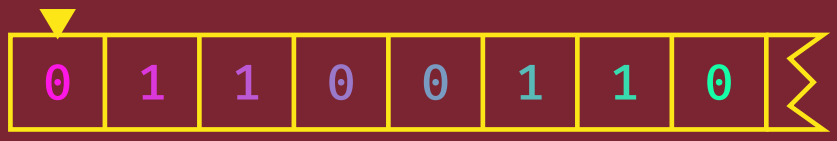


Théorie du calcul

Notes de cours (IFT503/711)

Michael Blondin

14 mars 2025

Avant-propos

La rédaction de ce document a été entamée à la session d'hiver 2025 comme notes personnelles du cours « IFT503/711 – Théorie du calcul » de l'Université de Sherbrooke. Ces notes ne contiennent que les chapitres que j'enseigne.

Les sections 1.3 et 1.4 du chapitre 1 s'inspirent fortement de [Sip06, chap. 3]. Le chapitre 4 s'inspire de [Sip06, chap. 6.2], mais contrairement à ce dernier, le chapitre 4 présente les détails techniques de la décidabilité et l'indécidabilité de l'arithmétique.

Si vous trouvez des coquilles ou des erreurs dans le document, ou si vous avez des suggestions, n'hésitez pas à me les indiquer par courriel à

michael.blondin@usherbrooke.ca.

Je remercie Alexandre Parent (A25) pour des suggestions au chapitre 1.

Université de Sherbrooke

Légende

Observation.

Les passages compris dans une région comme celle-ci correspondent à des observations jugées intéressantes mais qui dérogent légèrement du contenu principal.

Remarque.

Les passages compris dans une région comme celle-ci correspondent à des remarques jugées intéressantes mais qui dérogent légèrement du contenu principal.

Les chapitres et (sous-)sections marqués par « 🎓 » sont avancés et optionnels.

L'icône «  » dans la marge fournit un lien vers du code associé au passage.

Les icônes « ↑↓ » dans la marge permettent de naviguer vers les démonstrations, solutions ou autres descriptions qui se trouvent en annexe.

Exercices

Chaque exercice est annoté par trois symboles:

Symbole	Signification
♥	Sera fait en classe, ou à faire en priorité
♡	Autre exercice intéressant mais moins prioritaire

Symbole	Difficulté de l'exercice
☆	Standard
⭒	Plus difficile
★	Difficile ou très difficile

Symbole	Type d'exercice
⚙	Exécution mécanique d'un concept
✎	Conception de machine, algorithme, etc.
🔍	Analyse de machine, algorithme, etc.
≤	Réduction ou simulation
■	Démonstration mathématique
🎓	Matériel avancé qui peut dépasser le cadre du cours

Table des matières

Fondements

1

Calculabilité

1.1 Problèmes et langages

Considérons ce problème algorithmique simple:

ENTRÉE: $k \in \mathbb{N}$
QUESTION: est-ce que k est impair?

La résolution du problème dépend de la représentation de k. Par exemple, considérons trois représentations classiques: unaire; binaire avec bit de poids faible à droite; et binaire avec bit de poids faible à gauche. Les entiers naturels impairs $\{1, 3, 5, 7, 9, 11, 13, \ldots\}$ sont représentés respectivement comme suit:

$$X \coloneqq \{1, 111, 11111, 1111111, 111111111, 11111111111, 1111111111111, \ldots\},$$
$$Y \coloneqq \{1, 11, 101, 111, 1001, 1011, 1101, \ldots\},$$
$$Z \coloneqq \{1, 11, 101, 111, 1001, 1101, 1011, \ldots\}.$$

Nous appelons ces trois ensembles des langages. En théorie du calcul, un langage L représente un problème de décision: un algorithme qui résout L doit retourner `vrai` lorsque $w \in L$, et `faux` lorsque $w \notin L$.

Définissons formellement cette notion. Un *alphabet* est un ensemble fini Σ dont les éléments sont appelés des *lettres*. Un *mot* est une suite finie de lettres. Le *mot vide* est dénoté ε. La *concaténation* de deux mots u et v s'écrit $u \cdot v$ ou simplement uv.

La taille (ou longueur) d'un mot w est dénotée $|w|$. En particulier, l'unique mot de taille zéro est ε. Nous écrivons $|w|_\sigma$ afin de dénoter le nombre d'occurrence de la lettre σ dans w. Nous écrivons w_i afin de dénoter la $i^{\text{ème}}$ lettre du mot w (en comptant à partir de 1).

Exemple.

Considérons l'alphabet $\Sigma \coloneqq \{a, b, c\}$ et le mot $w \coloneqq bab$. Nous avons $|w| = 3$, $|w|_a = 1$, $|w|_b = 2$, $|w|_c = 0$, $w_1 = b$, $w_2 = a$ et $w_3 = b$.

Nous écrivons Σ^i afin de dénoter l'ensemble des mots de longueur i. Nous définissons

$$\Sigma^* \coloneqq \bigcup_{i \geq 0} \Sigma^i \quad \text{et} \quad \Sigma^+ \coloneqq \bigcup_{i > 0} \Sigma^i.$$

Autrement dit, Σ^* est l'ensemble de tous les mots, et $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Un *langage* est un ensemble de mots $L \subseteq \Sigma^*$. Un langage peut être fini ou infini. Par exemple, les ensembles $X \subseteq \{1\}^*$, $Y \subseteq \{0,1\}^*$ et $Z \subseteq \{0,1\}^*$ définis précédemment pour le problème de parité sont des langages infinis.

Un langage $L \subseteq \Sigma^*$ modélise un problème de décision comme suit: Un algorithme qui résout L reçoit une entrée $w \in \Sigma^*$ (pré-condition) et retourne `vrai` ssi $w \in L$ (post-condition).

1.2 Automates

Un problème se résout à l'aide d'un algorithme. Les algorithmes sont souvent décrits à l'aide de pseudocode. Or, cela dissimule des détails techniques qui peuvent avoir un impact notable, par exemple, sur le temps d'exécution. Nous cherchons donc à raisonner dans un modèle de calcul formel.

Un modèle très simple est celui des automates. Formellement, un *automate* est un quintuplet $(Q, \Sigma, \delta, q_0, F)$ où

— Q est un ensemble fini dont les éléments se nomment *états*;

— Σ est un alphabet;

— $\delta \colon Q \times \Sigma \to Q$ est une fonction dite de *transition*;

— $q_0 \in Q$ est l'*état initial*; et

— $F \subseteq Q$ est l'ensemble des états acceptants.

Un automate reçoit un mot $w \in \Sigma^*$ en entrée et débute dans l'état initial q_0. Il lit ensuite les lettres de w de gauche à droite (une et une seule fois). Lorsque la lettre w_i est lue, l'automate modifie l'état actuel q par l'état $\delta(q, w_i)$.

Nous disons qu'un automate $\mathcal{A}$ accepte un mot w ssi sa lecture termine dans un état acceptant. Nous disons que $\mathcal{A}$ *décide* le langage

$$\mathrm{L}(\mathcal{A}) \coloneqq \{w \in \Sigma^* : \mathcal{A} \text{ accepte } w\}.$$

Intuitivement, $\mathcal{A}$ est un « algorithme » qui résout le « problème » $\mathrm{L}(\mathcal{A})$.

Reconsidérons le problème où l'on cherche à déterminer si un nombre $k \in \mathbb{N}$ est impair. Les langages X, Y et Z décrits précédemment sont décidés respectivement par les automates $\mathcal{A}$, $\mathcal{B}$ et $\mathcal{C}$ de la figure 1.1. Ces automates correspondent environ aux programmes de la figure 1.2, où l'on imagine w comme un pointeur vers une chaîne.

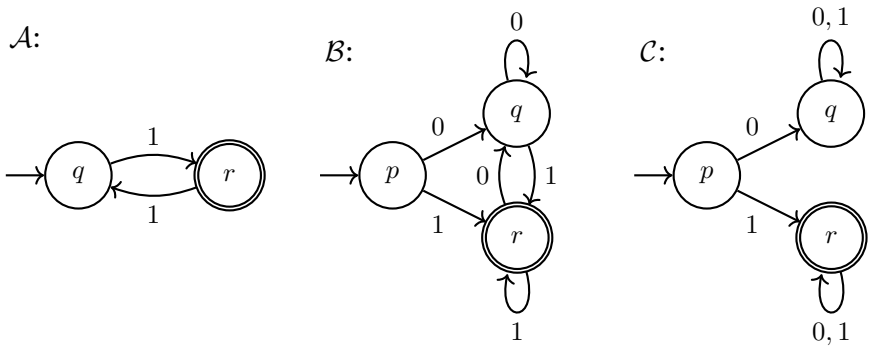


FIGURE 1.1 – Automates pour le problème de parité où le nombre en entrée est représenté en: unaire; binaire avec bit de poids faible à droite; et binaire avec bit de poids faible à gauche.

```
// A
init:
pair:   w++;
impair: w++; goto pair;

// B
init:   if (*(w++) == 1) { goto impair; }
pair:   if (*(w++) == 0) { goto pair; }
impair: if (*(w++) == 1) { goto impair; } else { goto pair; }

// C
init:   if (*(w++) == 1) { goto impair; }
pair:   goto pair;
impair: goto impair;
```

FIGURE 1.2 – Code qui correspond environ aux automates de la figure 1.1.

Contrairement aux ordinateurs, les automates ne possèdent pas de mémoire principale. En effet, leur unique mémoire est l'état actuel (qui correspond informellement à la ligne de code actuelle). Ainsi, la taille de la mémoire d'un automate est constante et par conséquent indépendante de la taille de l'entrée. Très peu de problèmes peuvent être résolus avec une mémoire de $\mathcal{O}(1)$.

Par exemple, on peut montrer qu'il n'existe aucun automate qui décide le langage $\{u\#u : u \in \{0,1\}^*\}$. Ce langage correspond à un problème où l'on doit déterminer si deux mots (séparés par un délimiteur) sont identiques.

1.3 Machines de Turing

Notre modèle de calcul principal sera la machine de Turing. Il s'agit essentiellement d'un automate avec une mémoire principale. Un graphe qui s'apparente à un automate décrit l'*unité de contrôle*, et la mémoire s'appelle le *ruban* (voir la figure 1.3). Intuitivement, l'unité de contrôle représente le code d'un programme et le ruban représente la mémoire principale.

Une machine possède une tête de lecture/écriture qui débute à la première case (à gauche) du ruban. Une machine peut lire et écrire sur son ruban; déplacer sa tête vers la gauche ou la droite; et modifier l'état de l'unité de contrôle.

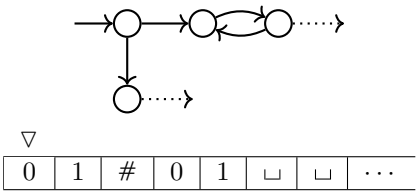


FIGURE 1.3 – Représentation visuelle d'une machine de Turing: unité de contrôle (haut) et ruban avec tête (bas).

Formellement, une *machine de Turing* est un septuplet $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ où

— Q est un ensemble fini dont les éléments se nomment *états*;
— Σ est un alphabet dit *d'entrée*;
— Γ est un alphabet dit *de travail* qui satisfait $\Sigma \subseteq \Gamma$ et $\sqcup \in \Gamma \setminus \Sigma$;
— $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{G}, \mathsf{D}\}$ est une fonction dite de *transition*;
— $q_0 \in Q$ est l'*état initial*;
— $q_{\text{acc}} \in Q$ est l'*état acceptant*; et
— $q_{\text{rej}} \in Q$ est l'*état rejetant*.

1.3.1 Fonctionnement

Une machine de Turing reçoit un mot $w \in \Sigma^*$ en entrée et débute dans l'état initial q_0. À droite de w, il y a une suite infinie constituée uniquement de la lettre « $\sqcup$ » qu'on appelle le « blanc ». La tête (de lecture/écriture) de la machine débute sur la première lettre w_1. À chaque étape, la machine

— consulte son état actuel q;
— lit la lettre σ actuellement sous la tête;
— consulte $\delta(q, \sigma) = (q', \sigma', d)$;
— remplace la lettre sous la tête par σ';
— déplace la tête dans la direction $d \in \{\mathsf{G}, \mathsf{D}\}$, où G et D dénotent respectivement la gauche et la droite; et

— remplace son état actuel par q'.

Nous considérons chaque telle étape comme une seule opération élémentaire. Lorsque la machine entre dans l'état q_{acc} (resp. q_{rej}), elle s'arrête et *accepte* (resp. *rejette*) l'entrée w.

Il y a deux cas limites à considérer:

— si la tête tente de se déplacer à gauche sur la première case, alors la transition a lieu mais la tête reste sur place;

— si le mot en entrée est $w = \varepsilon$, alors la tête débute sur la première occurrence de $\sqcup$.

Nous disons qu'une machine de Turing $\mathcal{M}$ accepte le mot w ssi elle atteint finalement l'état q_{acc} sur entrée w. Nous disons que $\mathcal{M}$ *reconnaît* le langage

$$\mathrm{L}(\mathcal{M}) \coloneqq \{w \in \Sigma^* : \mathcal{M} \text{ accepte } w\}.$$

Remarquons qu'une machine peut boucler à l'infini sur certaines entrées. Nous disons que $\mathcal{M}$ *termine* si, sur chaque entrée, $\mathcal{M}$ atteint finalement q_{acc} ou q_{rej}. Si $\mathcal{M}$ termine, alors nous disons qu'elle *décide* $\mathrm{L}(\mathcal{M})$. Intuitivement, une telle machine $\mathcal{M}$ est un « algorithme » qui résout le « problème » $\mathrm{L}(\mathcal{M})$.

1.3.2 Un premier exemple

Considérons le langage $L \coloneqq \{u\#u : u \in \{0,1\}^*\}$ mentionné à la section 1.2. Rappelons qu'il n'existe aucun automate qui décide L. Néanmoins, il est possible de décider ce langage avec la machine de Turing de la figure 1.4 dont l'approche est décrite à l'algorithme 1.

Algorithme 1 : Description d'une machine pour $\{u\#u : u \in \{0,1\}^*\}$.

Entrées : $w \in \{0,1,\#\}^*$
Sorties : w est de la forme $u\#u$ où $u \in \{0,1\}^*$?
lire σ
tant que $\sigma \neq \#$
  remplacer σ par x
  `// Gérer la prochaine lettre non marquée du côté droit`
  aller à droite jusqu'à la première lettre $\sigma' \neq x$ après $\#$
  si $\sigma \neq \sigma'$ **alors** **retourner** `faux`
  sinon **remplacer** σ' par x
  `// Gérer la prochaine lettre non marquée du côté gauche`
  aller à gauche jusqu'à $\#$
  aller à gauche jusqu'au premier x
  aller à droite et **lire** σ
`// Vérifier que l'entièreté du côté droit a été géré`
retourner `vrai` *ssi toutes les lettres à droite jusqu'à* $\sqcup$ *sont des* x

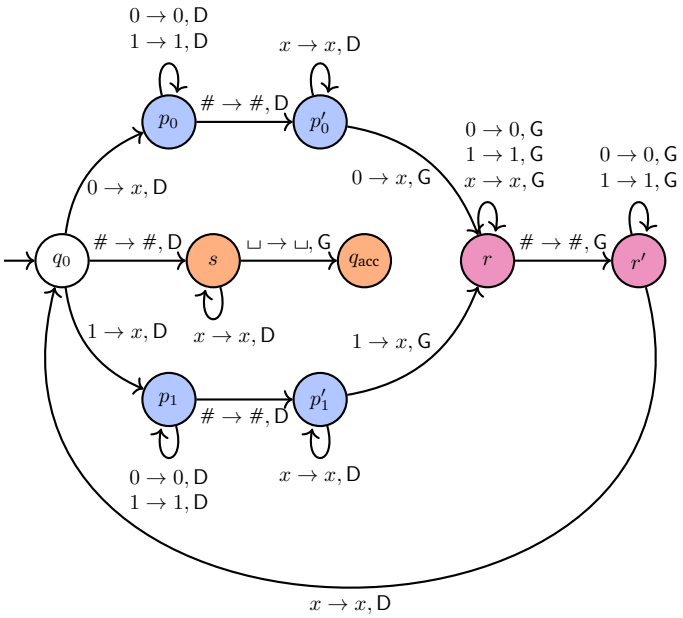
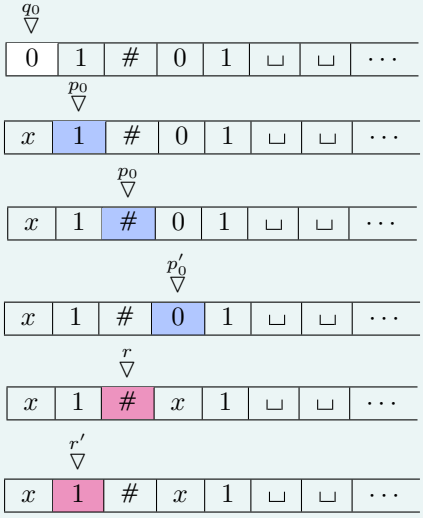
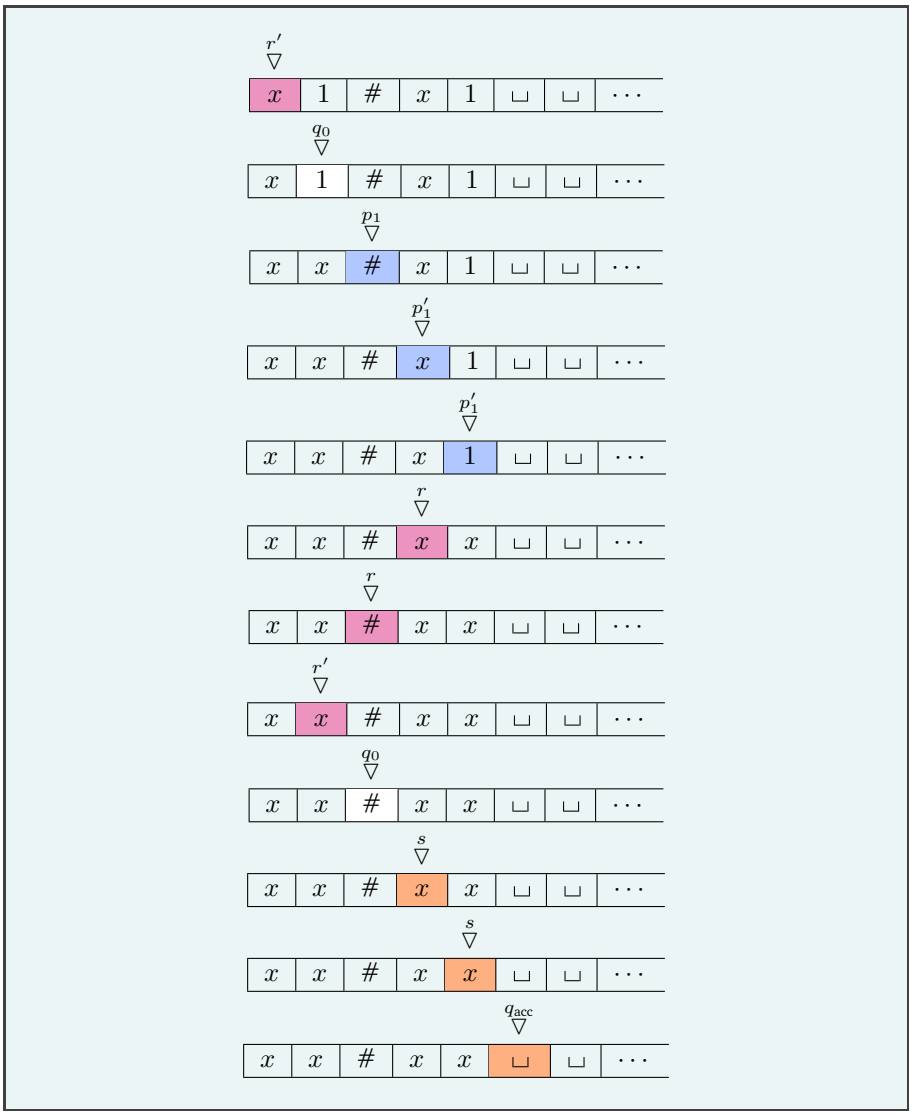


FIGURE 1.4 – Machine de Turing qui décide $\{u\#u : u \in \{0,1\}^*\}$. Pour alléger le diagramme, les transitions qui n'apparaissent pas mènent à l'état rejetant. Les noms des états sont assez arbitraires et pourraient être nommés autrement.

Exemple.

Voici la trace de la machine sur l'entrée $w := 01\#01$:





1.3.3 Configurations et temps d'exécution

À la section 1.3.1, nous avons expliqué intuitivement comment fonctionne une machine de Turing. Soyons maintenant plus formels. Une *configuration* d'une machine $\mathcal{M}$ est une suite uqv où $u, v \in \Gamma^*$ et $q \in Q$. Une telle configuration indique que $\mathcal{M}$ est actuellement dans l'état q; que son ruban contient $uv\sqcup\sqcup\cdots$; et que sa tête pointe sur la première lettre de v.

La *configuration initiale* sur l'entrée $w \in \Sigma^*$ est $\varepsilon q_0 w$. Une configuration uqv est *acceptante* si $q = q_{\text{acc}}$; *rejetante* si $q = q_{\text{rej}}$; et *terminale* si $q \in \{q_{\text{acc}}, q_{\text{rej}}\}$.

