut.

On montre ensuite la direction réciproque : si un graphe n'est pas une forêt, alors sa largeur arborescente est d'au moins 2. Il suffit en réalité de montrer que la largeur arborescente d'un cycle est d'au moins 2. En effet, si on a une décomposition arborescente T d'un graphe G qui n'est pas une forêt, alors G contient nécessairement un cycle C comme sous-graphe, et ainsi T est une décomposition arborescente de C donc elle est de largeur au moins 2.

Considérons un cycle C et supposons par l'absurde que C admette une décomposition arborescente T de largeur 1. On modifie d'abord T pour assurer la condition suivante (*) : il n'y a pas deux nœuds n et n' de T qui aient la même image pour β (c'est-à-dire $\beta(n) = \beta(n')$ et où $|\beta(n)| = 2$. En effet, si on a deux tels nœuds n et n', en prenant le chemin qui les relie dans T, la propriété de connexité implique que tous les nœuds ont la même image pour β. On peut alors les fusionner en un seul : cela ne change pas la largeur arborescente de la décomposition, cela respecte les conditions d'occurrences, et la connexion de connexité est vérifiée aussi car tout sous-arbre connexe de la décomposition initiale est un sous-arbre connexe de la nouvelle décomposition. Ainsi, en appliquant itérativement ce processus, on garantit la condition (*).

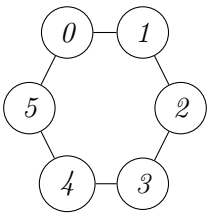
Considérons alors les trois premiers sommets du cycle (qui a au moins trois sommets) : u_1, u_2, u_3. On sait qu'il doit exister un nœud n de T tel que l'arête $\{u_2, u_3\}$ soit un sous-ensemble de $\beta(n)$, c'est-à-dire vu la contrainte de taille $\beta(n) = \{u_2, u_3\}$. De même on sait qu'il existe un nœud n_1 de T tel que $\beta(n_1) = \{u_1, u_2\}$. Enfin, il doit exister un nœud n_3 de T tel que $\beta(n_3) = \{u_3, u\}$, pour u le prochain sommet du cycle (soit u_4, soit u_1 si le cycle est un triangle). Posons $C_1 = n, n'_1, \ldots, n_1$ et $C_3 = n, n'_3, \ldots, n_3$ les chemins dans T allant de n à n_1 et n_3. Par connexité, le sommet u_2 apparaît sur l'image de β de chaque sommet de C_1, et le sommet u_3

apparaît sur l'image de β de chaque sommet de C_3. On sait donc que $n'_1 \neq n'_3$; en effet dans le cas contraire l'image par β de ce sommet serait la même que celle de n, contredisant le prétraitement (*) qu'on avait effectué plus haut.

On va maintenant aboutir à une contradiction. Considérons les nœuds de T $n_4, \ldots, n_{k-1}$, pour k la longueur du cycle, qui témoignent des arêtes $\{u_3, u_4\}, \ldots, \{u_{k-1}, u_k\}$ et n_k le nœud qui témoigne de l'arête $\{u_k, u_1\}$. (Noter que si $k = 3$ alors $n_k = n_3$ comme précédemment défini.) On sait qu'il y a un chemin C_4 reliant n_3 et n_4 dans lequel, par connexité, le sommet u_4 apparaît sur chaque nœud intermédiaire : donc il ne passe pas par le nœud n. De même, il y a un chemin C_5 reliant n_4 à n_5 qui ne passe pas par n pour la même raison. On répète le processus jusqu'à l'obtenion du chemin C_{k-1} qui aboutit en n_{k-1}, puis C_k qui aboutit en n_1. La concaténation des chemins $C_4 \cdots C_k$ donne donc un chemin de n_3 à n_1 qui ne passe pas par n. Or n_3 et n_1 sont deux voisins différents de n dans l'arbre T. Ceci contredit le fait que T soit un arbre. On a bien montré la seconde direction, ce qui conclut la démonstration. □

Pour la largeur arborescente de 2, d'autres exemples de graphes sont possibles. Par exemple, tout cycle a largeur arborescente de 2, ce qu'on démontre par un exemple :

Exemple 9.5. *On considère le 6-cycle G_1, c'est-à-dire le graphe $G = (V, E)$ où $V = \{0, \ldots, 5\}$ et où on a les arêtes $\{i, i+1\}$ pour $0 \leq i < 5$ et $\{0, 5\}$. Le graphe est illustré ci-dessous :*

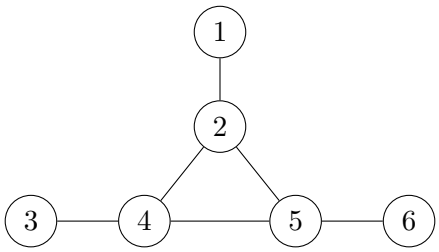


Une décomposition arborescente de G_1 est la suivante :

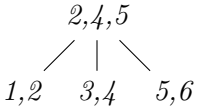
0, 1, 2 —— 0, 2, 3 —— 0, 3, 4 —— 0, 4, 5

Voici un exemple un peu plus complexe de décomposition arborescente :

Exemple 9.6. *On considère le graphe G_2 suivant :*



Un exemple de décomposition arborescente pour ce graphe est :



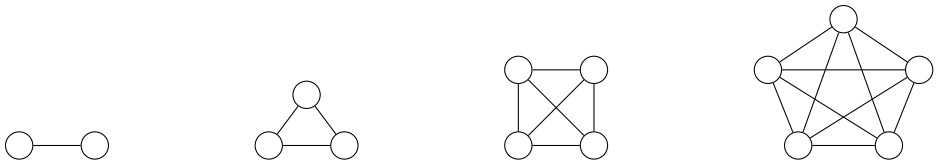


FIGURE 15 – Les cliques à 2, 3, 4, et 5 sommets

Graphes de forte largeur arborescente. On ne montrera pas de techniques permettant de donner des bornes inférieures sur la largeur arborescente d'un graphe donné, mais on se contentera de deux exemples : les *cliques*, et les *graphes grille*.

Definition 9.7. *Pour tout $n \in \mathbb{N}$, la n-*clique*, notée K_n, est le graphe à n sommets avec une arête entre chaque paire de sommets distincts. Les cliques K_2, K_3, K_4, K_5 sont illustrées en Figure 15.*

Definition 9.8. *Pour tout $n \in \mathbb{N}$, la n-*grille*, notée G_n, est le graphe dont les sommets V sont les couples d'entiers (i, j) avec $1 \leq i, j \leq n$ et avec une arête entre $(i, j) \in V$ et $(i', j') \in V$ si $|i - i'| + |j - j'| = 1$. Autrement dit, une arête relie chaque sommet au sommet obtenu en ajoutant ou retranchant 1 à l'une des composantes (à condition que celle-ci reste dans l'ensemble $\{1, \ldots, m\}$).*

Proposition 9.9. *Pour tout $n \geq 1$, la n-clique K_n a une largeur arborescente de $n - 1$.*

Démonstration. La borne supérieure est immédiate avec une décomposition arborescente consistant en un unique nœud n où $\beta(n)$ est l'ensemble de tous les sommets du graphe. On rappelle que cette construction triviale (mais généralement peu intéressante) vaut pour des graphes arbitraires.

Pour la borne inférieure, on traite d'abord la décomposition arborescente T pour garantir la condition (*) : il n'y a pas deux nœuds n et n' voisins dans T tels que $\beta(n) \subseteq \beta(n')$. En effet si c'est le cas on peut simplement fusionner n dans n', c'est-à-dire rattacher tous les voisins de n à n' et supprimer n. Ceci respecte manifestement les conditions d'occurrences (car n' témoigne de tout ce dont n pouvait témoigner) et pour les conditions de connexité tout sous-arbre connexe de T avant modification donne un sous-arbre connexe de T après modification. En répétant cette modification itérativement on garantit que (*) est respecté.

À présent supposons par l'absurde qu'on ait une décomposition arborescente T de K_n de largeur $< n - 1$, c'est-à-dire que pour chaque nœud b on a $|\beta(b)| \leq n - 2$. On peut supposer que T satisfait (*). Clairement T a au moins deux nœuds, car un seul nœud de cardinalité $\leq n - 2$ ne peut suffire à satisfaire la condition d'occurrence des sommets. Prenons donc deux nœuds p et p' voisins de T. Par la condition (*) on sait qu'aucun n'est un sous-ensemble de l'autre. On peut donc se donner deux sommets u et v de K_n qui en témoignent, c'est-à-dire que $u \in \beta(p')$ mais $u \notin \beta(p)$, et $v \in \beta(p)$ mais $v \notin \beta(p')$.

Comme $u \neq v$, l'arête $\{u, v\}$ existe dans K_n par définition de K_n. Par la condition d'occurrence des arêtes, l'arête $\{u, v\}$ doit être représentée dans un certain sommet x de T. Par la condition de connexité, comme $u \in \beta(p')$ et $u \in \beta(x)$, il y a un chemin C' de x à p' dans T tel que u apparaisse dans l'image par β de tous les sommets : en particulier comme $u \notin \beta(p)$ on sait que p n'apparaît pas dans ce chemin. Comme $v \in \beta(p)$ et $v \in \beta(x)$, il y a un chemin C de x à p dans T tel que v apparaisse dans l'image par β de tous les sommets : en particulier comme $v \notin \beta(p')$ on sait que p' n'apparaît pas dans ce chemin. Prenons y le dernier sommet commun de C et C' ; on sait que $y \neq p$ et $y \neq p'$ car p n'apparaît pas dans C' et p' n'apparaît pas dans C.