Nous écrivons $C \to C'$ si l'exécution d'une transition dans la configuration C mène à la configuration C'. Plus formellement, pour tout $u, v \in \Gamma^*$, $a, b \in \Gamma$ et $q \in Q$, la relation de transition est définie par ces règles:

$$\begin{aligned} u\ q\ bv &\to ub'\ q'\ v \quad \text{si } \delta(q,b) = (q', b', \mathsf{D}), \\ ua\ q\ bv &\to u\ q'\ ab'v \text{ si } \delta(q,b) = (q', b', \mathsf{G}), \\ \varepsilon\ q\ bv &\to q'\ b'v \quad \text{si } \delta(q,b) = (q', b', \mathsf{G}). \end{aligned}$$

Nous écrivons $C \to^* C'$ si $C = C'$ ou s'il existe des configurations $C_0, \ldots, C_k$ telles que $C = C_0 \to C_1 \to \cdots \to C_k = C'$.

Nous disons qu'une machine de Turing $\mathcal{M}$ *accepte* le mot $w \in \Sigma^*$ s'il existe une configuration acceptante C telle que $\varepsilon q_0 w \to^* C$. Nous disons que $\mathcal{M}$ *termine* si pour tout $w \in \Sigma^*$, il existe une configuration terminale C telle que $\varepsilon q_0 w \to^* C$.

Considérons une machine de Turing $\mathcal{M}$ qui termine. Étant donné $w \in \Sigma^*$, nous écrivons $f(w)$ afin de dénoter le nombre de transitions franchies par $\mathcal{M}$ avant d'atteindre une configuration terminale. Le *temps d'exécution (dans le pire cas)* de $\mathcal{M}$ est la fonction $t_{\max} \colon \mathbb{N} \to \mathbb{N}$ définie par

$$t_{\max}(n) \coloneqq \max\{f(w) : w \in \Sigma^n\}.$$

1.3.4 Un deuxième exemple

Considérons le problème qui consiste à déterminer si un nombre $k \in \mathbb{N}$ est une puissance de deux. On peut le résoudre avec l'algorithme 2. Or, cet algorithme est de trop haut niveau; il ne se soucie pas de la représentation de k.

Algorithme 2 : Identification des puissances de deux.

Entrées : $k \in \mathbb{N}$
Sorties : k est une puissance de deux?

si $k = 0$ **alors retourner** `faux`

tant que $k \neq 1$
  si k *est impair* **alors retourner** `faux`
  sinon

retourner `vrai`

Considérons le cas où k est représenté en unaire. Cela correspond au langage $\{1, 11, 1111, 11111111, 1111111111111111, \ldots\}$. L'algorithme 3 décrit la procédure précédente de façon plus bas niveau. La machine de Turing $\mathcal{M}$ illustrée à la figure 1.5 implémente les détails.

Lorsque $\mathcal{M}$ est dans l'état $q_{i,j}$, cela indique que le nombre unaire parcouru jusqu'ici est pair si $j = 0$, et impair si $j = 1$. De plus, i indique si ce nombre est égal à ou strictement plus grand que 1.

Analysons $\mathcal{M}$. La phase qui identifie la parité et divise le nombre en deux fonctionne en temps $\mathcal{O}(n)$, puisqu'on balaie entièrement le ruban. Notons que la

Algorithme 3 : Identification des puissance de deux en unaire.

Entrées : $w \in \{1\}^*$
Sorties : $|w|$ est une puissance de deux?

si *la première lettre est* $\sqcup$ **alors retourner** `faux`
écrire $\dot{1}$ pour marquer le début du ruban
tant que *qu'il reste des* 1
  `// Diviser le nombre en deux`
  remplacer un 1 sur deux avec x et mémoriser la parité
  si *le nombre est impair* **alors retourner** `faux`
  `// Rembobiner`
  ramener la tête au début du ruban
retourner `vrai`

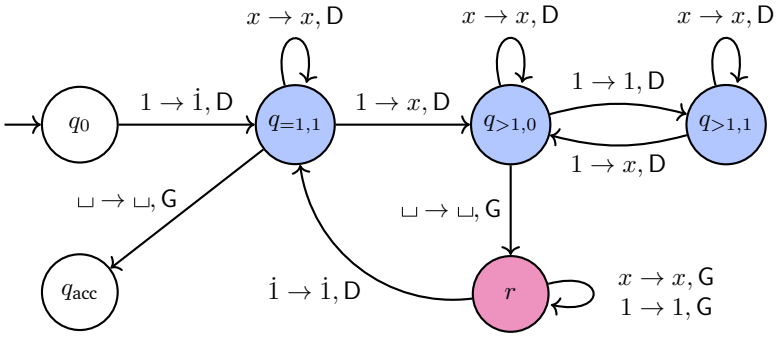


FIGURE 1.5 – Machine de Turing qui détermine si un entier unaire est une puissance de deux. Pour alléger le diagramme, les transitions qui n'apparaissent pas mènent à l'état rejetant.

taille du mot ne change jamais. De plus, il est impossible de diviser un nombre par deux plus de $\log n$ fois. Ainsi, le temps d'exécution appartient à $\mathcal{O}(n \log n)$.

1.4 Variantes de machines de Turing

Les machines de Turing forment un modèle de calcul de bas niveau. Leur simplicité permet de les analyser mathématiquement et de les utiliser comme modèle de référence. Par contre, les machines de Turing sont assez pénibles à programmer. Pour cette raison, nous allons étendre le modèle avec d'autres primitives. Par exemple, il sera pratique de permettre les déplacements stationnaires (dénotés « S »). Cela est simple à simuler (voir exercice 1.1)). Dans cette section, nous considérons deux autres extensions.

1.4.1 Rubans multiples

Une *machine de Turing à k rubans* $\mathcal{M}$ est une machine de Turing dont la fonction de transition est de la forme $\delta\colon Q\times\Gamma^k \to Q\times\Gamma^k\times\{\mathsf{G},\mathsf{S},\mathsf{D}\}^k$, où « S » désigne un mouvement stationnaire. Sur entrée $w\in\Sigma^*$, $\mathcal{M}$ débute avec w sur son premier ruban, alors que les $k-1$ autres rubans contiennent $\sqcup\sqcup\cdots$. Chaque transition dépend de l'état actuel et du contenu des lettres $(a_1,\ldots,a_k)$ sous les k têtes. Lors de l'exécution d'une transition, les k têtes ne sont pas forcées de faire la même écriture ou de se déplacer dans la même direction; elles sont indépendantes.

L'ajout de rubans n'augmente pas l'expressivité du modèle. Autrement dit, toute machine à plusieurs rubans peut être simulée par une machine standard. De plus, le temps d'exécution n'est affecté qu'à un polynôme près.

Théorème 1. *Soit $\mathcal{M}$ une machine de Turing à k rubans. Il existe une machine de Turing $\mathcal{M}'$ telle que $L(\mathcal{M}')=L(\mathcal{M})$. De plus, si $\mathcal{M}$ termine et fonctionne en temps $\mathcal{O}(t(n))$, alors $\mathcal{M}'$ termine et fonctionne en temps $\mathcal{O}(n+t(n)^2)$.*

⚠ En construction.

Ajouter la preuve.

Pour l'instant, consultez le théorème 3.13 du livre de Sipser [Sip06, p. 149].

1.4.2 Ruban bi-infini

Une *machine de Turing à ruban bi-infini* est une machine de Turing dont le ruban est infini dans les deux directions. Initialement, le ruban contient $\cdots\sqcup\sqcup w\sqcup\sqcup\cdots$ et la tête se situe au début de w. Ainsi, contrairement au modèle standard, un déplacement vers la gauche ne peut jamais « rebondir ».

L'usage d'un tel ruban n'augmente pas l'expressivité du modèle. Autrement dit, toute machine à ruban bi-infini peut être simulée par une machine standard. De plus, le temps d'exécution n'est pas affecté asymptotiquement.

Théorème 2. *Soit $\mathcal{M}$ une machine de Turing à ruban bi-infini. Il existe une machine de Turing $\mathcal{M}'$ telle que $L(\mathcal{M}')=L(\mathcal{M})$. De plus, si $\mathcal{M}$ termine et fonctionne en temps $\mathcal{O}(t)$, alors $\mathcal{M}'$ termine et fonctionne en temps $\mathcal{O}(t)$.*

Démonstration. Une première approche consiste à insérer la lettre \$ afin de marquer le début; puis faire exactement comme $\mathcal{M}$ sauf dans un cas: lorsqu'on doit se déplacer à gauche à partir de \$, on insère la lettre à écrire en décalant le contenu du ruban vers la droite (voir exercice 1.2)). Toutefois, cette approche fonctionne en temps $\mathcal{O}(t(n)^2)$, par ex. si $\mathcal{M}$ se déplace toujours vers la gauche, alors $\mathcal{M}'$ doit incessamment décaler son ruban.

Nous présentons donc une deuxième approche. Soit $\mathcal{M}=(Q,\Sigma,\Gamma,\delta,q_0,q_{\text{acc}},q_{\text{rej}})$. Décrivons $\mathcal{M}'=(Q',\Sigma,\Gamma',\delta',q_0',q_{\text{acc}},q_{\text{rej}})$.

On peut imaginer les rubans de $\mathcal{M}$ et $\mathcal{M}'$ comme des cases dont les indices appartiennent respectivement à $\{\ldots,-2,-1,1,2,\ldots\}$ et $\{0,1,2,\ldots\}$. Initialement la machine $\mathcal{M}'$ prend son entrée $w \in \Sigma^n$ et la convertit vers

$$\$ \begin{smallmatrix} w_1 \\ \sqcup \end{smallmatrix} \begin{smallmatrix} w_2 \\ \sqcup \end{smallmatrix} \cdots \begin{smallmatrix} w_n \\ \sqcup \end{smallmatrix} \sqcup \sqcup \cdots .$$

La lettre $\$$ à la position 0 identifie le début du ruban. La case $i \in \mathbb{N}_{\geq 1}$ représente simultanément les cases i et $-i$ du ruban de $\mathcal{M}$ comme suit:

— si la case contient une lettre de la forme $\begin{smallmatrix} a \\ b \end{smallmatrix}$, alors cela indique que les cases i et $-i$ de $\mathcal{M}$ contiennent respectivement a et b;

— si la case contient $\sqcup$, alors les cases i et $-i$ de $\mathcal{M}$ contiennent chacune $\sqcup$.

Formellement, nous posons $\Gamma' \coloneqq \Gamma \cup \{\begin{smallmatrix} a \\ b \end{smallmatrix} : a \in \Gamma, b \in \Gamma\} \cup \{\$\}$.

Afin de simuler les transitions de $\mathcal{M}$, la machine $\mathcal{M}'$ possède un « mode » qui indique si elle inspecte les lettres du haut (côté positif du ruban) ou les lettres du bas (côté négatif du ruban). Lorsque $\mathcal{M}'$ est dans le mode « négatif », elle effectue les déplacements dans le sens inverse de $\mathcal{M}$.

De façon plus détaillée:

— Chaque état q de $\mathcal{M}$ est remplacé par deux états q^+ et q^-;

— Après avoir transformé son entrée, la machine $\mathcal{M}'$ débute dans l'état q_0^+;

— Si $\mathcal{M}'$ est dans un état q^+ et lit $\begin{smallmatrix} a \\ b \end{smallmatrix}$, alors elle fait comme $\delta(q,a)$;

— Si $\mathcal{M}'$ est dans un état q^- et lit $\begin{smallmatrix} a \\ b \end{smallmatrix}$, alors elle fait comme $\delta(q,b)$ sauf pour le déplacement qui est renversé;

— Si $\mathcal{M}'$ est dans un état q^+ et lit $\$$, alors elle se déplace à droite dans q^-;

— Si $\mathcal{M}'$ est dans un état q^- et lit $\$$, alors elle se déplace à gauche dans q^+;

— Si $\mathcal{M}'$ est dans un état $q^\pm$ et lit $\sqcup$, alors elle écrit $\begin{smallmatrix} \sqcup \\ \sqcup \end{smallmatrix}$ et demeure stationnaire.

Finalement, la machine $\mathcal{M}'$ accepte lorsqu'elle atteint q^+_{acc} ou q^-_{acc}, et rejette lorsqu'elle atteint q^+_{rej} ou q^-_{rej}. □

1.5 Primitives de haut niveau

Dans cette section, nous allons nous convaincre que les algorithmes, décrits sous pseudocode ou dans un langage de programmation de haut niveau, peuvent être implémentés sur une machine de Turing, et ce sans trop affecter le temps d'exécution (à un polynôme près). Pour ce faire, nous allons introduire des primitives de plus en plus haut niveau.

Supposons sans perte de généralité que chaque ruban de nos machines débute par un marqueur spécial $\$$. Nous allons également utiliser le symbole « $\curvearrowleft$ » afin de spécifier un retour de la tête jusqu'à lettre à droite de $\$$. Une telle sous-routine, qui fonctionne en temps linéaire, s'implémente facilement.

Afin d'alléger la notation, nous écrivons « $\sigma \to d$ » afin de dénoter « $\sigma \to \sigma, d$ ». De plus, une transition sans étiquette indique que peu importe la lettre sous la tête, on n'écrit rien et on ne déplace pas la tête.

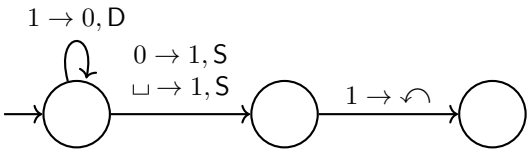
1.5.1 Variables

Afin de représenter une *variable (statique)*, nous utilisons simplement un ruban additionnel. Comme nous avons vu à la section 1.4.1, cela est permis car l'ajout de rubans ne change pas l'expressivité du modèle.

Par défaut, une variable entière non négative $x \in \mathbb{N}$ sera représentée en binaire. Il faut choisir une convention: bits de poids faible à gauche ou à droite. Comme on peut aisément renverser le contenu d'un ruban en temps polynomial, nous pourrons décrire les primitives pour l'une des deux conventions seulement.

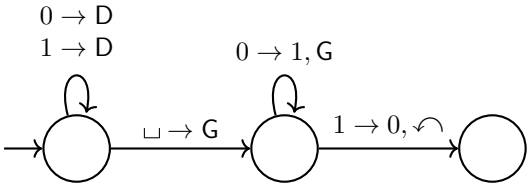
1.5.2 Addition et soustraction

L'incrémentation d'un entier non négatif x, dénotée $x{+}{+}$, s'effectue comme suit en temps linéaire, pour la convention avec bit de poids faible à gauche:



Par exemple, $011\text{␣␣}\cdots$ (6) devient $111\text{␣␣}\cdots$ (7), qui devient $0001\text{␣}\cdots$ (8).

La décrémentation d'un entier positif x, dénotée $x{-}{-}$, s'effectue comme suit en temps linéaire, pour la convention avec bit de poids faible à droite:



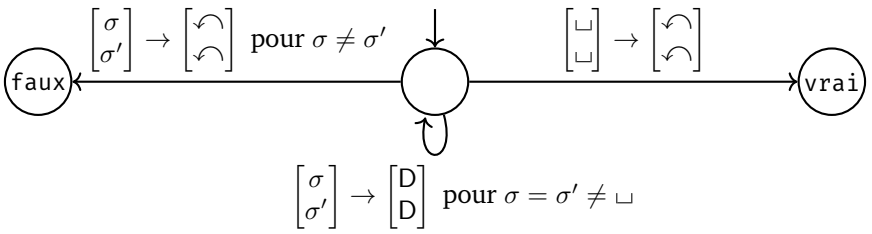
Par exemple, $110\text{␣}\cdots$ (6) devient $101\text{␣}\cdots$ (5), qui devient $100\text{␣}\cdots$ (4).

Considérons deux variables entières $x, y \in \mathbb{N}$ et un ruban vide pour z. L'addition « $z \leftarrow x + y$ » s'implémente en temps linéaire tel que décrit en annexe. ↑↓

Afin d'implémenter la soustraction, une option plutôt simple est d'introduire les entiers négatifs avec représentation par complément à deux.

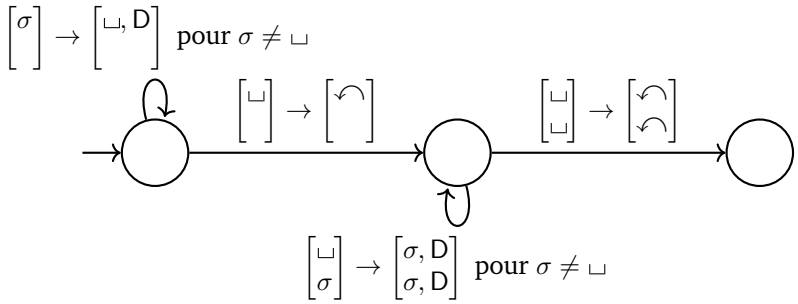
1.5.3 Comparaison et copie

La comparaison pure « $x = y$ » s'implémente ainsi:



Lorsque x et y sont des entiers, la comparaison pure est trop stricte: il faut ignorer les zéros non significatifs en considérant que $\sqcup$ et 0 sont équivalents.

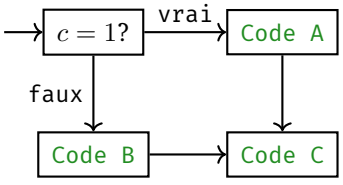
La copie « $x \leftarrow y$ » s'effectue en temps linéaire en effaçant le ruban pour x; en ramenant sa tête au début; en copiant y sur x; puis en ramenent les deux têtes au début:



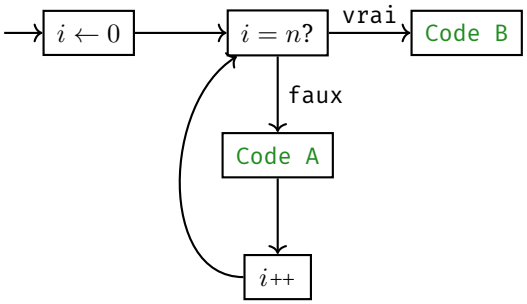
1.5.4 Sélection et itération

Les structures de contrôle de haut niveau s'implémentent en composant les primitives déjà établies. Nous en répertorions quelques-unes, où c dénote une variable booléenne représentée par un bit.

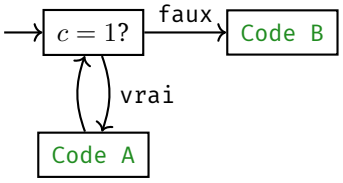
si c **alors**
  `// Code A`
sinon
  `// Code B`
`// Code C`



pour $i \in [0..n-1]$
  `// Code A`
`// Code B`

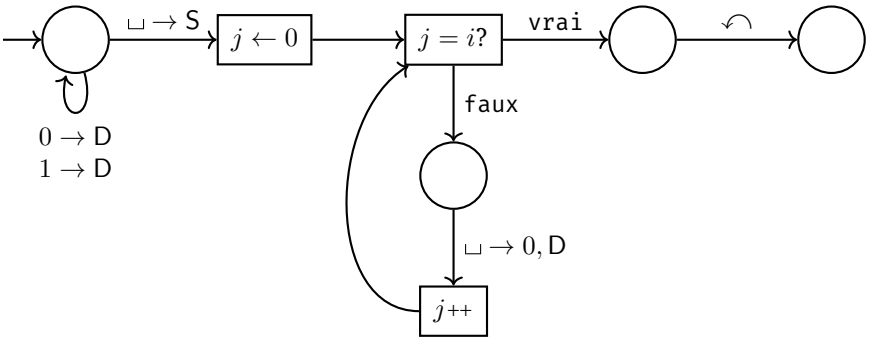


tant que c
  `// Code A`
`// Code B`

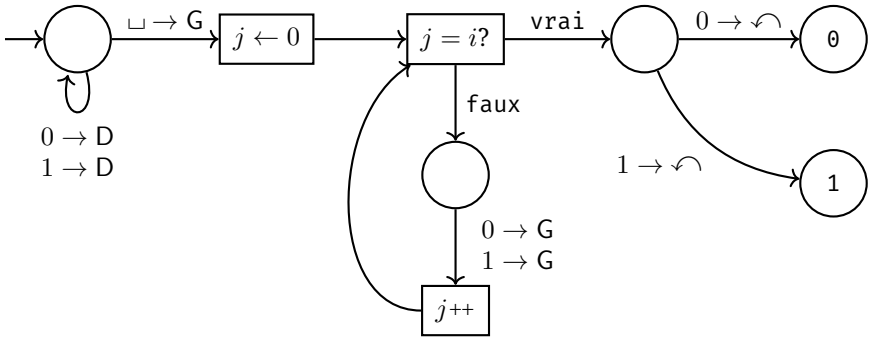


1.5.5 Arithmétique

Afin d'implémenter d'autres opérations arithmétiques, voyons d'abord deux primitives utiles. Considérons la convention avec bit de poids faible à droite. Dénotons « $x \xleftarrow{<<} i$ » l'opération « $x \leftarrow 2^i \cdot x$ ». Celle-ci s'implémente en insérant i occurrences de 0 à droite de x:



Nous pouvons accéder au $i^{\text{ème}}$ bit de x, où le bit 0 est le bit le moins significatif:



On peut implémenter la multiplication « $z \leftarrow x \cdot y$ » avec l'algorithme élémentaire classique, c'est-à-dire celui qui calcule $13 \cdot 11$ ainsi:

$$
\begin{array}{rll}
 & 1101 & (13) \\
\times & 1011 & (11) \\
\hline
 & 1101 & \\
 & 1101 & \\
+ & 0000 & \\
 & 1101 & \\
\hline
 & 10001111 & (143)
\end{array}
$$

En général, cet algorithme correspond au pseudocode suivant, qui peut être traduit dans nos primitives établies:

```
// Calcul de z ← x · y
z ← 0
pour i ∈ [0..n − 1]
    si y[i] = 1 alors
        // Calculer z ← z + 2^i · x
        b ← x
        b ←<< i
        a ← z
        z ← a + t
```

On peut adapter l'algorithme classique de division de façon similaire. L'opération « $z \leftarrow x \bmod y$ » s'obtient ensuite en traduisant ce pseudocode:

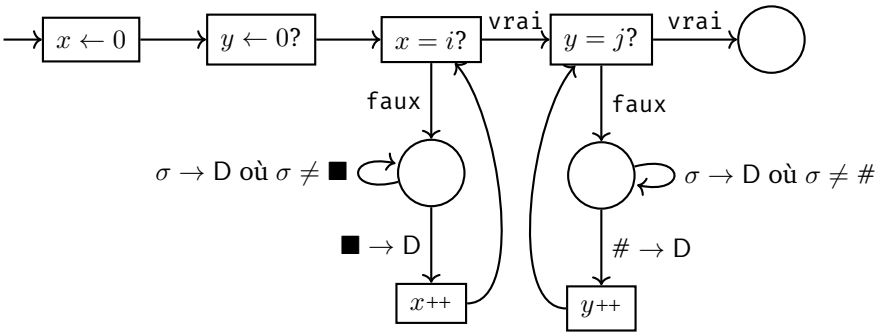
```
// Calcul de z ← x mod y
z ← x ÷ y
a ← z
z ← a · y
b ← z
z ← x − b
```

1.5.6 Tableaux et matrices

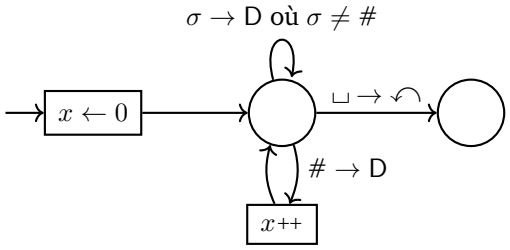
Afin d'implémenter les tableaux (dynamiques), on peut simplement utiliser un délimiteur # afin de séparer les éléments. Plus généralement, pour une matrice, on utilise le délimiteur ■ afin de séparer les lignes. Ainsi,

$$\mathbf{A} = \begin{pmatrix} 2 & 6 \\ 1 & 4 \end{pmatrix} \text{ est représentée par } \$\ 010\ \#\ 110\ \blacksquare\ 001\ \#\ 100\ \blacksquare\ \sqcup\ \sqcup\ \cdots .$$

On peut obtenir $\mathbf{A}[i, j]$, où i et j débutent à 0, comme suit:



Voyons également comment obtenir la taille d'un tableau unidimensionnel:



1.5.7 Exemple de haut niveau: accessibilité dans les graphes

Nous pouvons maintenant résoudre des problèmes de haut niveau. Par exemple, considérons le problème d'accessibilité dans les graphes.

Rappelons qu'un graphe dirigé peut être représenté par une matrice d'adjacence **A** telle que $\mathbf{A}[i, j] \in \{0, 1\}$ vaut 1 si et seulement s'il existe une arête du sommet i vers le sommet j. Nous considérons ce problème:

PATH

ENTRÉE:	un graphe dirigé $G = (V, E)$ décrit par une matrice d'adjacence, et deux sommets $s, t \in V$,
QUESTION:	existe-t-il un chemin de s vers t dans G?

Le problème PATH correspond au langage où la matrice d'adjacence de $\mathcal{G}$ est représentée dans le format de la section 1.5.6, et où les sommets s et t sont des entiers représentés en binaire. Nous résolvons PATH avec cet algorithme dont le pseudocode se traduit dans nos primitives:

Entrées : une matrice d'adjacence **A** et deux sommets s, t
Sorties : s peut atteindre t?

$k \leftarrow |\mathbf{A}|$
$c \leftarrow 1$
$T \leftarrow \overbrace{[0, 0, \ldots, 0]}^{k \text{ fois}}$ // $T[v] = 1$ ssi $s \to^* v$
$T[s] \leftarrow 1$ // Trivialement: $s \to^* s$

tant que c // Explorer tant qu'il y a des changements
  $c \leftarrow 0$
  pour $i \in [0..k-1]$
    pour $j \in [0..k-1]$
      // Si $s \to^* i$ et $i \to j$ alors $s \to^* j$
      si $T[i] = 1 \wedge \mathbf{A}[i, j] = 1 \wedge T[j] = 0$ **alors**
        $T[j] \leftarrow 1$
        $c \leftarrow 1$
retourner $(T[t] = 1)$

Analysons l'algorithme en supposant que toutes les opérations élémentaires s'effectuent en temps $\mathcal{O}(t)$. Soit k le nombre de sommets du graphe. Comme chaque itération de la boucle « **tant que** » marque un sommet, elle ne peut pas être itérée plus de k fois. Ainsi, l'algorithme fonctionne en temps $\mathcal{O}(t + k^3 \cdot t) = \mathcal{O}(k^3 \cdot t)$.

Soit n la taille de l'entrée. Toutes les constructions de bas niveau de la section fonctionnent en temps polynomial. De plus, la composition de deux polynômes est un polynôme. Il existe donc une constante $c \in \mathbb{N}$ telle que $t \in \mathcal{O}(n^c)$. Ainsi, l'algorithme fonctionne en temps $\mathcal{O}(k^3 \cdot n^c)$.

Pour une entrée bien formée, $n \in \Theta(k^2 + \log k)$ puisque la matrice d'adjacence contient k^2 bits et que les deux sommets sont spécifiés par $\mathcal{O}(\log k)$ bits. En particulier, $k \leq n$. Ainsi, l'algorithme fonctionne en temps $\mathcal{O}(n^{3+c})$, ce qui est polynomial.

Remarquons que notre analyse est pour une machine de Turing à plusieurs rubans. Par le théorème 1, nous obtenons une complexité de $\mathcal{O}((n^{3+c})^2) = \mathcal{O}(n^{2c+6})$ sur une machine de Turing standard.

Une telle complexité peut paraître aberrante! En effet, en algorithmique, on considère que l'accessibilité dans les graphes se résout en temps linéaire, par ex. avec un parcours en largeur. Or, dans notre contexte, on cherche à distinguer les ressources exponentielles, polynomiales, et parfois polylogarithmiques (au chapitre 5). Pour cette raison, nous n'avons fait aucun effort à optimiser le code pour réduire le degré du polynôme. Nous obtenons ce résultat:

Proposition 1. *Le problème d'accessibilité dans les graphes dirigés se résout en temps polynomial (sur une machine de Turing).*

Dans le jargon du chapitre 3 à venir, cela pourra également s'écrire:

Proposition 2. PATH $\in$ P.

Nous n'avons pas implémenté les fonctions et la récursion. Nous pourrions y arriver comme sur un ordinateur: avec une pile d'appel. Une pile s'implémente avec un tableau où on insère et retire les éléments à droite (du ruban).

De plus, l'allocation dynamique de mémoire peut s'effectuer avec un tableau stocké sur un ruban.

1.6 Exercices

1.1) ♥☆≤ Expliquez comment simuler les déplacements stationnaires tout en préservant le temps d'exécution asymptotique. ↑↓

1.2) ♥☆✎ Expliquez comment insérer une lettre σ à la position de la tête. Plus précisément, donnez une machine de Turing qui, à partir de la configuration upv, atteint la configuration $u\sigma qv$. Votre machine devrait fonctionner en temps $\mathcal{O}(|v|)$. On suppose que $\sqcup \notin v$. ↑↓

1.3) ♥☆✎ Concevez une machine de Turing qui décide $\{w \in \{0,1\}^* : |w|_0 \leq |w|_1\}$.

1.4) ♥☆✎ Concevez une machine de Turing à deux rubans qui décide $\{u\#u : u \in \{0,1\}^*\}$ en temps linéaire.

1.5) ♡☆✎ Concevez une machine de Turing qui décide $\{0^m 1^m : m \in \mathbb{N}\}$.

1.6) ♡☆≤ Expliquez comment simuler un automate à l'aide d'une machine de Turing.

1.7) ♥★≤ Une *machine à retour* est une machine où les déplacements possibles sont $\{\curvearrowleft, \mathsf{D}\}$, où D dénote un déplacement d'une case à droite, et $\curvearrowleft$ désigne un déplacement vers la toute première case du ruban. Expliquez comment simuler une machine de Turing avec une machine à retour. ↑↓

1.8) ♥★≤ Un *automate à file* est un automate qui a accès à une file. L'automate peut consommer une lettre de la file (à gauche) ou lui ajouter une lettre (à droite). Expliquez comment simuler une machine de Turing avec un automate à file. ↑↓

1.9) ♡★≤ Une *machine sans retour* est une machine de Turing dont les déplacements possibles sont $\{\mathsf{S}, \mathsf{D}\}$. Expliquez comment simuler une machine sans retour avec un automate.

1.10) ♡☆✎ Donnez les détails d'une machine de Turing qui convertit son entrée $w_1 w_2 \cdots w_n$ vers $\$ \overset{w_1}{\sqcup} \overset{w_2}{\sqcup} \cdots \overset{w_n}{\sqcup} \sqcup \sqcup \cdots$.

2

Décidabilité

Couvert par un autre enseignant. Voir les références fournies par cette personne.

3

Complexité du calcul : P vs. NP

Couvert par un autre enseignant. Voir les références fournies par cette personne.

Sujets avancés

4

(In)décidabilité de l'arithmétique

Dans ce chapitre, nous nous penchons sur la vérification automatique d'énoncés mathématiques. Plus précisément, étant donné un énoncé logique φ, est-ce possible pour un ordinateur de vérifier algorithmiquement si φ est vrai?

En logique, les objets considérés peuvent être des entiers, des nombres réels, des mots, des arbres, des graphes, etc. On peut quantifier sur des objets (premier ordre), sur des ensembles (monadique du second ordre), sur des relations (second ordre), etc. Nous nous concentrons sur la logique du premier ordre sur les entiers naturels. Dans ce contexte, nous avons accès à ces « briques » de base:

$$\underbrace{\exists x, \forall x}_{\text{quantification}} \qquad \underbrace{\wedge, \vee, \neg, \dots}_{\text{connecteurs logiques}} \qquad \underbrace{=, \leq, <, \dots}_{\text{comparaison}} \qquad \underbrace{x + y, x \cdot y, 2^x, \dots}_{\text{opérations arithmétiques}} .$$

L'*arithmétique de Peano* est la logique du premier ordre sur les entiers naturels où la comparaison est limitée à $=$, et les opérations arithmétiques sont limitées à l'addition et la multiplication. Nous dénotons cette arithmétique $\mathsf{FO}(\mathbb{N}, =, +, \cdot)$, où FO est de l'anglais « *first order* » pour « premier ordre ».

Par exemple, la formule $\exists y \ (x = y + y)$ spécifie que x est pair; et la formule $\exists z \ ((y = x + z) \wedge \neg(z = 0))$ spécifie que $x < y$. Ici, il est sous-entendu que les variables appartiennent à $\mathbb{N}$.

Exemple.

Voyons un exemple plus complexe. Cette formule affirme qu'il existe une infinité de nombres premiers:

$$\forall n \ \exists p \ [(p > n) \wedge \neg \exists x \ \exists y \ (x > 1 \wedge y > 1 \wedge p = x \cdot y)].$$

En mots, la formule spécifie que pour tout seuil n, il existe un entier p au-delà de ce seuil qui ne peut pas être factorisé.

Bien que nous sachions qu'il existe une infinité de nombres premiers depuis des millénaires, nous ne savons pas à ce jour s'il existe une infinité de nombres

premiers jumeaux. Une paire $(p, q) \in \mathbb{N} \times \mathbb{N}$ est une paire de *nombres premiers jumeaux* si p est premier, q est premier et $q = p+2$. Par exemple, $(5, 7)$ et $(41, 43)$ sont des nombres premiers jumeaux. On conjecture qu'il existe une infinité de tels nombres. Cela s'exprime ainsi dans $\mathsf{FO}(\mathbb{N}, =, +, \cdot)$:

$$\forall n\, \exists p\, \forall x, y\, [(p > n) \wedge ((x > 1 \wedge y > 1) \rightarrow (x \cdot y \neq p \wedge x \cdot y \neq p + 2))].$$

S'il était possible d'automatiser $\mathsf{FO}(\mathbb{N}, =, +, \cdot)$, nous pourrions vérifier ou infirmer la conjecture. Or, comme nous allons le voir, cela est impossible.

4.1 Indécidabilité de l'arithmétique de Peano

Nous disons qu'une formule logique est *close* si elle ne possède aucune variable libre. Par exemple, la formule $\exists x\, \forall y\, (x + y = y)$ est close, mais $\forall y\, (x + y = x)$ ne l'est pas puisque x est libre.

En général, nous disons qu'une logique $\mathcal{L}$ est (in)décidable si son problème de validité est (in)décidable:

ENTRÉE:	une formule close φ de $\mathcal{L}$
QUESTION:	φ est vraie?

L'arithmétique de Peano est indécidable. Pour simplifier les choses, nous allons plutôt prouver que l'*arithmétique de Peano étendue* avec l'exponentiation est indécidable, c.-à-d. $\mathsf{FO}(\mathbb{N}, =, +, \cdot, b^x)$ où b est une constante. La preuve s'adapte à l'arithmétique de Peano sans exponentiation (voir l'exercice 4.11)).

Nous procédons en donnant une réduction à partir du problème d'arrêt:

Proposition 3. *Le problème d'arrêt se réduit (au sens multivoque) au problème de validité de l'arithmétique de Peano étendue.*

4.1.1 Codage

Afin de démontrer la proposition 3, nous devons construire une formule $\varphi_{\mathcal{M},w}$ qui soit vraie si et seulement si une machine de Turing $\mathcal{M}$ donnée s'arrête sur une entrée w donnée. Autrement dit:

$$\langle \mathcal{M}, w \rangle \in A_{\mathsf{TM}} \iff \varphi_{\mathcal{M},w} \equiv \textit{vrai}.$$

Considérons une machine de Turing $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ et un mot $w \in \Sigma^*$. Posons $b := |\Gamma| + |Q| + 1$. Nous allons représenter des suites sur alphabet $\Gamma \cup Q \cup \{\#\}$ en base b, où le chiffre le moins significatif est à gauche. Par exemple, pour $\Gamma = \{0, 1, \sqcup\}$ et $Q = \{q_0, q_1, q_2, q_3\}$, nous pourrions choisir ce codage:

symbole	$\sqcup$	0	1	$\#$	q_0	q_1	q_2	q_3
chiffre	0	1	2	3	4	5	6	7

Ainsi, la configuration $q_0011\sqcup\sqcup\cdots$ serait représentée par

$$41220\,0\cdots_8 = 4\cdot 8^0 + 1\cdot 8^1 + 2\cdot 8^2 + 2\cdot 8^3 + 0\cdot 8^4 + 0\cdot 8^5 + \ldots$$

Le choix du codage est arbitraire, pour autant que $\sqcup$ soit représenté par 0.

Un *historique (de calcul)* est une suite $\#C_0\#C_1\#C_2\#\cdots C_k\sqcup\sqcup\cdots$ telle que C_0 est la configuration initiale de $\mathcal{M}$ sur entrée w, et $C_{i-1}\to C_i$ pour tout $i\in[1..k]$. Un historique se représente par un (possiblement très grand) nombre en base b. Par exemple, dans notre codage précédent, le nombre 3412232522322162_8 représente cet historique:

$$\#q_0011\#1q_111\#10q_21\sqcup\sqcup\cdots\,.$$

Nous allons construire une formule qui vérifie qu'un nombre représente un historique valide qui atteint une configuration terminale. Cette formule sera construite progressivement à partir de « macros ».

4.1.2 Manipulation de nombres

Nous pouvons voir un entier naturel x comme une suite (ou un tableau) dont le premier élément se situe à la position 0. Nous écrivons $x[i]$ afin de dénoter le $i^{\text{ème}}$ élément représenté par x. Par exemple, si $x = 412_8$ est un nombre octal, alors $x[0] = 4$.

Afin de permettre une telle manipulation en base b, introduisons d'abord la division et le modulo. Le quotient $q := x \div y$ et le reste $r := x \bmod y$ sont obtenus par cette formule:

$$\exists q\ \exists r\ [(x = q\cdot y + r) \wedge (r < y)]$$

La valeur $x[i]$ s'obtient par

$$(x \div b^i) \bmod b.$$

et la sous-suite $x[i:j]$ s'obtient par:

$$(x \div b^i) \bmod b^{j-i+1}.$$

Dans nos formules, nous considérons que $x[i]$ dénote le symbole qu'il représente. Par exemple, si le chiffre 3 représente le symbole #, alors nous nous permettons d'écrire $x[i] = \#$ comme sucre syntaxique pour $x[i] = 3$.

4.1.3 Manipulation de configurations

Cherchons maintenant à raisonner sur les configurations d'un historique représenté par un entier en base b.

Nous écrivons « $\exists! i\ \psi(i)$ » afin de dénoter qu'il existe un unique i qui satisfait la formule ψ. Cela se spécifie par:

$$\exists i\ \ [\psi(i) \wedge \forall j\ (j \neq i \to \neg\psi(j))]\,.$$

Étant donné un entier y, nous pouvons déterminer si y code une configuration qui débute par le délimiteur #:

$$\text{config}(y) \coloneqq (y[0] = \#) \wedge (\forall i\ (i > 0) \to (y[i] \neq \#)) \wedge (\exists! j\ y[j] \in Q).$$

L'expression « $y[j] \in Q$ » est du sucre syntaxique. Par exemple, si les états sont $Q = \{q_0, q_1, q_2, q_3\}$ et qu'ils sont respectivement codés par les chiffres 4, 5, 6 et 7, alors « $y[j] \in Q$ » signifie « $(4 \leq y[j]) \wedge (y[j] \leq 7)$ ».

Nous aurons à extraire deux configurations consécutives. Par exemple, considérons un entier x qui représente cette suite:

$$\#q_0011\#1q_111\#10q_21\sqcup\sqcup\cdots .$$

Les indices $i = 0$, $j = 5$ et $k = 10$ délimitent deux configurations successives.

En général, la formule suivante vérifie que les indices i, j et k délimitent deux configurations successives:

$$\begin{aligned}\text{succ}(x, i, j, k) \coloneqq (i < j < k) \wedge \text{config}(x[i : j-1]) \wedge \text{config}(x[j : k-1]) \wedge \\ [(x[k] = \#) \vee (\forall \ell\ (\ell \geq k) \to (x[\ell] = \sqcup))]\,.\end{aligned}$$

Ainsi, dans notre exemple, $\text{succ}(x, 0, 5, 10)$ et $\text{succ}(x, 5, 10, 15)$ sont vraies.

Étant donné un entier x et un indice j, on peut vérifier si $x[0 : j]$ est la toute première configuration de la suite:

$$\begin{aligned}\text{première}(x, j) \coloneqq \text{config}(x[0 : j]) \wedge \\ [(x[j+1] = \#) \vee (\forall \ell\ (\ell > j) \to (x[\ell] = \sqcup))]\,.\end{aligned}$$

Dans notre exemple précédent, $\text{première}(x, 5)$ est vraie.

4.1.4 Manipulation d'historiques

Rappelons que w est l'entrée d'une machine de Turing $\mathcal{M}$. Nous pouvons vérifier que la première configuration représentée par x est la configuration initiale de $\mathcal{M}$ sur entrée w comme suit:

$$\text{init}(x) \coloneqq \exists j\ \text{première}(x, j) \wedge \left(x[1 : j] = q_0 \cdot b^0 + \sum_{i=1}^{|w|} w_i \cdot b^i \right).$$

Ici, nous utilisons à nouveau du sucre syntaxique. En effet, q_0 et w_i sont des symboles, alors que b^0 et b^i sont des entiers, ce qui ne type pas. Or, nous considérons que q_0 et w_i dénotent les chiffres qui les représentent. Par exemple, si q_0 est représenté par le chiffre 4, alors « $q_0 \cdot b^0$ » signifie « $4 \cdot b^0$ ».

Nous définissons $\text{trans}(x, y)$ comme étant une formule qui vérifie si x et y codent des configurations C et C' telles que $C \to C'$. Pour l'implémenter, il faut extraire l'état, la position et les lettres voisines des deux configurations, et vérifier chaque transition de δ à l'aide d'une grande disjonction. Cela est laissé en exercice (voir l'exercice 4.10)).