En résumé, il y a donc un chemin C'' de p à p' qui passe par un troisième sommet $y \notin \{p, p'\}$ et C'' n'a aucun sommet répété (en effet par définition les deux morceaux de chemin ne partagent

aucun sommet). Mais par ailleurs p et p' sont reliés par une arête de T. Ceci contredit le fait que T soit un arbre. □

Proposition 9.10 (admis)**.** *Pour tout $n \geq 2$, la n-grille G_n a une largeur arborescente de n.*

Calcul de la largeur arborescente et de décompositions arborescentes. Des algorithmes de calcul de décompositions arborescentes existent, en théorie comme en pratique, de manière exacte ou approchée. On n'étudiera pas ces algorithmes, et on se contentera d'admettre les résultats suivants sur le calcul de la largeur arborescente :

Theorem 9.11 (admis)**.** *Il existe un algorithme qui prend en entrée un graphe G et un entier $k \in \mathbb{N}$ et qui, en temps $O(|G|^k)$, détermine si G est de largeur arborescente $> k$ ou non. Dans le cas où G est de largeur arborescente $\leq k$, l'algorithme produit une décomposition arborescente de G de largeur minimale.*

Par ailleurs, pour tout $k \in \mathbb{N}$, il existe une constante $c_k \in \mathbb{N}$ garantissant ce qui suit : il existe un algorithme qui prend en entrée un graphe G et qui, en temps $O(c_k|G|)$, détermine si G est de largeur arborescente $> k$ ou non, et produit une décomposition arborescente de G de largeur minimale dans le cas où G est de largeur arborescente $\leq k$.

Par ailleurs, il est NP-difficile, étant donné un graphe G et un entier $k \in \mathbb{N}$, de déterminer si la largeur arborescente de G est $\leq k$ ou non.

Les bornes de ces théorèmes ne sont pas utiles sur des graphes quelconques : en général, comme la largeur arborescente d'un graphe à n sommets peut être aussi grande que $n-1$, ces algorithmes ne donneront une décomposition arborescente optimale qu'au prix d'un calcul de complexité $O(|G|^{n-1})$ (pour le premier algorithme) ou bien $O(c_{n-1}|G|)$ (or, la croissante asymptotique de c_k en fonction de k est exponentielle). L'intérêt de ces algorithmes est de s'exécuter rapidement si on sait (ou si on espère) que le graphe d'entrée est de faible largeur arborescente. Dans le cas où on ne fait aucune hypothèse sur le graphe d'entrée, des approches algorithmiques efficaces sont possibles pour calculer une approximation de la largeur arborescente (avec des garanties formelles sur la qualité de l'approximation), ou bien pour calculer des décompositions arborescentes utilisables en pratique (sans garanties formelles).

Décompositions arborescentes normalisées. Outre ce qu'elles nous apprennent sur la structure des graphes, les décompositions arborescentes sont utiles pour leurs applications algorithmiques. Pour le voir concrètement, il sera plus commode de travailler sur des décompositions arborescentes dites *normalisées*. Dans ces décompositions, on choisit arbitrairement une racine, et on impose que l'arbre obtenu soit binaire complet (c'est-à-dire que chaque nœud interne a exactement deux enfants), comme dans le cas des Σ-arbres. On va voir que toute décomposition arborescente peut être normalisée dans difficultés, en temps linéaire ; et que cette forme normalisée sera plus commode pour les applications algorithmiques à venir. Remarquons au demeurant que d'autres formes normalisées de décompositions arborescentes sont également connues dans la littérature, en imposant des conditions plus fortes.

Definition 9.12. *Une* décomposition arborescente normalisée *est une décomposition arborescente T où l'on distingue un sommet r appelé* racine *et où on garantit que la racine a exactement deux voisins et que tout sommet distinct de la racine a soit exactement 1 voisin (une feuille) soit exactement 3 voisins (un nœud interne).*

On considérera fréquemment les décompositions arborescentes normalisées comme des arbres enracinés en leur nœud distingué. Dans ce cas, on remarquera que la racine sera un nœud interne avec exactement deux enfants, et que plus généralement chaque nœud interne a exactement deux enfants.

Proposition 9.13. *Étant donné un graphe G et une décomposition arborescente T de G de largeur k, on peut calculer en temps $O(|T|)$ une décomposition arborescente normalisée T' de G qui est toujours de largeur k.*

La difficulté principale consiste à transformer la décomposition arborescente pour borner le degré maximal.

Démonstration. L'algorithme procède en trois étapes. Premièrement, on garantit que le degré maximal de tous les nœuds est de 3 au plus. Deuxièmement, on enracine la décomposition arborescente en garantissant que la racine a au plus deux enfants. Troisièmement, on répare les cas où il y a des nœuds internes avec un seul enfant.

Pour la première étape, on élimine itérativement les nœuds de degré plus grand que 3 de la manière suivante, en préservant le fait qu'on a une décomposition arborescente et que la largeur est inchangée. Soit b un nœud de T de degré strictement supérieur à 3, et soient $b_1, \ldots, b_m$ ses voisins avec $m > 3$. On remplace b par des copies $b'_1, \ldots, b'_{m-2}$ de b avec $\beta(b'_i) := \beta(b)$ pour chaque $1 \leq i \leq m-2$, et on connecte les anciens voisins de b de la manière suivante : b_1 et b_2 sont connectés à b'_1, b_{m-1} et b_m sont connectés à b'_{m-2}, et pour chaque $3 \leq i \leq m-2$ on connecte b_i à b'_{i-1}. On se convainc sans peine que tous les nouveaux sommets $b'_1, \ldots, b'_{m-2}$ ont degré exactement 3 : ils ont chacun 1 voisin parmi les b_i sauf b'_1 et b'_{m-2} qui en ont 2, et ils ont chacun 2 voisins parmi les b'_i sauf b'_1 et b'_{m-2} qui en ont seulement 1. Au demeurant, le degré de tous les autres sommets est inchangé. Par ailleurs, cette transformation ne casse pas les propriétés d'une décomposition arborescente : les conditions sur les occurrences sont respectées parce que les ensembles obtenus comme images de β n'ont pas changé, et les conditions sur la connexité n'ont pas changé : les chemins qui passaient par b passent maintenant par un chemin dans les b'_i.

On remarque que la transformation effectuée élimine un sommet de degré > 3 pour le remplacer par des sommets de degré 3 sans changer les autres degrés. Ainsi, le processus termine après un nombre linéaire de remplacements. Pour être plus précis, on peut explorer la décomposition arborescente T par un parcours de graphe, et effectuer la modification à chaque nœud rencontré en temps linéaire en le degré de ce nœud : ceci permet de s'assurer que la première étape peut être effectuée en une complexité linéaire en $|T|$.

On appelle T_1 la décomposition arborescente obtenue au terme de la première étape : dans T_1, chaque nœud a au plus 3 voisins.

Pour la deuxième étape, on choisit un nœud n quelconque de T_1 de degré 1 : comme T_1 est un arbre, un tel nœud existe nécessairement. On prend n comme racine : elle n'a qu'un seul voisin. On appelle T_2 l'arbre enraciné ainsi obtenu. Dans T_2, la racine n'a qu'un seul enfant, et chaque nœud interne qui n'est pas la racine a au plus 2 enfants : en effet, il est relié à son parent et à ses enfants, or le degré de chaque nœud dans T_1 est de 3 au plus. Cette étape est manifestement en temps constant, ou en temps linéaire si on tient à reconstruire T_2 comme un arbre enraciné.

Ainsi, pour la troisième étape, il ne reste qu'à rectifier le problème suivant : certains nœuds internes (en particulier la racine) n'ont qu'un seul enfant. La parade est simple : pour chaque nœud n de ce type, on ajoute à n un second enfant n' avec $\beta(n') := \beta(n)$. Cette opération est manifestement en temps linéaire. Les conditions sur la décomposition arborescente ne sont pas affectés par cette transformation : les conditions d'occurrence sont vérifiées car on a ajouté des nœuds, et la condition de connexité est vérifiée car un sous-arbre d'occurrences inclut n si et seulement si il inclut n'. Du reste, la largeur est manifestement la même qu'auparavant.

Au terme de ces trois étapes, on obtient donc en temps linéaire une décomposition arborescente de même largeur qui est de plus normalisée. □

9.3 Décompositions arborescentes pour le problème de 3-coloriage avec contraintes

Maintenant qu'on a introduit les décompositions arborescentes, et les décompositions arborescentes normalisées, on va voir en quoi elles peuvent être utiles pour des problèmes algorithmiques. Plus précisément, on va s'intéresser au problème de 3-coloriage avec contraintes (3CC) introduit en Section 9.1.

L'objet de cette section est de démontrer le résultat suivant, sous réserve des résultats précédemment admis. Dans ce résultat, on désigne par *graphe étiqueté* l'entrée du problème 3CC comme en Définition 9.1, et la *largeur arborescente* d'un tel graphe est simplement la largeur arborescente du graphe non-orienté sous-jacent.