Nous pouvons déterminer si x code un historique de calcul comme suit:

$$\text{historique}(x) \coloneqq \text{init}(x) \wedge \\ [\forall i, j, k\ \text{succ}(x, i, j, k) \rightarrow \text{trans}(x[i : j-1], x[j : k-1])]\,.$$

4.1.5 La réduction

La formule suivante exprime que la machine de Turing atteint un jour son état acceptant ou rejetant:

$$\text{arrêt}(x) \coloneqq \exists i, j\ \text{config}(x[i : j]) \wedge \\ [\exists k\ (i < k \leq j) \wedge ((x[k] = q_{\text{acc}}) \vee (x[k] = q_{\text{rej}}))]\,.$$

Nous définissons finalement

$$\varphi_{\mathcal{M},w} \coloneqq \exists x\ \text{historique}(x) \wedge \text{arrêt}(x).$$

Par construction, la machine $\mathcal{M}$ s'arrête sur w ssi $\varphi_{\mathcal{M},w} \equiv \texttt{vrai}$. De plus, la description de $\varphi_{\mathcal{M},w}$ est constructive. Ainsi, la réduction est calculable, c.-à-d. qu'il existe une machine de Turing qui calcule $\varphi_{\mathcal{M},w}$ à partir d'une description de $\mathcal{M}$ et w. Cela prouve donc la proposition 3. Par conséquent:

Corollaire 1. *L'arithmétique de Peano étendue est indécidable.*

Démonstration. Rappelons que si un langage A est indécidable et $A \leq_m B$, alors B est indécidable. Ainsi, puisque le problème d'arrêt A_{TM} est indécidable, le problème de validité pour l'arithmétique de Peano étendue est indécidable. □

4.2 Décidabilité de l'arithmétique de Presburger

La plupart des logiques se butent à l'indécidabilité. Or, certaines logiques plus restreintes sont décidables. Nous considérons maintenant l'arithmétique de Presburger: $\mathsf{FO}(\mathbb{N}, =, +)$. Autrement dit, il s'agit de l'arithmétique de Peano sans la multiplication. Nous allons montrer que cette logique est décidable!

Afin de simplifier les choses, nous considérons seulement les formules φ obtenues par cette grammaire:

$$\varphi ::= (V = T) \mid (\varphi \wedge \varphi) \mid (\neg\varphi) \mid (\exists V\ \varphi) \\ T ::= V \mid (V + V)$$

où V est un ensemble de variables, et où les deux côtés d'un terme $V = T$ n'ont pas de variable commune (par ex. $y = x + x$ est permis, mais pas $x = x + y$).

La sémantique, c'est-à-dire le sens associé aux formules, est comme attendue. Par exemple, $y = x + x$ est notamment satisfaite par $x = 6$ et $y = 12$.

Formellement, la sémantique est définie comme suit. Une *interprétation* est une assignation $I : V \to \mathbb{N}$. Nous écrivons $I[x \mapsto n]$ afin de dénoter l'interprétation I' identique à I mais où $I'(x) := n$. Nous écrivons $I \models \varphi$ afin de dénoter que l'interprétation I *satisfait* la formule φ. Cette relation se définit récursivement:

$$\begin{aligned}
I \models (x = y) &\overset{\text{déf}}{\iff} I(x) = I(y),\\
I \models (z = x + y) &\overset{\text{déf}}{\iff} I(z) = I(x) + I(y),\\
I \models (\varphi \wedge \varphi') &\overset{\text{déf}}{\iff} (I \models \varphi) \wedge (I \models \varphi'),\\
I \models (\neg\varphi) &\overset{\text{déf}}{\iff} \neg(I \models \varphi),\\
I \models (\exists x\ \varphi) &\overset{\text{déf}}{\iff} \exists n \in \mathbb{N} : I[x \mapsto n] \models \varphi.
\end{aligned}$$

Écrivons $[\![\varphi]\!] := \{I : I \models \varphi\}$ afin de dénoter l'ensemble des interprétations qui satisfont la formule φ. Par exemple, si $V = \{x, y, z\}$, alors nous avons

$$[\![y = x + x]\!] = \{(a, 2a, b) : a, b \in \mathbb{N}\}.$$

4.2.1 Représentation des solutions

Étant donné une formule φ, nous allons représenter $[\![\varphi]\!]$ à l'aide d'un automate A_φ. Notre but ultime sera de vérifier si une formule close φ est fausse ou non. Pour ce faire, nous aurons simplement à vérifier si $\mathrm{L}(A_\varphi) = \emptyset$.

Posons $\Sigma := \{0, 1\}^V$. Nous considérons un mot $w \in \Sigma^*$ comme la représentation binaire, avec bit de poids faible à gauche, de la valeur des variables. Par exemple, le mot suivant représente $x = 6$ et $y = 12$:

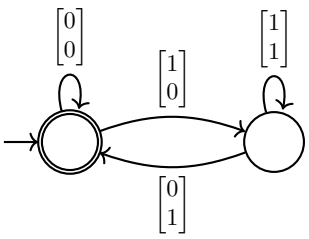
$$w = \begin{bmatrix} 0110 \\ 0011 \end{bmatrix}.$$

Nous considérons que le mot vide représente $x = y = 0$. Remarquons qu'il existe une infinité de représentations équivalentes puisque l'ajout de zéros à droite ne change pas la valeur. Par exemple, $(6, 12)$ se représente par

$$\begin{bmatrix} 0110 \\ 0011 \end{bmatrix}, \begin{bmatrix} 01100 \\ 00110 \end{bmatrix}, \begin{bmatrix} 011000 \\ 001100 \end{bmatrix}, \begin{bmatrix} 0110000 \\ 0011000 \end{bmatrix}, \ldots$$

4.2.2 Formules atomiques

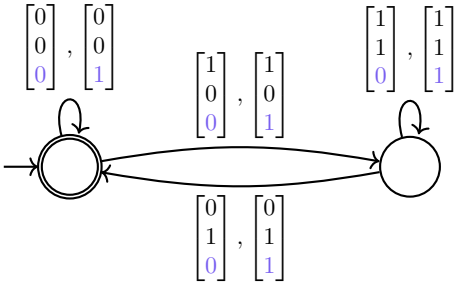
L'automate ci-dessous représente $[\![y = x+x]\!]$, où la première composante dénote un bit de x, et la seconde composante dénote un bit de y:



Par exemple, l'automate accepte $(6, 12)$ puisqu'il accepte le mot

$$\begin{bmatrix} 0110 \\ 0011 \end{bmatrix}.$$

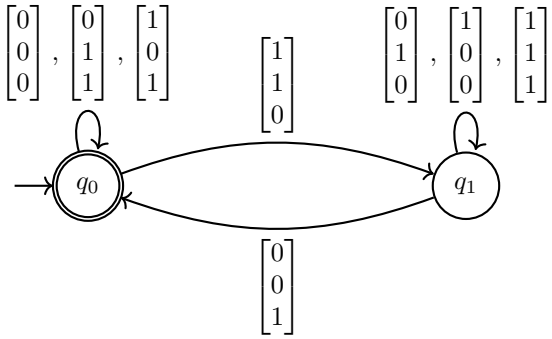
Techniquement, l'automate doit considérer toutes les variables de V. Or, les variables de $V \setminus \{x, y\}$ peuvent prendre n'importe quelle valeur comme elles n'apparaissent pas dans la formule. Afin d'alléger le diagramme, il est donc sous-entendu que tous les bits sont permis dans ces composantes. Par exemple, pour $V = \{x, y, z\}$, nous obtenons cet automate:



Pour alléger les diagrammes, il est également sous-entendu que les transitions qui n'apparaissent pas mènent à un état puits rejetant.

Une formule de la forme $x = y$ se représente par cet automate:

Il nous reste à considérer les formules de la forme $z = x + y$. On utilise un état q_r qui mémorise la dernière retenue r de la somme:



4.2.3 Conjonction et négation

Remarquons que $[\![\varphi \wedge \varphi']\!] = [\![\varphi]\!] \cap [\![\varphi']\!]$. Ainsi, pour représenter l'ensemble $[\![\varphi \wedge \varphi']\!]$, on construit deux automates A et B respectivement pour $[\![\varphi]\!]$ et $[\![\varphi']\!]$, puis on construit un automate C qui satisfait $\mathrm{L}(C) = \mathrm{L}(A) \cap \mathrm{L}(B)$.

Pour ce faire, nous utilisons la construction classique qui consiste à faire le produit de A et B, et les synchroniser sur les lettres lues. Autrement dit, si $p \to^a p'$ est une transition de A, et $q \to^a q'$ est une transition de B, alors $(p, q) \to^a (p', q')$ est une transition de C.

Remarquons que $[\![\neg\varphi]\!] = \overline{[\![\varphi]\!]}$. Ainsi, pour représenter l'ensemble $[\![\neg\varphi]\!]$, on construit un automate A pour $[\![\varphi]\!]$, puis on construit un automate B qui satisfait $\mathrm{L}(B) = \overline{\mathrm{L}(A)}$. Nous utilisons la construction classique qui consiste à rendre les états acceptants de A non acceptants, et vice versa.

4.2.4 Quantification existentielle

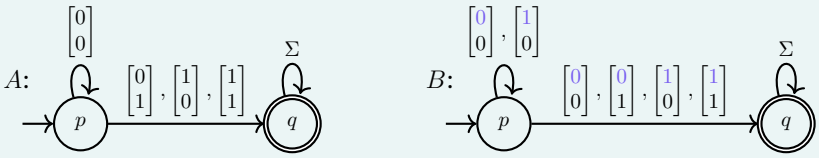
Considérons une formule de la forme $\exists x\ \varphi$. Nous construisons d'abord un automate A pour $[\![\varphi]\!]$. Ensuite, nous construisons un automate B obtenu en remplaçant chaque transition $p \to^\sigma q$ de A, par les transitions

$$p \to^{\sigma'} q \text{ où } \sigma'(y) = \sigma(y) \text{ pour } y \in V \setminus \{x\}.$$

Autrement dit, nous rendons la variable x libre en permettant n'importe quel bit dans sa composante, et en laissant les autres composantes inchangées.

Exemple.

La quantification existentielle de la première variable dans l'automate A ci-dessous mène à l'automate B:

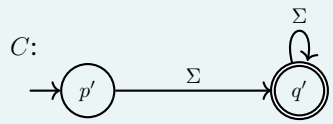


Notons que B n'est pas forcément valide. En effet, il est possible que dans un même état, il existe plus d'une transition sortante étiquetée par la même lettre. Cela se produit dans l'exemple précédent: par ex. dans l'état p, on peut lire la lettre $[0, 0]$ vers p ou vers q.

En théorie des automates, nous appelons cela un *automate non déterministe*. Il est bien connu que tout automate non déterministe B peut être converti dans un automate (déterministe) C qui accepte le même langage (par ex. en utilisant la construction classique par sous-ensembles décrite à l'exercice 4.13)). Dans le pire cas, cette déterminisation augmente le nombre d'états exponentiellement.

Exemple.

L'automate non déterministe B de l'exemple précédent est équivalent à cet automate déterministe:

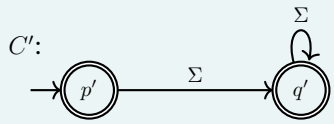


Il y a un autre enjeu: Il est possible que certains codages soient manquants. Par exemple, l'automate C ci-dessus accepte $[0, 0]$, mais pas ε bien que ces deux mots représentent les mêmes valeurs.

On peut réparer l'automate comme suit: si un état p peut atteindre un état acceptant en utilisant seulement $[0, \ldots, 0]^*$, alors on rend p acceptant.

Exemple.

L'état p' de l'automate C n'est pas acceptant, mais en lisant $[0, 0]$ dans p', on atteint l'état acceptant q'. On rend donc p' acceptant:



4.2.5 Approche complète

Soit φ une formule close. Afin de déterminer si $\varphi \equiv \texttt{vrai}$, on procède ainsi:

— On construit récursivement un automate A_φ qui représente l'ensemble $[\![\varphi]\!]$ en appliquant les constructions précédentes;

— On obtient $\mathrm{L}(A_\varphi) = \emptyset$ (si $\varphi \equiv \texttt{faux}$) ou $\mathrm{L}(A_\varphi) = \Sigma^*$ (si $\varphi \equiv \texttt{vrai}$).

Voyons un exemple complet. Considérons la formule

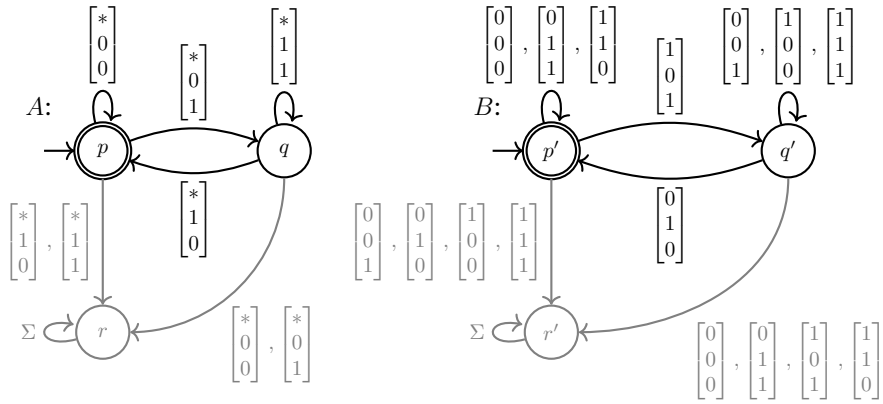
$$\varphi \coloneqq \forall x \exists y : (\exists z : y = 2z) \wedge (y \geq x).$$

Celle-ci affirme que pour tout entier naturel x, il existe un entier naturel pair y plus grand ou égal à x. Évidemment cette affirmation est vraie, mais vérifions-le algorithmiquement.

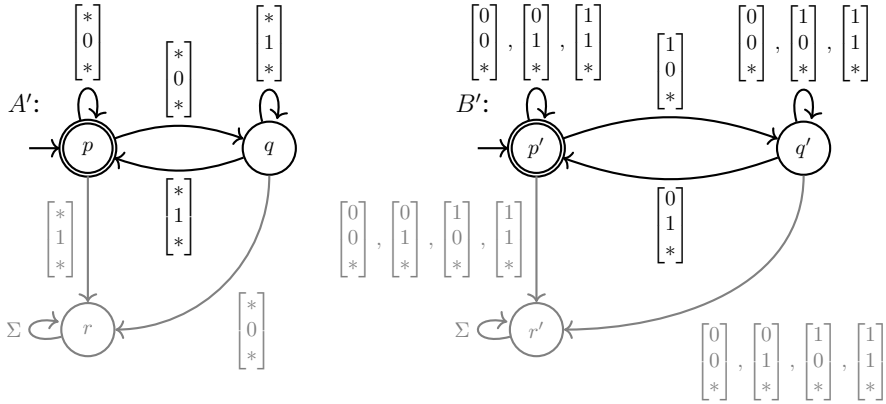
Avant de procéder, remarquons que φ ne respecte pas la syntaxe permise. Néanmoins, φ peut être réécrite de façon équivalente:

$$\neg(\exists x\ (\neg(\exists y\ ((\exists z\ (y = z + z)) \wedge (\exists z\ (y = x + z)))))).$$

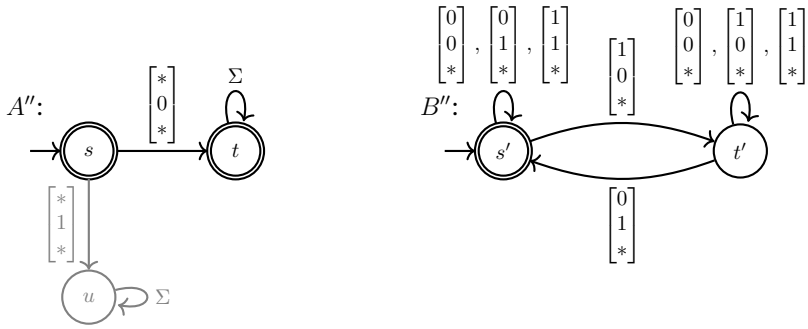
Convertissons cette formule sous forme d'automate. Les formules atomiques $y = z + z$ et $y = x + z$ sont traduites dans les automates A et B ci-dessous, où $[*, a, b]$ désigne les lettres $[0, a, b]$ et $[1, a, b]$.



Nous devons maintenant quantifier z existentiellement dans les deux automates. Nous obtenons ces automates non déterministes A' et B':



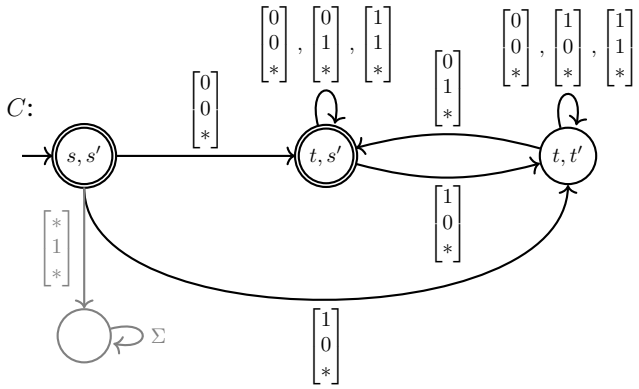
Après déterminisation, nous obtenons ces automates A'' et B'':



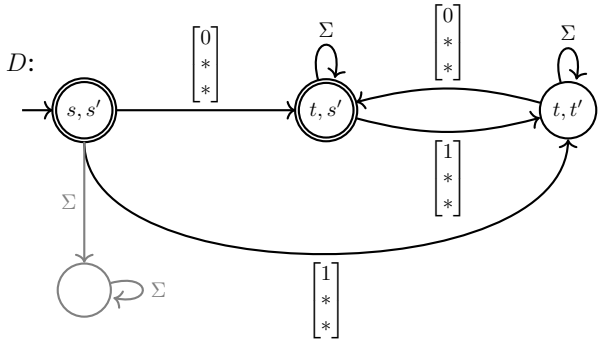
Informellement, la formule de départ s'écrit maintenant:

$$\neg(\exists x\ (\neg(\exists y\ (A'' \wedge B'')))).$$

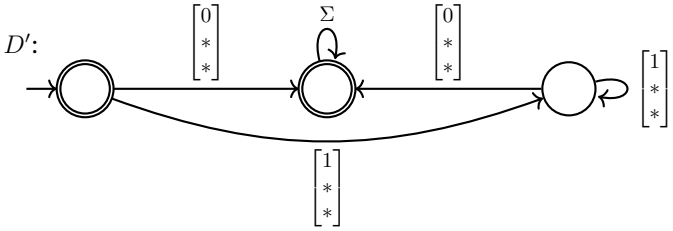
Nous devons intersecter A'' et B''. Cela mène à l'automate C ci-dessous. En mots, l'automate C représente « y est pair et $x \leq y$ ».



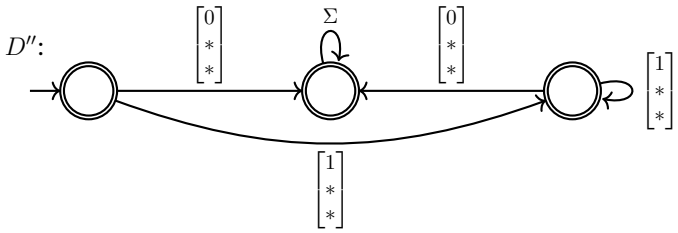
La quantification de y mène à cet automate non déterministe D:



Après déterminisation, on obtient cet automate D':



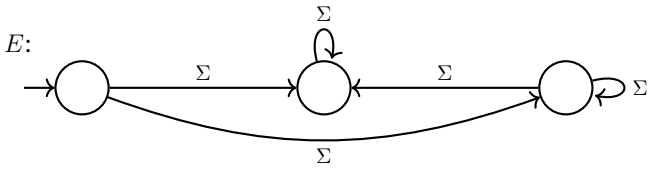
L'automate D' ne respecte pas les contraintes de codages. En effet, il accepte $[10, 00, 00]$ mais rejette $[1, 0, 0]$ qui est pourtant équivalent. Comme l'état tout à droite mène à l'état acceptant du centre en lisant $[0, 0, 0]$, on le rend acceptant:



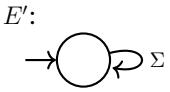
Informellement, la formule de départ s'écrit maintenant:

$$\neg(\exists x\ (\neg D'')).$$

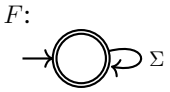
On doit complémenter D'', puis quantifier x existentiellement:



L'automate E est non déterministe. Après déterminisation, on obtient:



Il nous reste à complémenter l'automate une dernière fois:



Par construction, le langage $\mathrm{L}(F)$ représente $[\![\varphi]\!]$. Comme $\mathrm{L}(F) = \Sigma^*$, on conclut que $\varphi \equiv \texttt{vrai}$, comme attendu.

Théorème 3. *Le problème de validité de l'arithmétique de Presburger est décidable.*

Remarque.

Les résultats de décidabilité et d'indécidabilité demeurent les mêmes sur $\mathbb{Z}$ et $\mathbb{Q}$. Par contre, le portait est différent sur les nombres réels. En effet, $\mathsf{FO}(\mathbb{R}, \leq, +, \cdot)$ est décidable (voir l'exercice 4.17)). L'arithmétique de Skolem $\mathsf{FO}(\mathbb{N}, =, \cdot)$ est décidable (voir l'exercice 4.16)). Le fragment existentiel de l'arithmétique de Presburger est NP-complet. Pour un survol sur l'arithmétique de Peano, consultez [Haa18].

4.3 Exercices

4.1) ♥☆■ Considérons $\mathsf{FO}(\mathbb{N}, =, +, \psi)$ où $\psi(x, y)$ est un prédicat qui teste si x est une puissance de deux et y est divisible par x. Par exemple, $\psi(4, 12) \equiv$ `vrai` et $\psi(4, 5) \equiv$ `faux`. Montrez que cette extension de l'arithmétique de Presburger est décidable.