Theorem 9.14. *Pour toute constante $k \in \mathbb{N}$, il existe un algorithme qui prend en entrée un graphe étiqueté G et accomplit en temps $O(|G|)$ la tâche suivante : déterminer que G a une largeur arborescente $> k$ et échouer, sinon déterminer si le problème 3CC sur G admet une solution.*

La démarche de preuve est la suivante. À partir du graphe étiqueté $G = (V, E, \lambda)$, on utilise d'abord le second résultat du Théorème 9.11 pour obtenir, en temps linéaire en G, une décomposition arborescente du graphe non-orienté sous-jacent $G' = (V, E)$ de largeur au plus k ; ou bien pour déterminer que G est de largeur arborescente $> k$. Ensuite, on utilise la Proposition 9.13 pour calculer, toujours en temps linéaire, une décomposition arborescente *normalisée* T de G' de largeur au plus k. Il reste donc à prouver le résultat suivant :

Proposition 9.15. *Pour toute constante $k \in \mathbb{N}$, étant donné un graphe étiqueté $G = (V, E, \lambda)$ et une décomposition arborescente normalisée T de largeur $\leq k$ du graphe sous-jacent $G' = (V, E)$, on peut déterminer en temps $O(|T|)$ si le problème 3CC sur G a une solution.*

Pour mémoire, la Proposition 9.4 montre que les graphes de largeur arborescente ≤ 1 sont précisément les forêts. Ainsi, cette proposition généralise le résultat de tractabilité de 3CC sur les arbres que nous avions précédemment établi en TD.

L'intérêt de la Proposition 9.15 n'est pas vraiment le résultat en lui-même, car le problème 3CC est un exemple jouet qui n'a guère d'intérêt intrinsèque. Ce qui est intéressant est la technique de preuve, qui procède par programmation dynamique sur une décomposition arborescente. Cette façon de procéder se généralise à de nombreux autres problèmes, et permet de conclure que quantité d'autres tâches peuvent s'accomplir en temps linéaire quand la largeur arborescente du graphe d'entrée est bornée par une constante. On mentionnera à la fin en quoi ce résultat peut être reformulé en termes d'automates d'arbres.

Démontrons à présent la Proposition 9.15 :

Démonstration. On procède par programmation dynamique sur la décomposition arborescente normalisée T. Soit $G = (V, E, \lambda)$ le graphe étiqueté d'entrée. Pour chaque nœud b de la décomposition arborescente, on appelle T_b le sous-arbre de T formé de b et de tous ses descendants, et on appelle G_b le sous-graphe de G induit par les sommets qui apparaissent dans T_b. En d'autres termes, si on pose V_b le sous-ensemble de V obtenu comme $\bigcup_{b' \text{ descendant de } b} \beta(b')$, alors G_b est le sous-graphe de G induit par V_b, c'est-à-dire $G_b = (V_b, E_b, \lambda_b)$ où $E_b = \{e \in E \mid e \subseteq V_b\}$ et λ_b est la restriction de λ à V_b. On remarque, ce qui sera important pour la preuve inductive, que pour chaque nœud b de la décomposition arborescente, T_b est une décomposition arborescente de G_b (de largeur inférieure ou égale à celle de T bien sûr). En effet toutes les conditions sont faciles à vérifier : la seule subtilité est la condition d'occurrence des arêtes, où il faut remarquer que si le sommet de T qui témoigne d'une arête e avec $e \subseteq V_b$ est hors du sous-arbre T_b, alors comme

les sommets de e sont dans V_b donc apparaissent dans des nœuds de T_b la connexité impose que $e \subseteq \beta(b)$ ainsi on pouvait choisir b comme sommet témoin pour e dans T_b.

Pour chaque nœud b de T, on va calculer un tableau $T[b]$ indiquant les informations suivantes : pour chaque coloriage des sommets de $\beta(b)$, c'est-à-dire pour chaque fonction c de $\beta(b)$ dans $\{R, G, B\}$, alors $T[b][c]$ vaudra 1 s'il existe un coloriage de G_b dont la restriction à $\beta(b)$ soit c, et 0 sinon. Noter que le nombre de tels coloriages est borné par 3^{k+1}, car $|\beta(b)| \leq k+1$ par définition de la largeur arborescente.

En particulier, pour r la racine de G, on a $V_r = V$, donc $G_r = G$, et $T[r][c]$ pris sur toutes les valeurs possibles de c nous indiquera s'il existe un coloriage de G ou non.