4.2) ♥☆🖉 Comment peut-on utiliser le prédicat ψ de l'exercice précédent afin d'exprimer ces deux prédicats: « x est une puissance de deux » et « x est la plus grande puissance de deux qui divise y ». Donnez également des automates pour ces deux prédicats.

4.3) ♥☆🖉 Comment peut-on introduire les constantes 0 et 1 à l'aide de la quantification existentielle, l'addition et la comparaison?

4.4) ♡☆🖉 Comment peut-on introduire $z = \min(x, y)$ et $z = \max(x, y)$ dans l'arithmétique de Presburger?

4.5) ♥★≤ Montrez que $\mathsf{FO}(\mathbb{N}, =, +, \div)$ est indécidable.

4.6) ♡★≤ Montrez que $\mathsf{FO}(\mathbb{N}, =, +, \cdot, 2^x)$ est indécidable.

4.7) ♥☆🖉 Donnez un automate qui représente $[\![y = x + 1]\!]$.

4.8) ♡★🖉 Donnez un automate qui représente $[\![x \bmod 2 = y \bmod 2 \wedge x \geq y]\!]$.

4.9) ♡☆■ Montrez que le problème, qui consiste à déterminer si deux formules de l'arithmétique de Presburger sont équivalentes, est décidable.

4.10) ♡☆🖉 Considérons la transition $\delta(p, a) = (q, b, \mathsf{D})$ d'une machine de Turing. Décrivez une formule $\mathrm{trans}(x, y)$ de l'arithmétique de Peano étendue qui vérifie si x et y codent des configurations C et C' telles que $C \to C'$.

4.11) ♡★🖉 Le but de cet exercice est de montrer que l'arithmétique de Peano est indécidable.

a) Soit b un nombre premier. Donnez une formule de $\mathsf{FO}(\mathbb{N}, =, +, \cdot)$ qui détermine si x est une puissance de b.

b) À la section 4.1, nous avons prouvé l'indécidabilité de l'arithmétique de Peano étendue, à l'aide des macros $x[i]$ et $x[i : j]$ définies avec l'exponentiation. Cherchons à nous débarrasser de l'exponentiation. Nous allons représenter les indices i et j indirectement par les puissances $d = b^i$ et $e = b^j$. Implémentez les macros $x[d]$ et $x[d : e]$ dans $\mathsf{FO}(\mathbb{N}, =, +, \cdot)$.

c) Montrez que l'arithmétique de Peano est indécidable à l'aide des nouvelles macros.

4.12) ♡☆🖉 Donnez une formule de $\mathsf{FO}(\mathbb{D}, \leq, +, \cdot)$ qui est fausse lorsque $\mathbb{D} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}\}$ et vraie lorsque $\mathbb{D} = \mathbb{R}$.

4.13) ♡★🎓 Un *automate non déterministe* est un quintuplet $(Q, \Sigma, \delta, q_0, F)$ où ↑↓

— Q est un ensemble fini dont les éléments se nomment *états*;

— Σ est un alphabet;

— $\delta \subseteq Q \times \Sigma \times Q$ est une relation de *transition*;

— $q_0 \in Q$ est l'*état initial*; et

— $F \subseteq Q$ est l'ensemble des états acceptants.

Autrement dit, la seule différence avec un automate (déterministe) est le fait que zéro, une ou plusieurs transitions étiquetées par la même lettre puissent quitter un état. Un automate non déterministe A accepte un mot w si et seulement si A posssède au moins un chemin qui accepte w.

Il est bien connu que l'automate (déterministe) $A' = (Q', \Sigma, \delta', q_0', F')$ suivant accepte le même langage que A:

— $Q' \coloneqq \{P : P \subseteq Q\}$;

— $\delta'(P, a) \coloneqq \{q \in Q : \exists p \in P\ (p, a, q) \in \delta\}$;

— $q_0' \coloneqq \{q_0\}$;

— $F' \coloneqq \{P : P \cap F \neq \emptyset\}$.

Reprenez un automate non déterministe de la section 4.2.5 et déterminisez-le à l'aide de la construction ci-dessus.

4.14) ♡★🎓 La logique monadique du second ordre, dénotée MSO, étend la logique du premier ordre en permettant de quantifier sur des ensembles. Donnez des formules de $\mathsf{MSO}(\mathbb{N}, =, +)$ qui déterminent si

— X décrit l'ensemble des nombres pairs;

— x est un nombre premier;

— z est le plus petit commun mulitple de x et y;

— z est le plus grand diviseur commun de x et y.

4.15) ♡★🎓 Montrez que $\mathsf{MSO}(\mathbb{N}, =, +)$ est indécidable.

4.16) ♡★🎓 Le but de cet exercice est de montrer que l'arithmétique de Skolem, c.-à-d. $\mathsf{FO}(\mathbb{N}, =, \cdot)$, est décidable. Soit $\mathbb{P}$ l'ensemble des nombres premiers. Rappelons que chaque entier naturel possède une décomposition unique en facteurs premiers. Ainsi, chaque entier naturel peut être vu comme un multiensemble fini de $\mathbb{P}$. Par exemple, $20 \equiv \{2, 2, 5\}$ et $30 \equiv \{2, 3, 5\}$. ↑↓

a) Expliquez comment coder un multiensemble fini de $\mathcal{P}$ par un mot.

b) Expliquez comment manipuler chaque « brique de base » de l'arithmétique de Skolem à l'aide d'automates.

c) Déduisez-en la décidabilité de l'arithmétique de Skolem.

4.17) ♡★🎓 Le but de cet exercice est de montrer (en partie) que l'arithmétique non linéaire, c.-à-d. $\mathsf{FO}(\mathbb{R}, \leq, +, \cdot)$, est décidable. ↑↓

a) Une formule du premier ordre φ est en *forme prénexe* si tous ses quantificateurs apparaissent à gauche. Par exemple, « $\forall x\ \exists y\ (x + y = 0)$ » est en forme prénexe, mais pas « $\forall x\ ((x > 0) \to (\exists y\ (y > x)))$ ». Montrez qu'il est toujours possible de mettre une formule du premier ordre en forme prénexe.

b) Soit V un ensemble fini de variables. Nous disons que $X \subseteq \mathbb{R}^V$ est un *ensemble semi-algébrique* s'il est égal à une union finie d'ensembles de la forme

$$\left\{ \boldsymbol{x} \in \mathbb{R}^V : \bigwedge_{i=1}^{m} (p_i(\boldsymbol{x}) = 0) \wedge \bigwedge_{j=1}^{n} (q_j(\boldsymbol{x}) > 0) \right\},$$

où $p_1, \ldots, p_m, q_1, \ldots, q_n \in \mathbb{R}[V]$ sont des polynômes.

Montrez que l'intersection, l'union et la complémentation d'ensembles semi-algébriques demeure semi-algébrique.

c) Étant donnés $v \in V$ et $\boldsymbol{x} \in \mathbb{R}^V$, nous écrivons $\pi_v(\boldsymbol{x})$ afin de dénoter la projection de $\boldsymbol{x}$ sur $\mathbb{R}^{V\setminus\{v\}}$, c.-à-d. le vecteur identique à $\boldsymbol{x}$ mais sans la composante v. Soit $Y \subseteq \mathbb{R}^{V\setminus\{v\}}$ un ensemble semi-algébrique. Montrez que $\{\boldsymbol{x} \in \mathbb{R}^V : \pi_v(\boldsymbol{x}) \in Y\}$ est un ensemble semi-algébrique (ce qui est quasiment trivial).

d) Pour tout $X \subseteq \mathbb{R}^V$, posons $\pi_v(X) \coloneqq \{\pi_v(\boldsymbol{x}) : \boldsymbol{x} \in X\}$. Le théorème de Tarski–Seidenberg affirme que la projection d'un ensemble semi-algébrique est un ensemble effectivement semi-algébrique. Plus formellement, pour tout $v \in V$ et pour tout ensemble semi-algébrique $X \subseteq \mathbb{R}^V$, l'ensemble $\pi_v(X)$ est semi-algébrique, et il est possible d'en calculer une représentation.

Nous n'allons pas démontrer le théorème (cela nécessiterait bien plus qu'un exercice). Voyons tout de même un exemple. Considérons l'ensemble semi-algébrique $X = \{(a, b, c, x) \in \mathbb{R}^4 : a^2x + bx + c = 0\}$. Convainquez-vous que $\pi_x(X)$ est semi-algébrique.

e) Montrez que pour toute formule $\varphi \in \mathsf{FO}(\mathbb{R}, \leq, +, \cdot)$, l'ensemble $[\![\varphi]\!]$ est semi-algébrique et on peut en obtenir une représentation. Utilisez le théorème de Tarski–Seidenberg.

f) Montrez que $\mathsf{FO}(\mathbb{R}, \leq, +, \cdot)$ est décidable.

5

Calcul parallèle et ses limitations

Nous avons appris que certains problèmes sont indécidables, alors que d'autres sont décidables. Parmi ces derniers, certains se résolvent efficacement, alors que pour d'autres cela semble impossible. Pour ceux qui sont solubles efficacement, est-ce possible d'être encore plus efficace? Par exemple, pouvons-nous tirer profit des nombreux processeurs (ou cœurs) des ordinateurs?

À priori, les machines de Turing se prêtent mal à ce questionnement puisqu'elles sont intrinsèquement séquentielles. Bien qu'on puisse étendre le modèle avec plusieurs unités de calcul qui partagent des registres, il y a de nombreux détails techniques à considérer. Il est donc coutume d'étudier le calcul parallèle à l'aide d'un autre modèle: les circuits. Ceux-ci sont (1) plus faciles à analyser d'un point de vue mathématique; (2) assez réalistes comme nos processeurs sont constituée de circuits; et (3) équivalents aux machines de Turing parallèles.

5.1 Circuits

Un *circuit (logique)* est un graphe dirigé acyclique qui satisfait ces propriétés:

- Les sommets d'entrance zéro sont les sommets dits d'*entrée*;
- Les sommets de sortance zéro sont les sommets dits de *sortie*;
- Les sommets internes d'entrance un sont étiquetés par la porte logique $\neg$;
- Les sommets internes d'entrance deux ou plus sont étiquetés par les portes logiques $\{\wedge, \vee\}$;
- Les n sommets d'entrée sont étiquetés respectivement par des variables $x_1, \ldots, x_n$.

Un circuit avec n entrées et m sorties calcule naturellement une fonction $f\colon \{0,1\}^n \to \{0,1\}^m$. Par exemple, le circuit de la figure 5.1 calcule $f(x_1, x_2) \coloneqq x_1 \oplus x_2$, où $\oplus$ dénote le OU exclusif.

La direction des arêtes est rarement indiquée dans les diagrammes; on suppose que la direction va du haut vers le bas.

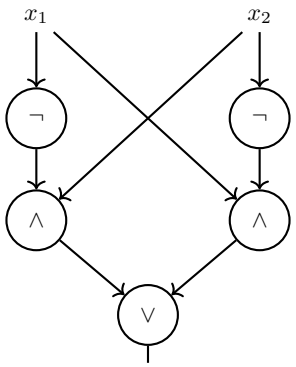


FIGURE 5.1 – Exemple de circuit qui calcule la parité de deux bits.

La *taille* d'un circuit est son nombre de portes, par ex. 5 pour le circuit de la figure 5.1. La profondeur d'un circuit est la quantité maximale de portes sur un chemin allant d'une entrée vers une sortie, par ex. 3 pour le circuit de la figure 5.1. Plus formellement, la *profondeur* d'un sommet est définie récursivement:

$$\begin{aligned}
\text{prof}(x_i) &\coloneqq 0,\\
\text{prof}(\vee_{1\le i\le n}\, v_i) &\coloneqq \max(\text{prof}(v_1),\dots,\text{prof}(v_n))+1,\\
\text{prof}(\wedge_{1\le i\le n}\, v_i) &\coloneqq \max(\text{prof}(v_1),\dots,\text{prof}(v_n))+1,\\
\text{prof}(\neg v) &\coloneqq \text{prof}(v)+1.
\end{aligned}$$

La *profondeur* d'un circuit C est définie par

$$\text{prof}(C) \coloneqq \max\{\text{prof}(u) : u \text{ est un sommet de sortie de } C\}.$$

Nous interdirons parfois les portes d'entrance arbitraire, c.-à-d. les sommets avec plus de deux prédecesseurs. Il est possible de remplacer une porte d'entrance arbitraire par une cascade de portes équivalentes; par exemple, $\bigvee_{1\le i\le 8} x_i$ devient

$$\Big(\big(x_1 \vee x_2\big) \vee \big(x_3 \vee x_4\big)\Big) \vee \Big(\big(x_5 \vee x_6\big) \vee \big(x_7 \vee x_8\big)\Big).$$

Lorsque l'entrée est de taille impaire, nous découpons environ en deux. Plus formellement, étant donné une porte $\circ \in \{\vee, \wedge\}$, nous utilisons cette réécriture récursive:

$$\text{casc}(\bigcirc_{1\le i\le n}\, x_i) \coloneqq \begin{cases} x_1 & \text{si } n = 1,\\ x_1 \circ x_2 & \text{si } n = 2,\\ \big(\text{casc}(\bigcirc_{1\le i\le n\div 2}\, x_i)\big) \circ \big(\text{casc}(\bigcirc_{n\div 2<i\le n}\, x_i)\big) & \text{si } n > 2. \end{cases}$$

Cette transformation crée un circuit de taille linéaire et de profondeur logarithmique:

Proposition 4. *Soit* $u \coloneqq \mathrm{casc}(\bigcirc_{1\leq i\leq n} x_i)$ *où* $\circ \in \{\vee, \wedge\}$. *Nous avons* $\mathrm{taille}(u) = n-1$ *et* $\mathrm{prof}(u) \leq \lceil \log n \rceil$.

5.2 Classes de complexité

Contrairement aux machines de Turing ou aux programmes, un circuit ne peut gérer qu'une seule taille d'entrée. Ainsi, pour accomplir une tâche calculatoire, il faut utiliser une famille de circuits, c'est-à-dire un circuit pour chaque taille d'entrée.

Soit $i \in \mathbb{N}$. Nous disons qu'une fonction $f\colon \{0,1\}^* \to \{0,1\}^*$ appartient à la classe de complexité FAC^i s'il existe une famille de circuits $\{C_0, C_1, \ldots\}$ et une constante $c \in \mathbb{N}$ telles que

— $\mathrm{taille}(C_n) \in \mathcal{O}(n^c)$,

— $\mathrm{prof}(C_n) \in \mathcal{O}(\log^i n)$ où $\log^i n$ dénote $(\log n)^i$,

— $C_n(x) = f(x)$ pour toute entrée $x \in \{0,1\}^n$.

La classe FNC^i est définie de la même façon, mais en interdisant les portes d'entrance arbitraire. Les classes AC^i et NC^i sont définies respectivent comme FAC^i et FNC^i mais en ne permettant qu'un seul bit en sortie; autrement dit, la fonction calculée doit être de la forme $f\colon \{0,1\}^* \to \{0,1\}$. Ainsi, AC^i et NC^i sont des classes de problèmes de décision.

Posons

$$\mathsf{AC} \coloneqq \bigcup_{i\in\mathbb{N}} \mathsf{AC}^i \text{ et } \mathsf{NC} \coloneqq \bigcup_{i\in\mathbb{N}} \mathsf{NC}^i.$$

En mots, un problème de décision appartient à AC s'il existe une famille de circuits de taille polynomiale et de profondeur polylogarithmique qui calcule f.

Comme la taille d'un circuit correspond informellement au nombre d'unités de calcul, et que la profondeur d'un circuit correspond au temps d'exécution, la classe NC est considérée comme l'ensemble des problèmes de décision qui peuvent être résolus efficacement en parallèle.

Par la proposition 4, si une porte possède n^c entrées, alors elle peut être remplacée par un sous-circuit équivalent de taille $n^c - 1$ et de profondeur au plus $\lceil \log(n^c) \rceil = \lceil c \log n \rceil \in \mathcal{O}(\log n)$. Ainsi, si un circuit de taille polynomiale et de profondeur $\mathcal{O}(\log^i n)$ possède des portes d'entrance arbitraire, alors ce circuit peut être converti en un circuit équivalent de taille polynomiale et de profondeur $\mathcal{O}(\log(n) \cdot \log^i(n)) = \mathcal{O}(\log^{i+1} n)$.

Par conséquent, $\mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$ pour tout $i \in \mathbb{N}$, ce qui implique $\mathsf{AC} = \mathsf{NC}$ car

$$\mathsf{NC}^0 \subseteq \mathsf{AC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \mathsf{AC}^2 \subseteq \cdots.$$

Ainsi, pour montrer l'appartenance à NC, il suffit de raisonner sur AC sans se soucier des portes d'entrance arbitraire.

5.3 Addition

Considérons le problème qui consiste à additionner deux entiers naturels:

ADD

ENTRÉE: deux entiers $a, b \geq 0$ décrits en binaire avec n bits,
SORTIE: $a + b$ décrit en binaire avec $n + 1$ bits.

Nous allons montrer que ce problème appartient à FAC^0 et ainsi qu'il est parallélisable.

5.3.1 Somme de deux bits

Supposons que l'on souhaite additionner deux bits a et b. La somme peut être représentée sur deux bits c_1c_0 puisque la somme varie entre 0 et 2 inclusivement. La table de vérité est comme suit:

a	b	c_1	c_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

En inspectant la table, on remarque que $c_1 = a \wedge b$ et $c_0 = a \oplus b$. Rappelons que « $a \oplus b$ » n'est pas directement permis, mais peut être émulé par $(a \wedge \neg b) \vee (\neg a \wedge b)$.

5.3.2 Somme de trois bits

Supposons que l'on souhaite additionner les bits a, b et r. La somme peut être représentée sur deux bits c_1c_0 puisque la somme varie entre 0 et 3 inclusivement. La table de vérité est comme suit:

a	b	r	c_1	c_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
1	0	0	0	1
0	1	1	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

En inspectant la table, on remarque que $c_1 = 1$ ssi $\{a, b, r\}$ contient au moins deux occurrences de 1, et que c_2 est la parité du nombre d'occurrences de 1. Ainsi, $c_1 = (a \wedge b) \vee ((a \vee b) \wedge r)$ et $c_0 = (a \oplus b) \oplus r$.

Rappelons que « $(a \oplus b) \oplus r$ » n'est pas directement permis, mais peut être émulé par

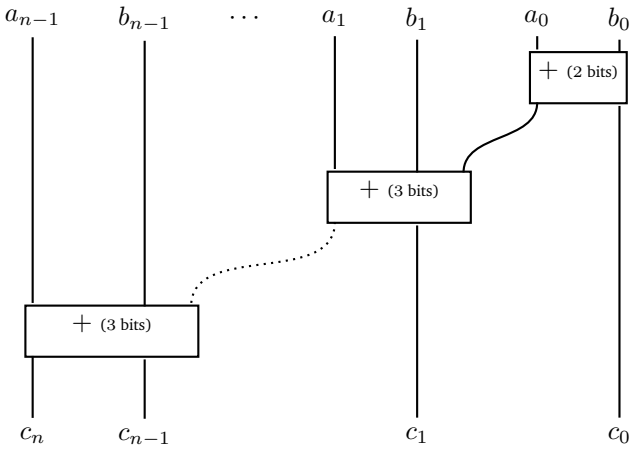
$$(y \wedge \neg r) \vee (\neg y \wedge r) \text{ où } y = (a \wedge \neg b) \vee (\neg a \wedge b).$$

5.3.3 Somme de deux nombres: approche séquentielle

Nous pouvons additionner deux entiers binaires $a_{n-1} \cdots a_1 a_0$ et $b_{n-1} \cdots b_1 b_0$ en additionnant séquentiellement leurs bits, du poids faible vers le poids fort. Plus formellement, en réutilisant les identités établies, nous avons

$$\begin{aligned} c_0 &\coloneqq a_0 \oplus b_0, & c_i &\coloneqq (a_i \oplus b_i) \oplus r_{i-1}, & c_n &\coloneqq r_{n-1}, \\ r_0 &\coloneqq a_0 \wedge b_0, & r_i &\coloneqq (a_i \wedge b_i) \vee ((a_i \vee b_i) \wedge r_{i-1}). \end{aligned}$$

Cela mène à ce circuit C_n:



Analysons la taille de C_n. La portion qui calcule $a_0 + b_0$ possède six portes. La portion qui calcule $a_i + b_i$, où $i > 0$, possède quatorze portes. Ainsi, taille$(C_n) = 6 + 14(n-1) = 14n - 8 \in \Theta(n)$. Comme chaque retenue r_i dépend de la retenue précédente r_{i-1}, la profondeur de C_n est linéaire. En effet, la proposition suivante montre que prof$(C_n) \in \Theta(n)$:

Proposition 5. *Nous avons* prof$(r_i) = 2i + 1$ *et* prof$(c_i) = \max(2i + 2, 1)$ *pour tout* $0 \leq i < n$. ↑↓

Rappelons que nous cherchons à obtenir une profondeur polylogarithmique. Nous devons donc trouver une meilleure approche.