On calcule T de bas en haut sur la décomposition arborescente normalisée T. Le cas de base est celui d'une feuille b. Dans ce cas, $V_n = \beta(b)$, et G_n est simplement le sous-graphe de G induit par $\beta(b)$. Ainsi, on considère chaque coloriage $c\colon \beta(b) \to \{R, G, B\}$, et pour chaque tel c on vérifie qu'on respecte les conditions de 3CC : c'est-à-dire que si $u \in \beta(b)$ est dans le domaine de λ on impose que $\lambda(u) = c(u)$, et si deux sommets $u, v \in \beta(b)$ sont reliés par une arête on vérifie que $c(u) \neq c(v)$. On remplit T en conséquence. La condition inductive est manifestement vérifiée.

Pour le cas d'induction, considérons un nœud b de la décomposition arborescente, et soit b_1 et b_2 ses enfants. Considérons chaque coloriage $c\colon \beta(b) \to \{R, G, B\}$. On vérifie d'abord que ce coloriage satisfait les conditions de 3CC sur les sommets de $\beta(b)$ et pour les arêtes entre sommets de $\beta(b)$, comme dans le cas de base : donc on ne considère que de tels coloriages c. Notre objectif pour l'invariant inductif est d'étudier le graphe G_b et de savoir s'il admet un coloriage dont la restriction à $\beta(b)$ corresponde à c.

On observe alors la propriété cruciale suivante (*) : $V_{b_1} \cap V_{b_2} \subseteq \beta(b)$. Autrement dit, tout sommet de G qui apparaît à la fois dans T_{b_1} et dans T_{b_2} doit également apparaître dans $\beta(b)$. En effet, comme tous les chemins entre T_{b_1} et T_{b_2} passent par b, ce serait une violation de la propriété de connexité s'il en était autrement. Donc, pour définir un coloriage de V_b, on peut choisir un coloriage de V_{b_1}, un coloriage de V_{b_2}, et un coloriage de $\beta(b)$; et on peut simplement vérifier que ces coloriages soient cohérents en vérifiant que l'image des coloriages coïncide sur les sommets de $\beta(b)$.

Ainsi, ayant fixé le coloriage c de $\beta(b)$, on cherche à savoir si on peut l'étendre en un coloriage de V_b qui satisfasse les conditions de 3CC. On considère alors tous les coloriages c_1 de $\beta(b_1)$ et c_2 de $\beta(b_2)$ qui soient cohérents avec c et qui puissent être étendus en des coloriages de G_{b_1} et G_{b_2} respectivement, comme nous l'avons récursivement calculé dans les tableaux $T[b_1]$ et $T[b_2]$. Si on trouve de tels coloriages c_1 et c_2, alors on met $T[n][c]$ à vrai, sinon à faux.

Montrons la correction de ce calcul dans le cas d'induction (le cas de base est immédiat). Si G_b admet un coloriage qui satisfait 3CC et dont la restriction à $\beta(b)$ soit c, alors on sait que c satisfait les conditions de 3CC sur les sommets de $\beta(b)$ et entre eux, donc on a considéré c dans l'algorithme. De plus, considérons ses restrictions c_1 et c_2 à V_{b_1} et V_{b_2} respectivement : ces coloriages sont cohérents avec c et sont extensibles en des coloriages de G_{b_1} et G_{b_2}, de fait le coloriage de départ témoigne de ce fait. Par hypothèse d'induction on avait $T[b_1][c_1]$ et $T[b_2][c_2]$ à vrai, et donc on a mis $T[b][c]$ à vrai ce qui est correct.

Le sens réciproque est plus délicat. Supposons que $T[b][c]$ soit à vrai, et construisons un coloriage témoin qui satisfasse 3CC. Prenons c_1 et c_2 les coloriages témoins de $\beta(b_1)$ et $\beta(b_2)$. Par hypothèse d'induction, ces coloriages peuvent être étendus en des coloriages c'_1 et c'_2 de G_{b_1} et G_{b_2} qui satisfont les conditions. Par construction, c et c_1 et c et c_2 sont cohérents. Ainsi, par la propriété (*) montrée plus haut, c'_1 et c'_2 sont cohérents. Posons c' le coloriage de G_b obtenu à partir de c, c'_1, et c'_2 : nous avons montré qu'il est bien défini. Montrons qu'il satisfait les conditions de 3CC. Pour les conditions sur les sommets, c'est facile : chaque sommet u apparaît par définition de G_b soit dans b, soit dans T_{b_1}, soit dans T_{b_2}, ainsi on a bien choisi une couleur pour u dans c' qui soit compatible avec λ si $\lambda(u)$ est défini.

Pour les conditions sur les arêtes, à présent, il faut d'abord constater que T_b est une décomposition arborescente normalisée de G_c. En effet, les conditions d'occurrence pour les sommets et la condition de connexité sont claires, du reste T_b est manifestement un arbre binaire enraciné où chaque nœud interne est de degré au plus 3. Pour les arêtes, on sait que chaque arête $\{u, v\}$ de G apparaît dans un nœud b' de T. Montrons qu'on peut prendre b' comme descendant de b. Or on sait qu'il y a des nœuds descendants de b où chacun de u et de v apparaît, notons-les b_u et b_v. Si l'on prend un sommet b' tel que $u, v \in \beta(b')$, soit b' est descendant de b' et c'est terminé, soit le chemin dans T entre b' et b_u passe par b, et le chemin entre b' et b_v passe par b. Ainsi, la condition de connexité sur T garantit que $u, v \in \beta(b)$ de sorte qu'on pouvait en réalité choisir $b' := b$.

Ainsi, on a démontré que pour chaque arête $e = \{u, v\}$ de G_c il y avait un nœud b_e de T_b satisfaisant $e \subseteq \beta(b_e)$. Ainsi, si $b_e = b$, on a bien vérifié en définissant c que c' donnait des couleurs différentes à u et v. Sinon, soit b_e est descendant de b_1, soit de b_2. Dans le premier cas, on sait que e est aussi une arête de G_{c_1}. Par hypothèse d'induction, on sait alors que c'_1 donne des couleurs différentes à u et à v. Il en va donc de même pour c'. Le second cas est symétrique.

On a donc démontré que notre algorithme était correct. Le temps d'exécution revient à tester à chaque nœud au plus 3^{k+1} coloriages, et à tester au plus $(3^{k+1})^2$ façons de les étendre (choix de c_1 et de c_2). Ainsi, la complexité est en $O((3^{k+1})^3 \times |T|)$. Pour k constant, on obtient bien $O(|T|)$ comme annoncé. (À noter que la complexité en k obtenue ici n'est qu'une majoration grossière ; de meilleures bornes sont possibles.) □

Lien avec les automates d'arbres et généralisations. La preuve que nous avons donnée pour la Proposition 9.15 procède par programmation dynamique sur la décomposition arborescente, et elle est très analogue aux méthodes à base d'automate d'arbres que nous avons vu en TD pour résoudre le problème 3CC sur les arbres (en supposant qu'ils sont enracinés et que tout nœud interne a exactement deux enfants). Une question naturelle est donc de savoir si on peut utiliser les automates d'arbres sur les décompositions arborescentes pour prouver le résultat de la Proposition 9.15 avec de telles techniques.

La réponse à cette question est positive, mais il y a une subtilité : il faut définir sur quel alphabet fini est-ce que l'automate d'arbre s'exécute. Lorsque l'on travaille directement avec un arbre, cela ne pose pas de difficulté car on découvre les nœuds les uns après les autres, et les arêtes sont données par la structure de l'arbre. Mais, dans une décomposition arborescente, on ne voit pas directement les arêtes du graphe original. En fait, lorsque l'on considère une décomposition arborescente normalisée dont la largeur est bornée par $k + 1$, il faut s'imaginer que l'étiquette d'un nœud n est en quelque sorte le sous-graphe du graphe original induit par les sommets de $\beta(n)$. Ainsi, on sait quelles sont les arêtes qui existent entre ces sommets dans le graphe original ; et on est sûr que chaque arête sera vue sur un nœud de T.

Plus précisément, pour voir une décomposition arborescente normalisée comme un Σ-arbre pour un alphabet fini Σ, il faut que Σ consiste en l'ensemble des graphes possibles sur $k + 1$ sommets : c'est un ensemble très grand, de taille $O(2^{(k+1)^2})$, mais fini et dépendant uniquement de k et non de la taille du graphe. Il faut également incorporer dans Σ l'information de quels nœuds sont égaux à quels nœuds : en effet, lorsque l'on considère un nœud interne n ayant pour enfants n_1 et n_2, l'étiquette de n doit non seulement nous indiquer quels arêtes existent entre les nœuds de $\beta(n)$, mais aussi quels nœuds de $\beta(n)$ sont égaux à des nœuds de $\beta(n_1)$ et/ou de $\beta(n_2)$, et à quels nœuds précisément. (Souvenez-vous qu'on ne peut pas directement utiliser le nom des nœuds dans Σ, puisque Σ ne peut dépendre que de k et non de la taille du graphe original.) Mais là encore, sachant que les graphes ont au plus $k + 1$ sommets, il faut stocker pour chacun des éléments de $\beta(n)$ (au plus $k + 1$) à quel élément de $\beta(n_1)$ et de $\beta(n_2)$ il est égal (ou potentiellement aucun). On peut majorer la taille de ces informations grossièrement par

$((k+2)^2)^{k+1}$ par exemple.