5.3.4 Somme de deux nombres: approche parallèle déraisonnable

Il est possible de calculer la somme de deux nombres par force brute en figeant la table d'addition dans le circuit. Cela mène à un circuit de profondeur constante, mais de taille au moins exponentielle comme il y a $2^{n+n} = 2^{2n} = 4^n$ lignes

dans la table. Techniquement, cela mène à un temps parallèle constant, mais, comme la taille correspond informellement au nombre d'unités de calcul, cela est déraisonnable. Formellement, un tel circuit ne satisfait pas les constraintes de FAC^0 en raison de sa taille non-polynomiale. Voyons donc une façon plus ingénieuse d'obtenir une profondeur constante.

5.3.5 Somme de deux nombres: approche parallèle raisonnable

Rappelons les identités de la section 5.3.3:

$$\begin{aligned} c_0 &= a_0 \oplus b_0, & c_i &= (a_i \oplus b_i) \oplus r_{i-1}, & c_n &\coloneqq r_{n-1}, \\ r_0 &= a_0 \wedge b_0, & r_i &= (a_i \wedge b_i) \vee ((a_i \vee b_i) \wedge r_{i-1}). \end{aligned}$$

Nous pouvons exprimer chaque retenue de telle sorte à ce que sa valeur ne dépende plus des retenues précédentes:

$$\begin{aligned} c'_0 &\coloneqq a_0 \oplus b_0, & c'_i &\coloneqq (a_i \oplus b_i) \oplus r'_{i-1}, & c'_n &\coloneqq r'_{n-1}, \\ r'_0 &\coloneqq a_i \wedge b_i, & r'_i &\coloneqq \bigvee_{i \geq j \geq 0} \left((a_j \wedge b_j) \wedge \bigwedge_{i \geq k > j} (a_k \vee b_k) \right). \end{aligned}$$

Voyons d'abord informellement pourquoi nous avons choisi ces nouvelles identités pour les retenues. La somme de a_i, b_i et r'_{i-1} mène à une retenue ssi nous avons $a_i = b_i = 1$, ou $(a_i \vee b_i) = 1$ et $r'_{i-1} = 1$. Similairement, $r'_{i-1} = 1$ ssi $a_{i-1} = b_{i-1} = 1$, ou $(a_{i-1} \vee b_{i-1}) = 1$ et $r'_{i-2} = 1$. Comme on ne peut pas dérouler ce raisonnement à l'infini, on doit forcément s'arrêter à un certain indice j où $a_j = b_j = 1$. Graphiquement, $r'_i = 1$ ssi nous avons quelque chose du genre:

$$\begin{array}{cccccccccc} & i & & \longleftarrow & k & \longrightarrow & & j & & \\ + & 0 & 1 & 0 & 0 & 1 & \cdots & 1 & \cdots \\ & 1 & 0 & 1 & 1 & 0 & \cdots & 1 & \cdots \\ \hline \end{array}$$

Formellement, la nouvelle identité ne change en effet pas les valeurs calculées:

Proposition 6. *Nous avons $r_i = r'_i$ pour tout $0 \leq i < n$, et $c_i = c'_i$ pour tout $0 \leq i \leq n$.* ↑↓

Analysons la taille de C'_n, sans se soucier des « $\oplus$ » qui ne changent la taille et la profondeur que d'un facteur constant. Il y a deux portes pour le calcul de c'_0 et r'_0. Le nombre de portes pour le calcul de r'_i est borné par

$$\sum_{i \geq j \geq 0} \left(3 + \sum_{i \geq k > j} 2 \right) \in \mathcal{O}(n^2).$$

Le nombre de portes pour le calcul de c'_i est égal à celui pour r'_i plus deux, donc aussi de $\mathcal{O}(n^2)$. Par conséquent, taille$(C'_n) \in \mathcal{O}(2+n \cdot n^2) = \mathcal{O}(n^3)$. Remarquons

que cette analyse se raffine. En effet, les termes de la forme $\bigwedge_{i \geq k > j}(a_k \vee b_k)$ peuvent être partagés par $r'_1, \ldots, r'_n$ sans être entièrement recalculés par chaque r'_i. Une telle analyse mène à taille$(C'_n) \in \mathcal{O}(n^2)$, bien que la complexité exacte n'importe pas dans notre contexte.

Analysons maintenant la profondeur du circuit. Il découle directement des définitions que prof$(c'_0) =$ prof$(r'_0) = 1$, prof$(r'_i) = 4$ et prof$(c'_i) = 5$.

Comme la taille est polynomiale, que la profondeur est constante, et que $\mathcal{O}(1) = \mathcal{O}(\log^0 n)$, nous venons de montrer ce résultat:

Théorème 4. ADD $\in$ FAC^0.

En mots, ce résultat signifie qu'il est possible d'additionner deux nombres en temps parallèle constant.

5.4 Accessibilité

Reconsidérons maintenant le problème d'accessibilité introduit au chapitre 1:

PATH

ENTRÉE:	un graphe dirigé $G = (V, E)$ décrit par une matrice d'adjacence, et deux sommets $s, t \in V$,
QUESTION:	existe-t-il un chemin de s vers t dans G?

Dans cette section, nous allons montrer que ce problème appartient à AC$^1 \subseteq$ NC^2 et ainsi qu'il est parallélisable.

Étant donné un graphe $G = (V, E)$ et deux sommets $u, v \in V$, nous définissons $\mathbf{A}^\ell[u, v]$ comme suit:

$$\mathbf{A}^0[u, v] \coloneqq 1 \text{ si } u = v, 0 \text{ sinon,}$$

$$\mathbf{A}^1[u, v] \coloneqq 1 \text{ si } u = v \text{ ou } (u, v) \in E, 0 \text{ sinon,}$$

$$\mathbf{A}^\ell[u, v] \coloneqq \bigvee_{x \in V} \left(\mathbf{A}^{\lfloor \ell/2 \rfloor}[u, x] \wedge \mathbf{A}^{\lceil \ell/2 \rceil}[x, v] \right).$$

Proposition 7. *Il existe un chemin de u vers v de longueur $\leq \ell$ ssi $\mathbf{A}^\ell[u, v] = 1$.*

Démonstration. Nous procédons par induction sur $\ell \in \mathbb{N}$. Les cas où $\ell \in \{0, 1\}$ découlement trivialement de la définition de $\mathbf{A}$. Soit $\ell > 1$.

$\Rightarrow$) Supposons qu'il existe un chemin de u vers v de longueur au plus ℓ. En particulier, il existe un chemin de longueur au plus $\lfloor \ell/2 \rfloor$ de u vers un certain sommet w, et un chemin de longueur au plus $\lceil \ell/2 \rceil$ de w vers v. Par hypothèse d'induction, nous avons $\mathbf{A}^{\lfloor \ell/2 \rfloor}[u, w] = 1$ et $\mathbf{A}^{\lceil \ell/2 \rceil}[w, v] = 1$. Rappelons que

$$\mathbf{A}^\ell[u, v] = \bigvee_{x \in V} \left(\mathbf{A}^{\lfloor \ell/2 \rfloor}[u, x] \wedge \mathbf{A}^{\lceil \ell/2 \rceil}[x, v] \right).$$

Comme $\mathbf{A}^{\lfloor \ell/2 \rfloor}[u, w] \wedge \mathbf{A}^{\lceil \ell/2 \rceil}[w, v]$ est vrai, nous avons forcément $\mathbf{A}^\ell[u, v] = 1$.

$\Leftarrow$) Supposons que $\mathbf{A}^\ell[u,v] = 1$. Par définition, nous avons

$$\mathbf{A}^\ell[u,v] = \bigvee_{x \in V} \left(\mathbf{A}^{\lfloor \ell/2 \rfloor}[u,x] \wedge \mathbf{A}^{\lceil \ell/2 \rceil}[x,v] \right) = 1.$$

Il existe donc $x \in V$ tel que $\mathbf{A}^{\lfloor \ell/2 \rfloor}[u,x] = 1$ et $\mathbf{A}^{\lceil \ell/2 \rceil}[x,v]$. Par hypothèse d'induction, il existe un chemin de longueur au plus $\lfloor \ell/2 \rfloor$ de u vers x, et un chemin de longueur au plus $\lceil \ell/2 \rceil$ de x vers v. Par conséquent, il existe un chemin de longueur au plus $\lfloor \ell/2 \rfloor + \lceil \ell/2 \rceil = \ell$ de u vers v. □

Remarquons que s'il existe un chemin entre deux sommets, alors il en existe forcément un qui est simple (c.-à-d. sans répétitions). En effet, il est toujours possible de retirer les cycles d'un chemin afin d'en obtenir un plus court. Cette observation implique ce lemme:

Lemme 1. *Soit $G = (V, E)$ un graphe dirigé. S'il existe un chemin de u vers v dans G, alors il existe un chemin de u vers v de longueur au plus $2^{\lceil \log |V| \rceil}$.*

Nous sommes maintenant prêt à montrer que le problème d'accessibilité se parallélise.

Théorème 5. PATH $\in$ AC^1.

Démonstration. Soit $G = (V, E)$ le graphe le graphe dirigé en entrée décrit par une matrice d'adjacence $\mathbf{B}$, et soient $s, t \in V$ les sommets en entrée. En combinant la proposition 7 et le lemme 1, nous remarquons qu'il existe un chemin de s vers t ssi $\mathbf{A}^k[s,t] = 1$ où $k \coloneqq 2^{\lceil |V| \rceil}$. Il suffit donc de donner un circuit qui calcule $\mathbf{A}^k[s,t]$.

Définissons un sommet $w_{i,u,v}$ du circuit qui calcule $\mathbf{A}^{2^i}[u,v]$. Posons $w_{0,u,v} \coloneqq 1$ si $u = v$, et $w_{0,u,v} \coloneqq \mathbf{B}[u,v]$ sinon. De plus, posons

$$w_{i,u,v} \coloneqq \bigvee_{x \in V} \left(w_{i-1,u,x} \wedge w_{i-1,x,v} \right).$$

La sortie du circuit global est $w_{\lceil \log |V| \rceil, s, t}$. Par définition, le circuit détermine correctement s'il existe un chemin de s vers t dans G.

Analysons la taille du circuit. Remarquons que la taille n de l'entrée appartient à $\Theta(|V|^2 + 2 \log |V|) = \Theta(|V|^2)$ puisque l'entrée est formée d'une matrice d'adjacence et de l'identifiant de deux sommets. Chaque sommet $w_{i,u,v}$ crée deux portes: une conjonction d'entrance deux, et une disjonction d'entrance arbitraire. Au total, le nombre de portes est donc d'au plus

$$2 \cdot \lceil \log |V| \rceil \cdot |V| \cdot |V| \in \mathcal{O}(|V|^2 \log |V|) \subseteq \mathcal{O}(n \log n).$$

Analysons maintenant la profondeur du circuit. Il n'y a que deux portes entre deux sommets de la forme $w_{i,*,*}$ et $w_{i-1,*,*}$. Comme i varie de 1 à $\lceil \log |V| \rceil$, la profondeur du circuit appartient à

$$\mathcal{O}(\log |V|) \subseteq \mathcal{O}(\log \sqrt{n}) = \mathcal{O}(\log(n^{1/2})) = \mathcal{O}(1/2 \log n) = \mathcal{O}(\log n).$$

Comme le circuit est de taille polynomiale et de profondeur logarithmique, nous obtenons PATH $\in$ AC^1. □

5.5 Uniformité

Rappelons qu'afin de montrer l'appartenance à AC ou NC, nous utilisons une *famille* de circuits. Dans nos exemples précédents, tous les circuits d'une même famille étaient similaires et relativement simples à produire. Or, techniquement, la définition permet de « tricher » en choisissant $\{C_0, C_1, C_2, \ldots\}$ de telle sorte à ce que les circuits soient très différents ou difficilement calculables.

En fait, pour des raisons pédagogiques, nous n'avons volontairement pas vu la dernière contrainte de la vraie définition de AC et NC: il faut que les circuits soient uniformes, ce qui informellement signifie qu'il est possible et simple d'obtenir le $n^{\text{ème}}$ circuit de la famille.

Par exemple, posons

$$L \coloneqq \{0^n : n \in \mathbb{N}, \text{ écrit en binaire, est le code d'une machine de Turing qui s'arrête sur son propre code}\}.$$

Nous savons que L est indécidable. Or, selon la définition de la section 5.2, le langage L appartient à AC^0 car la famille de circuits de la figure 5.2 décide L.

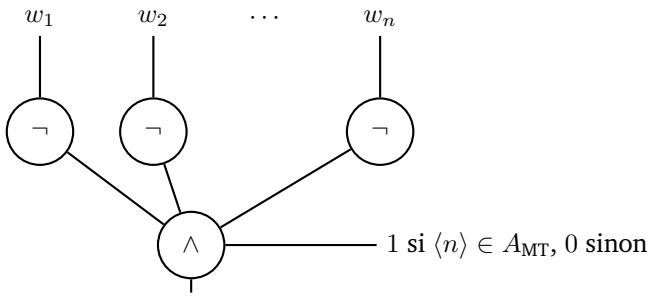


FIGURE 5.2 – Exemple d'une famille de circuits pour un problème indécidable.

La famille de circuits de la figure 5.2 existe mais elle n'est pas constructive: bien que le « bit magique » existe, il est impossible de le produire algorithmiquement.

Formellement, nous disons qu'une famille de circuits $\{C_0, C_1, \ldots\}$ est *uniforme* s'il existe une machine de Turing qui, étant donné n décrit en unaire, produit C_n. Cette définition n'est toujours pas satisfaisante. En effet, s'il était permis de passer un temps exponentiel à construire C_n, alors la haute complexité serait dissimulée. Nous restreignons donc les ressources permises pour qu'elles soient très limitées. Nous disons qu'une famille de circuits $\{C_0, C_1, \ldots\}$ est *log-uniforme* s'il existe une machine de Turing qui, étant donné n décrit en unaire, produit C_n en espace logarithmique.

Clarifions ce que nous entendons par « espace logarithmique ». En effet, cette contrainte est en apparence trop forte car l'entrée est de taille n, et qu'il faut la

lire en entier pour espérer produire quelque chose de pertinent. En fait, dans ce contexte, on suppose que la machine de Turing possède plusieurs rubans:

— Un ruban d'entrée en lecture seule;

— Un nombre constant de rubans en lecture/écriture qui ne peuvent excéder une taille de $\mathcal{O}(\log n)$.

Remarquons qu'une telle machine fonctionne forcément en temps polynomial. En effet, chaque case des rubans auxiliaires contient une lettre de l'alphabet de ruban Γ. Comme la taille des rubans auxiliaires est bornée par $c \log n$ pour une certaine constante c, la machine possède au plus $|\Gamma|^{c \log n}$ configurations différentes pour un même état. Cette valeur est polynomiale puisque

$$|\Gamma|^{c \log n} = (2^{\log |\Gamma|})^{c \log n} = 2^{\log |\Gamma| \cdot c \log n} = 2^{\log(n^{\log |\Gamma| \cdot c})} = n^{\log |\Gamma| \cdot c}.$$

Ainsi, une famille de circuits est log-uniforme ssi son $n^{\text{ème}}$ circuit peut être produit en temps polynomial et en espace logarithmique.

Les véritables définitions de FAC^i, FNC^i, AC^i, NC^i et $\mathsf{AC} = \mathsf{NC}$ demandent à ce que la famille de circuits soit log-uniforme. Tous les résultats établis jusqu'ici satisfont cette contrainte supplémentaire.

5.6 NC vs. P

Remarquons que les problèmes qui peuvent être résolus efficacement en parallèle peuvent être résolus efficacement en séquentiel:

Théorème 6. $\mathsf{NC} \subseteq \mathsf{P}$.

Démonstration. Soit $L \in \mathsf{NC}$. Montrons que $L \in \mathsf{P}$ en donnant une machine de Turing qui décide L en temps polynomial. Sur entrée $w \in \{0,1\}^n$, nous générons le $n^{\text{ème}}$ circuit C_n de la famille. Par définition de NC, il est possible d'y arriver en espace logarithmique, et donc en temps polynomial. On assigne ensuite w aux sommets d'entrées de C_n, puis on évalue C_n en le parcourant en ordre topologique. On accepte w ssi C_n évalue à 1. □

À ce jour, nous ne savons pas si $\mathsf{P} \subseteq \mathsf{NC}$ et ainsi si $\mathsf{NC} = \mathsf{P}$. Il existe donc une théorie de la complétude comme pour P vs. NP. Formellement, nous disons qu'un langage L est P-complet ssi

— $L \in \mathsf{P}$, et

— $L' \in \mathsf{P}$ implique $L' \leq_m^{\log} L$ (tout langage de P se réduit à L en espace logarithmique).

L'analogue de SAT en P-complétude est le problème d'évaluation de circuits:

CVP

ENTRÉE: un circuit C et une entrée x,
SORTIE: C évalue à 1 sur entrée x?

Théorème 7. CVP *est* P*-complet.*

Démonstration. Le problème CVP appartient à P puisqu'il est possible d'évaluer C sur entrée x en l'évaluant progressivement en ordre toplogique. Il nous reste donc à montrer que CVP est P-difficile.

Soit L un langage qui appartient à P. Nous devons montrer que L se réduit à CVP. Puisque $L \in \mathsf{P}$, il existe une machine de Turing $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ qui décide L en temps au plus n^c, où c est une constante. Nous allons montrer comment simuler $\mathcal{M}$ sur une entrée $w \in \{0,1\}^n$ à l'aide d'un circuit C. Nous décrivons C en plusieurs étapes.

Codage. Chaque élément d'une configuration de $\mathcal{M}$ appartient à $\Gamma \cup (Q \times \Gamma)$. Nous représentons un tel élément avec un codage *one-hot*, par ex. si $\Gamma = \{0, 1, \sqcup\}$ et $Q = \{q_0, q_1\}$, alors « $q_0 1$ » est représenté par 000010000:

0	1	$\sqcup$	$q_0\,0$	$q_0\,1$	$q_0\,\sqcup$	$q_1\,0$	$q_1\,1$	$q_1\,\sqcup$
0	0	0	0	1	0	0	0	0

Les configurations atteintes par $\mathcal{M}$ sur entrée w ne peuvent excéder une taille de n^c. Ainsi, toute configuration se représente avec n^c telles suites de bits.

Organisation. Le circuit C est organisé sous forme de « grille ». Le but de la $i^{\text{ème}}$ ligne du circuit est d'indiquer la configuration obtenue après i étapes de l'exécution de la machine $\mathcal{M}$. Par exemple, si $\Gamma = \{0, 1, \sqcup\}$, $Q = \{q_0, q_1\}$ et $w = 101$, alors le premier étage reçoit en entrée une suite de bits w' qui représente la configuration initiale $q_0 1\,0\,1\,\sqcup \cdots \sqcup$:

$$w' = 000010000\ 100000000\ 010000000\ 001000000\ \cdots\ 001000000.$$

Chaque autre ligne du circuit est calculée par rapport à la précédente. Voyons comment. Soit $u_{i,j,x}$ le sommet à la ligne i et colonne j du circuit qui indique si la $j^{\text{ème}}$ position de la $i^{\text{ème}}$ configuration contient l'élément $x \in \Gamma \cup (Q \times \Gamma)$.

Implémentation. Pour un élément de la forme $qb \in Q \times \Gamma$, nous définissons le sommet comme suit:

$$u_{i,j,qb} \coloneqq u_{i-1,j,b} \wedge \Bigg(\bigvee_{(p,a)\to(q,*,D)\in\delta} (j > 0 \wedge u_{i-1,j-1,pa}) \vee \bigvee_{(p,a)\to(q,*,G)\in\delta} (j < n^c \wedge u_{i-1,j+1,pa}) \Bigg).$$

En mots, l'expression ci-dessus affirme que la tête est en ce moment à la position j dans l'état q avec la lettre b ssi à l'étape précédente b était à la position j sans la tête, et

— la tête était à la position $j-1$ dans l'état p avec la lettre a, et la machine a exécuté une transition de la forme $(p, a) \to (q, *, D)$; ou

— la tête était à la position $j+1$ dans l'état p avec la lettre a, et la machine a exécuté une transition de la forme $(p,a) \to (q,*,G)$.

Pour un élément de la forme $b \in \Gamma$, nous définissons le sommet comme suit:

$$u_{i,j,b} \coloneqq (u_{i-1,j,b} \wedge \bigwedge_{q \in Q} \neg u_{i,j,qb}) \vee \bigvee_{(p,a)\to(*,b,*)\in\delta} u_{i-1,j,pa}.$$

En mots, l'expression ci-dessus affirme qu'en ce moment, la $j^{\text{ème}}$ case du ruban contient la lettre b, avec la tête ailleurs, ssi

— à l'étape précédente, b était déjà là, avec la tête ailleurs, et aucun déplacement n'a amené la tête; ou

— à l'étape précédente, la tête était là dans un état p avec une certaine lettre a, et la machine a exécuté une transition qui a écrasé a par b, puis s'est déplacée (à droite ou gauche).

Acceptation. Afin de vérifier si la machine $\mathcal{M}$ accepte w, le circuit vérifie si l'état d'acceptation a été atteint au dernier étage en inspectant chaque colonne. Le sommet v de sortie du circuit est donc défini comme suit:

$$v \coloneqq \bigvee_{1 \le j \le n^c} \bigvee_{a \in \Gamma} u_{n^c,j,q_{\text{acc}}a}.$$

Fin de la preuve. Clairement, le circuit C et son entrée w' sont constructibles en temps polynomial à partir de $\mathcal{M}$ et w. Avec un peu plus de travail, on pourrait argumenter que cela se fait en espace logarithmique. De plus, $w \in L$ ssi $\mathcal{M}$ accepte w ssi $C(w') = 1$. Ainsi, $L \le_m^{\log}$ CVP. □

Notons qu'il est aussi connu que l'optimisation linéaire est P-complète. Dans sa variante décisionnelle, ce problème consiste à déterminer s'il existe une solution $\boldsymbol{x} \in \mathbb{R}^k$ à un système d'inéquations $\mathbf{A}\boldsymbol{x} \le \boldsymbol{b}$, où $\mathbf{A} \in \mathbb{Z}^{m\times k}$ et $\boldsymbol{b} \in \mathbb{Z}^m$. Ainsi, bien qu'il soit possible de résoudre de tels systèmes en temps polynomial, nous ne savons toujours pas s'il est possible de tirer profit du parallélisme pour faire mieux!

Comme en théorie de la NP-complétude, si $A \le_m^{\log} B$ et A est P-difficile, alors B est P-difficile. De plus, si on arrive à montrer qu'un problème P-complet appartient à NC, alors NC $=$ P; et si on arrive à montrer qu'un problème P-complet n'appartient pas à NC, alors NC $\neq$ P.

5.7 🎓 Relation entre parallélisme et mémoire

Au début du chapitre, nous avons soulevé cette question: quels sont les problèms dans P que l'on peut résoudre encore plus efficacement. Une ressource que l'on peut chercher à réduire est la mémoire (aussi appelée l'espace). Il s'avère que les problèmes solubles avec une quantité logarithmique de mémoire sont étroitement liés aux problèmes parallélisables efficacement.

Rappelons qu'un problème se résout en *espace logarithmique* s'il existe une machine de Turing dont l'usage de mémoire auxiliaire appartient à $\mathcal{O}(\log n)$. Nous appelons L la classe des langages solubles en espace logarithmique.

Par exemple, le langage $\{a^k b^\ell : k \leq \ell\}$ appartient à L puisqu'on peut (1) vérifier que l'entrée est de la forme a^*b^*; (2) écrire k et ℓ en binaire sur deux rubans auxiliaires, ce qui nécessite $\mathcal{O}(\log k + \log \ell) = \mathcal{O}(\log n)$ bits; et (3) vérifier que $k \leq \ell$ en comparant les deux rubans bit à bit.

Comme nous l'avons vu à la section 5.5, une machine qui fonctionne en espace logarithmique possède forcément un temps d'exécution polynomial. Ainsi, $\mathsf{L} \subseteq \mathsf{P}$. Il est également possible de montrer que $\mathsf{NC}^1 \subseteq \mathsf{L}$.

La classe NL est en quelque sorte l'analogue de NP: il s'agit de l'ensemble des langages qu'on peut résoudre avec une machine de Turing non déterministe qui fonctionne en espace logarithmique. Bien évidemment: $\mathsf{L} \subseteq \mathsf{NL}$. Dans cette section, nous montrons que $\mathsf{NL} \subseteq \mathsf{AC}^1$ et étudions la NL-complétude.

5.7.1 NL se parallélise efficacement

Soit $L \in \mathsf{NL}$. Par définition, il existe une machine de Turing $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ qui décide L en espace $c \log n$. Soit k le nombre de rubans auxiliaires de $\mathcal{M}$. Étant donné $n \in \mathbb{N}$, nous allons définir un graphe dirigé $G_n = (V_n, E_n)$ dont les arêtes sont étiquetées par des lettres de Σ. Soit C_n l'ensemble des configurations possibles d'un ruban auxiliaire:

$$C_n \coloneqq \{w \in (\Gamma \cup \{\triangledown\})^* : |w| = c\log(n) \text{ et } |w|_\triangledown = 1\}.$$

Ici, la lettre $\triangledown$ représente la position de la tête. Posons

$$V_n \coloneqq Q \times [1..n+1] \times C_n^k \cup \{t_n\}.$$

Intuitivement, le sommet $(q, i, c_1, \ldots, c_k) \in V_n$ indique que la machine $\mathcal{M}$ est dans l'état q; avec sa tête principale à la position i; et son $j^{\text{ème}}$ ruban auxilaire dans la configuration c_j. Le sommet t_n est un sommet additionnel qui indique que $\mathcal{M}$ a terminé et accepté son entrée.

Pour chaque $u = (q, i, c_1, \ldots, c_k)$, le graphe contient une arête $u \to^a v$ ssi la machine $\mathcal{M}$ peut passer de la configuration u à la configuration v en une transition lorsque la $i^{\text{ème}}$ lettre du mot d'entrée est a. De plus, si $q = q_{\text{acc}}$, alors nous ajoutons l'arête $u \to^a t_n$ au graphe.

Notons que G_n est de taille polynomiale. En effet, $|E_n| \in \mathcal{O}(|V_n|^2)$ et

$$\begin{aligned}
|V_n| &\leq |Q| \cdot (n+1) \cdot (\Gamma+1)^{kc\log n} + 1 \\
&= |Q| \cdot (n+1) \cdot 2^{\log(\Gamma+1)\cdot kc \log n} + 1 \\
&= |Q| \cdot (n+1) \cdot 2^{\log(n^{\log(\Gamma+1)\cdot kc})} + 1 \\
&= |Q| \cdot (n+1) \cdot n^{\log(\Gamma+1)\cdot kc} + 1 \\
&\in \mathcal{O}(n^{1+\log(\Gamma+1)\cdot kc}).
\end{aligned}$$

Définissons s_n comme étant le sommet qui représente la configuration initiale (indépendamment du mot en entrée):

$$s_n \coloneqq (q_0, 1, \triangledown ⌴⌴ \cdots ⌴, \ldots, \triangledown ⌴⌴ \cdots ⌴).$$

Nous disons qu'une arête $(q, i, c_1, \ldots, c_k) \to^a v$ est w-*valide* si $w_i = a$. Nous définissons $G_n[w]$ comme étant le sous-graphe de G_n où l'on ne conserve que les arêtes w-valides. Observons que

Lemme 2. *La machine $\mathcal{M}$ accepte l'entrée w ssi $s_n \to^* t_n$ dans $G_n[w]$.*

Ce graphe nous permet ainsi d'établir l'inclusion suivante.

Théorème 8. $\mathsf{NL} \subseteq \mathsf{AC}^1$.

Démonstration. Soient $L \in \mathsf{NL}$, $n \in \mathbb{N}$ et G_n le graphe décrit précédemment pour L et n. Décrivons le $n^{\text{ème}}$ circuit d'une famille. Celui-ci cherche à déterminer s'il existe un chemin $s_n \to^* t_n$ dans le graphe $G_n[w]$. Rappelons que nous avons vu au théorème 5 que le problème d'accessibilité appartient à AC^1. Dans notre contexte, le graphe G_n est fixe et peut être figé dans le circuit. Par contre, le graphe $G_n[w]$ est un sous-graphe de G_n obtenu dynamiquement à partir de l'entrée w. On adapte donc légèrement le circuit du théorème 5 en posant

$$\mathbf{A}^1[u, v] \coloneqq 1 \text{ si } u = v \text{ ou } (u, a, v) \in E \text{ est } w\text{-valide}, \ 0 \text{ sinon.}$$

Par le lemme 2, le circuit évalue à 1 sur entrée w ssi $w \in L$. Ainsi, $L \in \mathsf{AC}^1$. □

Soit $\mathsf{coNL} \coloneqq \{\overline{L} : L \in \mathsf{NL}\}$. Rappelons qu'à ce jour nous ne savons ni si $\mathsf{P} = \mathsf{NP}$ ni si $\mathsf{NP} = \mathsf{coNP}$. Bien qu'on ne sache pas si $\mathsf{L} = \mathsf{NL}$, surprenamment, par le théorème d'Immerman-Szelepcsényi, $\mathsf{NL} = \mathsf{coNL}$.

Nous obtenons ainsi cette chaîne d'inclusions:

$$\mathsf{NC}^0 \subseteq \mathsf{AC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} = \mathsf{coNL} \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \cdots \subseteq \mathsf{NC} = \mathsf{AC} \subseteq \mathsf{P} \begin{matrix} \subsetneq \mathsf{NP} \\ \subsetneq \mathsf{coNP} \end{matrix}$$

5.7.2 NL-complétude

Nous disons qu'un langage B est NL-*complet* s'il appartient à NL et que $A \leq_m^{\log} B$ pour tout langage $A \in \mathsf{NL}$.

Proposition 8. *Le problème* PATH *est* NL*-complet.*

Démonstration. Montrons d'abord que $\text{PATH} \in \mathsf{NL}$. S'il existe un chemin du sommet de départ s vers le sommet cible t, alors il en existe forcément un sans répétition, qui utilise ainsi au plus $|V| - 1$ arêtes, où V est l'ensemble des sommets. On résout le problème comme suit:

1. On stocke l'indice i du sommet de départ en binaire sur un ruban;
2. On initialise un compteur binaire c à zéro sur un autre ruban;
3. On choisit l'indice j d'un sommet de façon non déterministe;
4. On vérifie qu'il y a bien une arête du $i^{\text{ème}}$ sommet vers le $j^{\text{ème}}$ sommet;
5. On remplace i par j, et on incrémente c;
6. On répète les étapes 3 à 5 jusqu'à ce que i soit l'indice du sommet cible (auquel cas on accepte), ou que $c = |V|$ (auquel cas on rejette).

Si $s \to^* t$, alors au moins une branche de calcul de la machine est acceptante. Autrement, toutes les branches de calcul rejettent.

Notons que l'étape 4 cache des détails: pour vérifier l'existence d'une arête, il faut inspecter la matrice d'adjacence. Pour ce faire, on peut utiliser deux compteurs binaires auxiliaires afin d'itérer jusqu'à la $i^{\text{ème}}$ ligne et la $j^{\text{ème}}$ colonne. Comme tous les compteurs utilisent $\mathcal{O}(\log |V|) \subseteq \mathcal{O}(\log n)$ bits, nous pouvons conclure que PATH $\in$ NL.

Il nous reste à montrer que PATH est NL-difficile. Soit L un langage appartenant à NL. Posons $f(w) := (G_{|w|}[w], s_n, t_n)$. Par le lemme 2, $w \in L$ ssi $f(w) \in$ PATH. Nous avons donc $L \leq_m$ PATH. On peut se convaincre en exercice que la réduction se calcule en espace logarithmique. □

Au chapitre 3, nous avons vu que 3-SAT est NP-complet. Nous allons démontrer que 2-SAT est plus simple.

Soit $X := \{x_1, \ldots, x_k\}$ un ensemble de variables booléennes. Nous écrivons $\overline{X}$ afin de dénoter $\overline{X} := \{\neg x_1, \ldots, \neg x_k\}$. Rappelons qu'un *littéral* est une variable ou la négation d'une variable de X, c.-à-d. un élément de $X \cup \overline{X}$. Nous considérons que $\neg(\neg x_i) = x_i$. Nous disons qu'une formule φ est en *2-FNC* si elle est de la forme:

$$\varphi = \bigwedge_{1 \leq i \leq m} (\ell_i \vee \ell'_i) \qquad \text{où chaque } \ell_i, \ell'_i \in X \cup \overline{X}.$$

Nous associons un *graphe d'implication* G_φ à φ. Plus précisément, $G_\varphi = (V, E)$ est le graphe dirigé défini par:

$$V := X \cup \overline{X},$$
$$E := \{(\neg \ell_i, \ell'_i) : i \in [1..m]\} \cup \{(\neg \ell'_i, \ell_i) : i \in [1..m]\}.$$

Un *cycle contradictoire* de G_φ est un chemin de la forme $x_i \to^* \neg x_i \to^* x_i$. Par exemple, le graphe d'implication de la formule $\varphi = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_3 \vee x_1)$, illustré à la figure 5.3, contient le cycle contradictoire $x_1 \to \neg x_3 \to \neg x_2 \to \neg x_1 \to \neg x_3 \to \neg x_2 \to x_1$.

Nous allons montrer que, contrairement à 3-SAT et 3-UNSAT qui sont respectivement NP-complet et coNP-complet, les problèmes 2-SAT et 2-UNSAT sont NL-complets:

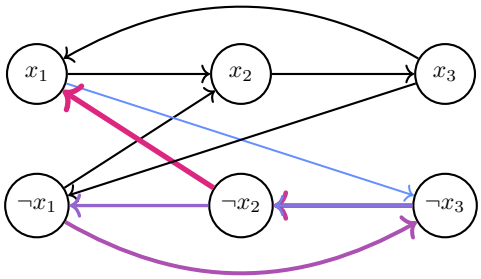


FIGURE 5.3 – Exemple d'un cycle contradictoire.

2-SAT

ENTRÉE: une formule en 2-FNC φ,
QUESTION: est-ce que φ est satisfaisable?

2-UNSAT

ENTRÉE: une formule en 2-FNC φ,
QUESTION: est-ce que φ n'est pas satisfaisable?

Lemme 3. *Soit φ une formule en 2-FNC satisfaite par une assignation* val$\colon X \cup \overline{X} \to \{\texttt{faux}, \texttt{vrai}\}$. *Si $\ell \to^* \ell'$ dans G_φ et* val$(\ell) = \texttt{vrai}$, *alors* val$(\ell') = \texttt{vrai}$.

Lemme 4. *Soit φ une formule en 2-FNC. Tous les littéraux ℓ et ℓ' de G_φ satisfont $\ell \to^* \ell'$ ssi $\neg\ell' \to^* \neg\ell$.*

Proposition 9. *Une formule φ en 2-FNC est non satisfaisable ssi G_φ contient un cycle contradictoire.*

Démonstration. $\Leftarrow$) Soit $x_i \to^* \neg x_i \to^* x_i$ un cycle contradictoire de G_φ. Afin d'obtenir une contradiction, supposons que φ est satisfaite par une assignation val. Si val$(x_i) = \texttt{vrai}$, alors, par le lemme 3, nous avons val$(\neg x_i) = \texttt{vrai}$, ce qui est impossible. Similairement, si val$(x_i) = \texttt{faux}$, alors val$(\neg x_i) = \texttt{vrai}$ et, par le lemme 3, val$(x_i) = \texttt{vrai}$, ce qui est à nouveau une contradiction.

$\Rightarrow$) Montrons la contraposée. Autrement dit, supposons que G_φ ne contient aucun cycle contradictoire, et montrons que φ est satisfaisable. Nous prétendons que cet algorithme construit une assignation qui satisfait φ:

Entrées : formule φ en 2-FNC sur variables X telle que G_φ ne contient aucun cycle contradictoire
Résultat : une assignation val : $X \cup \overline{X} \to \{\texttt{faux}, \texttt{vrai}\}$ qui satisfait φ

```
1 val := [ℓ ↦ ? : ℓ ∈ X ∪ X̄]                    // Aucun litéral assigné
  // Construire l'assignation
2 tant que ∃ℓ ∈ X ∪ X̄ tel que val(ℓ) = ? et ℓ ↛* ¬ℓ
3 |   pour ℓ′ ∈ X ∪ X̄ tel que ℓ →* ℓ′
4 |   |   val(ℓ′)  := vrai
5 |   |   val(¬ℓ′) := faux
6 retourner val
```

Il est clair que l'algorithme termine puisqu'un littéral assigné demeure assigné. De plus, l'algorithme assigne une valeur à chaque littéral, car G_φ ne contient aucun cycle contradictoire et ainsi $x \not\to^* \neg x$ ou $\neg x \not\to^* x$ pour tout $x \in X$. De plus, les lignes 4 et 5 garantissent qu'une variable et sa négation ne se contredisent pas. Il reste à prouver que l'assignation satisfait φ lorsque l'algorithme se termine.

Montrons d'abord que la boucle **pour** ne modifie pas l'assignation d'un litéral ℓ' déjà assigné. Afin d'obtenir une contradiction, supposons que c'est le cas. Cela signifie qu'il existe $\ell' \in X \cup \overline{X}$ tel que $\ell \to^* \ell'$ et $\ell \to^* \neg\ell'$. Par le lemme 4, nous avons $\neg\ell' \to^* \neg\ell$, et ainsi $\ell \to^* \neg\ell' \to^* \neg\ell$. Cela contradit le fait que $\ell \not\to^* \neg\ell$.

Montrons maintenant que si un litéral ℓ' est déjà assigné, alors la boucle **tant que** ne modifie pas son assignation. Soit ℓ le littéral choisi au début d'une itération. Supposons sans perte de généralité que $\text{val}(\ell') = \texttt{vrai}$ et qu'on tente d'assigner `faux`. Cela signifie qu'à une itération antérieure nous avons choisi g tel que $g \to^* \ell'$. De plus, comme nous tentons d'assigner `faux`, nous avons $\ell \to^* \neg\ell'$. Par le lemme 4, $\ell' \to^* \neg\ell$, et par conséquent $g \to^* \ell' \to^* \neg\ell$. Rappelons que ℓ est choisi au début de l'itération car il n'est pas assigné. Or, g a été considéré à une itération antérieure, ce qui a déjà assigné `faux` à ℓ en raison de $g \to^* \neg\ell$. Il y a donc contradiction.

Montrons finalement que lorsque l'algorithme se termine, toutes les clauses de φ sont satisfaites. Supposons qu'une clause $C = (g \vee g')$ ne l'est pas. Cela signifie que $\text{val}(g) = \text{val}(g') = \texttt{faux}$. Ces assignations ont été faites à des itérations (possiblement distinctes) de la boucle **tant que** via des littéraux ℓ et ℓ' tels que $\ell \to^* \neg g$ et $\ell' \to^* \neg g'$. Rappelons que $\neg g \to g'$ et $\neg g' \to g$ par définition de C et G_φ. Nous obtenons donc $\ell \to^* g'$ et $\ell' \to^* g$. Ainsi, ℓ ou ℓ' a renversé une assignation de `faux` à `vrai` (pour g' ou g), ce qui contredit nos observations précédentes (c.-à-d. qu'une assignation ne change jamais). □

Proposition 10. 2-UNSAT $\in$ NL.

Démonstration. Par la proposition 9, identifier un cycle contradictoire est équivalent à montrer que la formule φ en entrée n'est pas satisfaisable. Nous pouvons donc deviner une variable x_i; vérifier que $x_i \to^* \neg x_i$; puis vérifier que

$\neg x_i \to^* x_i$. Afin d'identifier ces deux chemins simples dans G_φ, nous devinons leur longueur ainsi que leurs sommets (à la volée):

Entrées : formule φ en 2-FNC sur variables $X = \{x_1, \ldots, x_k\}$
Résultat : φ est non satisfaisable?

choisir $i \in [1..k]$ de façon non déterministe
choisir $c \in [1..2k]$ de façon non déterministe `// ` $\exists c : x_i \to^c \neg x_i$?
$v \coloneqq x_i$
tant que $c > 0$
  choisir $v' \in X \cup \overline{X}$ de façon non déterministe
  si $v \not\to v'$ **alors rejeter**
  sinon $v \coloneqq v'$; $c \coloneqq c - 1$
si $v \neq \neg x_i$ **alors rejeter**
choisir $c \in [1..2k]$ **de façon non déterministe** `// ` $\exists c : \neg x_i \to^c x_i$?
tant que $c > 0$
  choisir $v' \in X \cup \overline{X}$ de façon non déterministe
  si $v \not\to v'$ **alors rejeter**
  sinon $v \coloneqq v'$; $c \coloneqq c - 1$
accepter si $v = x_i$, sinon **rejeter**

La procédure stocke un indice i, un compteur c et un sommet v. Leur représentation binaire se stocke sur $\mathcal{O}(\log k)$ bits, et par conséquent en espace logarithmique par rapport à la taille $n \geq k$ de l'entrée. □

Proposition 11. PATH $\leq_m^{\log}$ 2-UNSAT.

Démonstration. Soient $G = (V, E)$ un graphe dirigé et $s, t \in V$. Posons

$$\varphi_{G,s,t} \coloneqq (s \vee s) \wedge \bigwedge_{(x,y)\in E} (\neg x \vee y) \wedge (\neg t \vee \neg t).$$

Observons que $\varphi_{G,s,t}$ est en 2-FNC et que

$$\varphi_{G,s,t} \equiv s \wedge \bigwedge_{(x,y)\in E} (x \to y) \wedge \neg t.$$

Montrons que la fonction f définie par $f(\langle G, s, t\rangle) \coloneqq \varphi_{G,s,t}$ est une réduction. Autrement dit, nous devons montrer que $s \to^* t$ dans G ssi $\varphi_{G,s,t}$ n'est pas satisfaisable.

$\Rightarrow$) Supposons que $s \to^* t$ dans G et que $\varphi_{G,s,t}$ est satisfaite par une assignation val. Nous avons forcément $\text{val}(s) = \texttt{vrai}$ et $\text{val}(t) = \texttt{faux}$. Par définition de la formule, un raisonnement inductif montre que $\text{val}(y) = \texttt{vrai}$ pour tout $s \to^* y$. Comme $s \to^* t$, nous obtenons une contradiction.