C'est à cause de ces problèmes techniques qu'on ne spécifie pas en détail comment utiliser des automates d'arbres sur des décompositions arborescentes normalisées : mais c'est donc possible. La complexité d'algorithmes utilisant de tels automates sera donc très mauvaise en k (en particulier on ne voudra généralement pas décrire explicitement les transitions de l'automate, et on les représentera plutôt de manière implicite), mais elle sera linéaire en le graphe d'entrée à k fixé.

Une dernière question que l'on peut se poser est celle des problèmes pour lesquels ces techniques sont applicables. En effet, il n'est manifestement pas vrai que tous les problèmes computationnels imaginables sont faciles à résoudre sur les graphes de largeur arborescente bornée, ou même plus simplement sur les arbres. (Considérer par exemple le problème de déterminer si le nombre de nœuds d'un arbre d'entrée correspond à l'index d'une machine de Turing qui termine.) Y a-t-il une façon générale de décrire des problèmes qui, comme 3CC, sont faciles à résoudre sur les arbres et les graphes de largeur arborescente bornée ? Une réponse positive à cette question est donnée par le *théorème de Courcelle* : il concerne un formalisme logique appelé *logique monadique du second ordre* (MSO), et il montre que tout problème computationnel exprimable en MSO peut se ramener, sur les graphes de largeur arborescente constante, à un problème MSO sur une forme de décomposition arborescente normalisée. Or, le formalisme MSO sur les arbres a le même pouvoir d'expression que les automates d'arbres (ceci est d'ailleurs également vrai sur les mots). Ces considérations datant des années 90 sont le point de départ de nombreuses recherches, toujours en cours, qui visent à identifier des notions analogues à la largeur arborescente qui garantissent la tractabilité de problèmes moins complexes en imposant des restrictions structurelles plus faibles ; ou qui visent à démontrer par exemple que certains problèmes sont infaisables sur toute classe de graphes de largeur arborescente non-bornée ; ou qui visent à analyser plus finement la complexité en fonction du graphe d'entrée et de la largeur arborescente (par exemple au travers du domaine de la *complexité paramétrée*).

Références

[1] *Deterministic pushdown automata*. Addison-Wesley Longman Publishing Co., Inc., USA, 1969.

[2] M. Arenas, L. A. Croquevielle, R. Jayaram, and C. Riveros. #NFA admits an FPRAS : Efficient enumeration, counting, and uniform generation for logspace classes. *Journal of the ACM (JACM)*, 68(6), 2021. `https://arxiv.org/abs/1906.09226`.

[3] T. Colcombet. Regular cost functions, part i : logic and algebra over words. *Logical methods in computer science*, 9, 2013.

[4] W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki. The reachability problem for Petri nets is not elementary. *Journal of the ACM (JACM)*, 68(1), 2020.

[5] W. Czerwiński and Ł. Orlikowski. Reachability in vector addition systems is ackermann-complete. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1229–1240. IEEE, 2022.

[6] e_noether (https ://cs.stackexchange.com/users/4980/e noether). Is determinism = non determinism for one counter automata ? Computer Science Stack Exchange. URL : `https://cs.stackexchange.com/q/7286` (version : 2012-12-10).

[7] T. N. Hibbard and J. Ullian. The independence of inherent ambiguity from complementedness among context-free languages. *Journal of the ACM (JACM)*, 13(4) :588–593, 1966.

[8] M. Holzer and M. Kutrib. Descriptional complexity of (un) ambiguous finite state machines and pushdown automata. In *International Workshop on Reachability Problems*, pages 1–23. Springer, 2010.

[9] H. J. (https ://math.stackexchange.com/users/45179/hendrik jan). Show that this language cannot be accepted by a deterministic push-down automaton. Mathematics Stack Exchange. URL :https ://math.stackexchange.com/q/359804 (version : 2017-04-13).

[10] C. A. Kapoutsis. Two-way automata versus logarithmic space. *Theory of Computing Systems*, 55, 2014.

[11] O. Martynova and A. Okhotin. Non-closure under complementation for unambiguous linear grammars. *Information and Computation*, 292 :105031, 2023.

[12] C. Moore. Lecture notes on automata, languages, and grammars, 2012. `https://sites.santafe.edu/~moore/500/automata-notes.pdf`.

[13] J.-E. Pin. Mathematical foundations of automata theory. `https://www.irif.fr/~jep/PDF/MPRI/MPRI.pdf`, 2019.

[14] S. R. Schwer. The context-freeness of the languages associated with vector addition systems is decidable. *Theoretical Computer Science*, 98(2), 1992.

[15] Wikipedia. Linear bounded automaton, 2024.