$\Leftarrow$) Nous montrons la contraposée. Autrement dit, nous supposons que $s \not\to^* t$ et montrons que $\varphi_{G,s,t}$ est satisfaisable. Posons $S \coloneqq \{v \in V : s \to^* v\}$. Nous

prétendons que $\varphi_{G,s,t}$ est satisfaite par l'assignation définie par val$(x) \coloneqq$ `vrai` ssi $x \in S$. Comme $s \not\to^* t$, nous avons $t \notin S$ et ainsi val$(t) =$ `faux`. De plus, val$(s) =$ `vrai`. Afin d'obtenir une contradiction, supposons qu'il existe un clause $(\neg x \vee y)$ enfreinte. Cela signifie que val$(x) =$ `vrai` et val$(y) =$ `faux`. Comme val$(x) =$ `vrai`, nous avons $x \in S$ et ainsi $s \to^* x$. De plus, par définition de $\varphi_{G,s,t}$, nous avons $x \to y$. Cela implique $s \to^* y$ et ainsi $y \in S$, ce qui contradit val$(y) =$ `faux`. □

Corollaire 2. 2-SAT *et* 2-UNSAT *sont* NL*-complets.*

Démonstration. La proposition 10 montre que 2-UNSAT $\in$ NL. Comme PATH est NL-difficile, la proposition 11 implique que 2-UNSAT est NL-difficile. Par conséquent, 2-UNSAT est NL-complet. Par le théorème d'Immerman-Szelepcsényi, NL $=$ coNL, ce qui implique que 2-SAT est aussi NL-complet. □

5.8 Exercices

5.1) ♥☆✎ Montrez que ce problème appartient à AC^0: ↑↓

ENTRÉE: deux entiers $a, b \geq 0$ décrits en binaire avec n bits,
QUESTION: $a > b$?

5.2) ♥☆✎ Montrez que ce problème appartient à FAC^0: ↑↓

ENTRÉE: deux entiers $a, b \geq 0$ décrits en binaire avec n bits,
SORTIE: $\max(a, b)$.

5.3) ♡★✎ Montrez que ce problème appartient à FNC^1:

ENTRÉE: des entiers $x_1, \ldots, x_n$ décrits en binaire avec n bits,
SORTIE: $x_1 + \ldots + x_n$ sur $2n$ bits.

Pensez à une façon d'exprimer la somme de trois nombres $a+b+c$ comme la somme de deux autres nombres $d + e$.

5.4) ♡★✎ Montrez que ce problème appartient à FNC^1:

ENTRÉE: deux entiers $a, b \geq 0$ décrits en binaire avec n bits,
SORTIE: $a \cdot b$ décrit en binaire avec $2n$ bits.

5.5) ♥★≤ Montrez que ce problème est P-difficile: ↑↓

ENTRÉE: une matrice $\mathbf{A} \in \mathbb{Z}^{m \times k}$ et un vecteur $\boldsymbol{b} \in \mathbb{Z}^m$,
QUESTION: il existe $\boldsymbol{x} \in \mathbb{R}^k$ tel que $\mathbf{A}\boldsymbol{x} \leq \boldsymbol{b}$?

5.6) ♡★≤ Montrez que CVP demeure P-difficile lorsque la négation est interdite. ↑↓

5.7) ♡★≤ Une *formule de Horn* est une expression booléenne en forme normale conjonctive dont au plus un littéral par clause est sans négation. Par exemple, $(\neg x_2 \vee \neg x_4 \vee x_1) \wedge (\neg x_3 \vee \neg x_1 \vee \neg x_2)$ est une formule de Horn. Le problème HORNSAT, défini ci-dessous, se résout en temps polynomial. Montrez qu'il est P-difficile.

ENTRÉE: une formule de Horn φ,
QUESTION: φ est-elle satisfaisable?

5.8) ♡☆■ Nous avons défini la P-difficulté à l'aide de $\leq_m^{\log}$. Montrez que la P-difficulté serait inutile si on la définissait avec $\leq_m^{\text{poly}}$. ↑↓

5.9) ♡★■ Montrez qu'un circuit de profondeur $\mathcal{O}(\log n)$ dont les portes sont d'entrance deux a forcément un nombre polynomial de portes.

6

Complexité du calcul : au-delà de NP

Couvert par un autre enseignant. Voir les références fournies par cette personne.

7

Calcul quantique

Couvert par un autre enseignant. Voir les références fournies par cette personne.

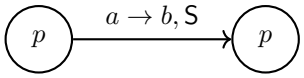
Annexes

Solutions des exercices

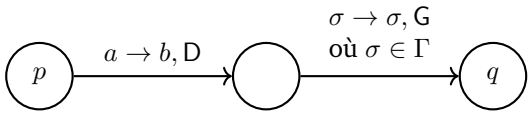
Cette section présente des solutions à certains des exercices du document. Dans certains cas, il ne s'agit que d'ébauches de solutions.

Chapitre 1

1.1) On convertit chaque transition

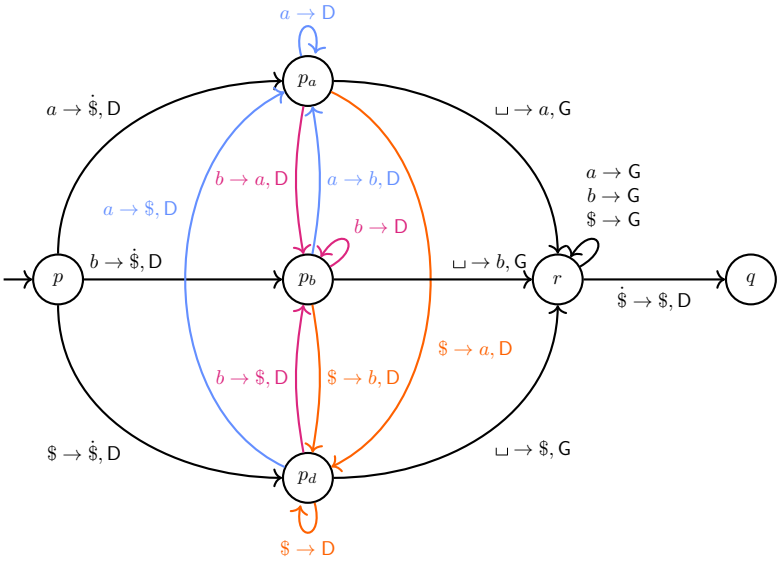


par



Attention, on émule S par DG et non GD car à la première case, cela ferait rebondir la tête.

1.2) On mémorise la lettre actuelle; on écrit $\dot{\sigma}$; puis on recommence à droite avec la lettre mémorisée jusqu'à atteindre le bout du ruban. On revient ensuite jusqu'à $\dot{\sigma}$ et on retire le point. Par exemple, pour $\sigma = \$$ et $\Gamma = \{a, b, \$, \sqcup\}$, on construit cette machine:



Très formellement, la machine est définie par les états $Q \coloneqq \{p, r, q\} \cup \{p_x :$

$x \in \Gamma \setminus \{\sqcup\}$ et ces transitions:

$$\begin{aligned}
\delta(p, x) &= (p_x, \dot{\sigma}, \mathsf{D}) && \text{pour } x \in \Gamma \setminus \{\sqcup\},\\
\delta(p_x, y) &= (p_y, x, \mathsf{D}) && \text{pour } x, y \in \Gamma \setminus \{\sqcup\},\\
\delta(p_x, \sqcup) &= (r, x, \mathsf{G}) && \text{pour } x \in \Gamma \setminus \{\sqcup\},\\
\delta(r, x) &= (r, x, \mathsf{G}) && \text{pour } x \in \Gamma \setminus \{\sqcup\},\\
\delta(r, \dot{\sigma}) &= (q, \sigma, \mathsf{D}).
\end{aligned}$$

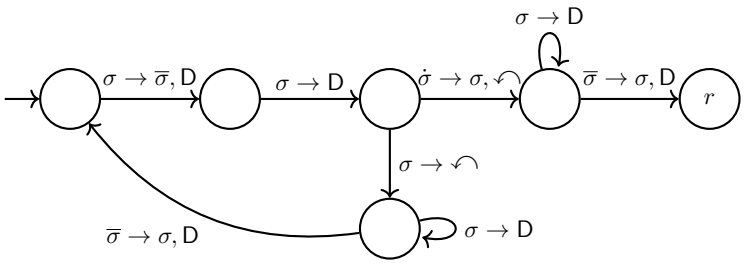
1.7) Soit $\mathcal{M}$ une machine de Turing. Nous devons la simuler par une machine à retour $\mathcal{M}'$. On peut trivialement simuler les déplacements à droite. Expliquons comment simuler les déplacements à gauche à l'aide d'un exemple. Supposons que le ruban soit actuellement comme suit et que l'on cherche à simuler la transition $\delta(q, a) = (r, c, \mathsf{G})$:

				$\overset{q}{\triangledown}$				
a	a	b	a	a	b	b	$\sqcup$	$\cdots$

Nous remplaçons a par $\dot{c}$ afin d'effectuer l'écriture et de mémoriser la position de la tête, et nous retournons au début:

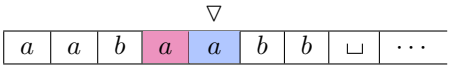
	$\triangledown$							
a	a	b	a	$\dot{c}$	b	b	$\sqcup$	$\cdots$

Nous devons maintenant ramener la tête à gauche de $\dot{c}$, ce qu'on ne peut pas accomplir en une seule étape. Pour y arriver, on cherche la case qui se situe deux cases avant celle de $\dot{c}$. Pour ce faire, on inspecte la lettre σ à la première case $i = 1$; on la marque avec $\overline{\sigma}$; et on regarde deux cases plus loin. Si on trouve le point, alors on l'efface; on retourne à la case marquée par une ligne; on efface la ligne; et on se déplace à la case de droite. Sinon, on recommene à la case $i + 1$. Cela correspond à cette machine, où σ dénote une lettre arbitraire de Γ:

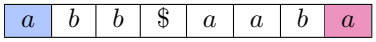


L'approche ci-dessus, suppose que la lettre marquée par un point possède au moins deux cases à sa gauche. Il faut gérer les deux autres cas limites séparément.

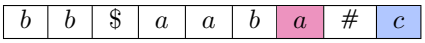
1.8) Soit $\mathcal{M}$ une machine de Turing. Nous devons simuler $\mathcal{M}$ par un automate à file $\mathcal{M}'$. Initialement, $\mathcal{M}'$ enfile la lettre \$ afin de marquer la fin du ruban. Voyons comment représenter une configuration de $\mathcal{M}$ à l'aide d'un exemple. Cette configuration de $\mathcal{M}$:



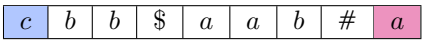
est représentée par cette file:



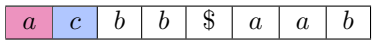
Les déplacements à droite sont simples à simuler. Expliquons comment simuler les déplacements à gauche à l'aide d'un exemple. Supposons que le ruban soit actuellement comme ci-dessus et que l'on cherche à simuler la transition $\delta(q, a) = (r, c, \mathsf{G})$. On défile la lettre a, puis on enfile $\#c$:



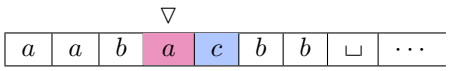
Ensuite, on défile une lettre σ qu'on mémorise temporairement dans un état; et on défile la prochaine lettre σ'. Si $\sigma' \neq \#$, alors on enfile σ et on mémorise σ'. Si $\sigma' = \#$, alors on enfile $\#$, puis on enfile σ. Après avoir effectué ces opérations, $\#$ a été permutée avec son voisin de gauche:



À partir d'ici, on défile et renfile jusqu'à trouver $\#$ qu'on ne renfile pas:



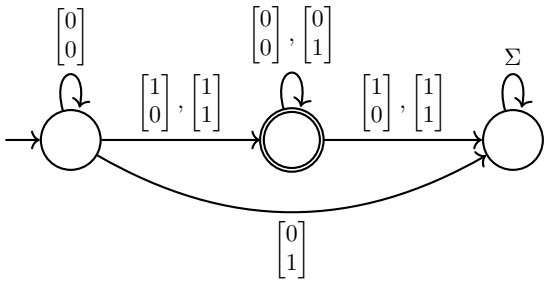
La file ci-dessus représente la configuration attendue:



Il y a un cas limite à gérer: si on tente d'écrire sur \$, alors il faut d'abord insérer un blanc. Pour y arriver, on peut défiler \$; enfiler $\#␣\$$; défiler juqsu'à $\#$ qu'on ne renfile pas.

Chapitre 4

4.1) Il suffit de donner un automate qui représente $[\![\psi]\!]$:



4.2) À l'aide ces formules:

$$\psi(x,x),$$
$$\psi(x,y) \wedge \forall x' \ ((x' > x) \to \neg\psi(x',y)).$$

4.3) On peut remplacer 0 par la variable z, et préfixer la formule par $\exists z \ (z = z+z)$. On peut remplacer 1 par u, et ajouter $(z < u) \wedge \neg(\exists x \ (z < x < u))$.

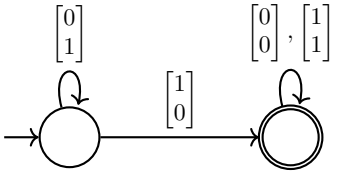
4.4) Le maximum s'exprime par $((x \leq y) \to (z = y)) \wedge ((y \leq x) \to (z = x))$.

4.5) Toute formule de l'arithmétique de Peano se réécrit dans cette logique. En effet, il suffit de substituer $z = xy$ par

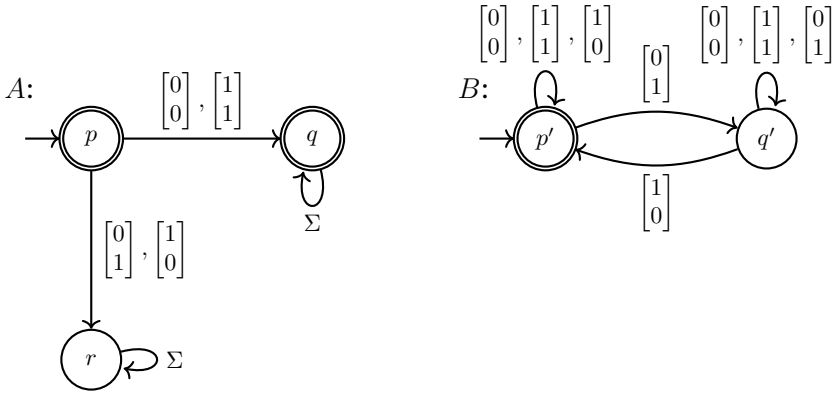
$$(z = y = 0) \vee (z = x = 0) \vee$$
$$[x > 0 \wedge y > 0 \wedge z > 0 \wedge \big(z \div y = x \wedge (z-1) \div y \neq x\big)].$$

4.6) Dans la preuve d'indécidabilité de la section 4.1, nous avons utilisé b^x. Nous avons choisi b comme un nombre suffisamment grand pour coder $\Gamma \cup Q \cup \{\#\}$. Nous pouvons choisir b encore plus grand et ne simplement pas utiliser les chiffres additionnels. Ainsi, nous pouvons choisir b comme une puissance de deux. Ensuite, il est simple d'exprimer b^x. Par exemple, si l'on choisit $b = 8$, alors on peut exprimer $y = 8^x$ par $\exists z \ (y = 2^z \wedge z = 3x)$.

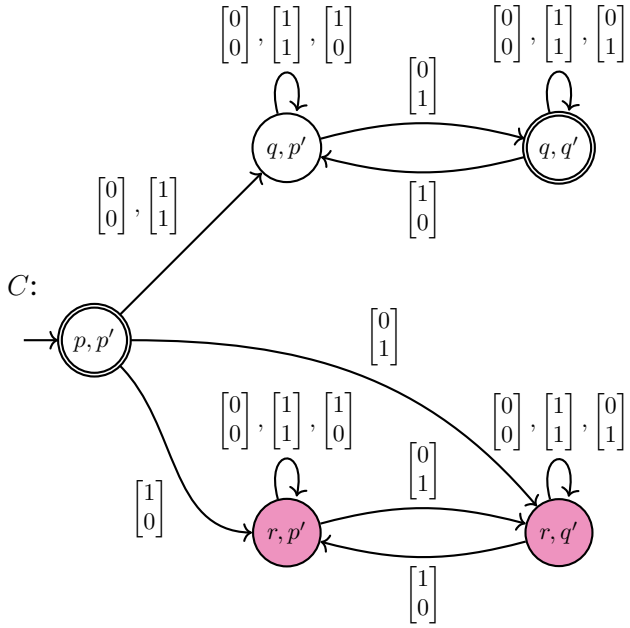
4.7)



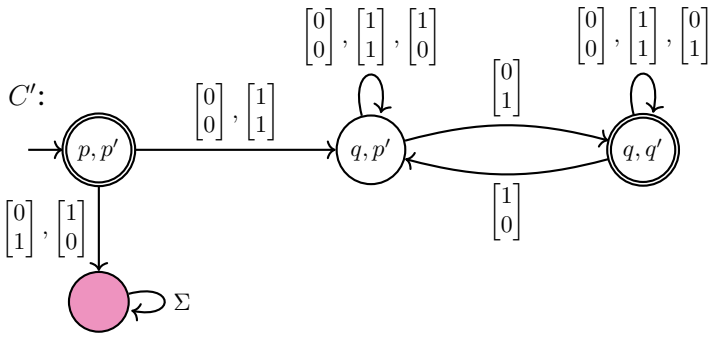
4.8) Construisons d'abord des automates pour les deux côtés de la conjonction:



Intersectons maintenant les deux automates:



Bien que ce ne soit pas nécessaire, remarquons que les états colorés peuvent être contractés en un seul état puits rejetant:



4.9) Soient $x_1, \ldots, x_n$ les variables libres. Afin de déterminer si $\varphi \equiv \psi$, on peut simplement vérifier si cette formule est vraie:

$$\forall x_1, \ldots x_n \\ (\varphi(x_1, \ldots, x_n) \to \psi(x_1, \ldots, x_n)) \wedge (\psi(x_1, \ldots, x_n) \to \varphi(x_1, \ldots, x_n)).$$

4.10)

$$\text{config}(x) \wedge \text{config}(y) \wedge \\ \forall i \, \big((x[i] = p \wedge x[i+1] = a) \to (y[i] = b \wedge y[i+1] = q)\big).$$

4.11)

a) On utilise cette observation: « x est une puissance de b si et seulement si le seul nombre premier qui divise x est b »:

$$\text{puiss}_b(x) \coloneqq \forall p \, [((\text{premier}(p) \wedge (\exists q \, (q \geq 1 \wedge p \cdot q = x))) \to (p = b)].$$

b) Ces macros supposent que $d \leq e$ sont des puissances de b:

$$x[d] \coloneqq (x \div d) \bmod b, \\ x[d:e] \coloneqq (x \div d) \bmod ((e \div d) \cdot b).$$

c) Nous calquons la réduction de la section 4.1 en modifiant chaque macro. Nous pouvons supposer que b est un nombre premier. En effet, rappelons que nous avons défini $b \coloneqq |\Gamma| + |Q| + 1$ afin que b puisse codé chaque symbole. Si b n'est pas premier, alors nous pouvons simplement choisir b comme le prochain nombre premier. Voici

les nouvelles macros:

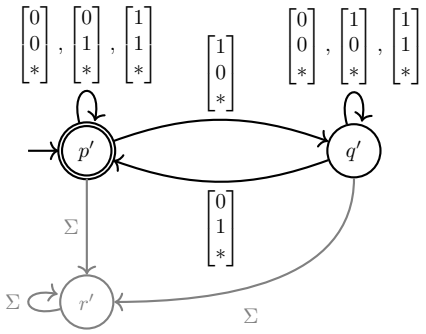
$$\begin{aligned}
\text{config}(y) &\coloneqq (y[1] = \#) \wedge \\
&\quad (\forall d\ ((d > 1) \wedge \text{puiss}_b(d)) \to (y[d] \neq \#)) \wedge \\
&\quad (\exists! e\ (\text{puiss}_b(e) \wedge y[e] \in Q)), \\
\text{succ}(x, d, e, f) &\coloneqq (d < e < f) \wedge \\
&\quad \text{puiss}_b(d) \wedge \text{puiss}_b(e) \wedge \text{puiss}_b(f) \wedge \\
&\quad \text{config}(x[d : e \div b]) \wedge \text{config}(x[e : f \div b]) \wedge \\
&\quad \big[(x[f] = \#) \vee \\
&\qquad \big(\forall g\ (g \geq f \wedge \text{puiss}_b(g)) \to (x[g] = \sqcup)\big)\big], \\
\text{première}(x, e) &\coloneqq \text{config}(x[1 : e]) \wedge \big[(x[e \cdot b] = \#) \vee \\
&\quad (\forall f\ (f > e \wedge \text{puiss}_b(f)) \to (x[f] = \sqcup))\big], \\
\text{init}(x) &\coloneqq \exists d\ \text{puiss}_b(d) \wedge \text{première}(x, d) \wedge \\
&\qquad \left(x[b : d] = q_0 + \sum_{i=1}^{|w|} w_i \cdot b^i\right), \\
\text{historique}(x) &\coloneqq \text{init}(x) \wedge \\
&\quad \Big[\forall d, e, f\ \Big(\text{succ}(x, d, e, f) \wedge \\
&\qquad \text{puiss}_b(d) \wedge \text{puiss}_b(e) \wedge \text{puiss}_b(f)\Big) \to \\
&\qquad \text{trans}(x[d : e \div b], x[e : f \div b])\Big], \\
\text{arrêt}(x) &\coloneqq \exists d, e\ \text{puiss}_b(d) \wedge \text{puiss}_b(e) \wedge \text{config}(x[d : e]) \wedge \\
&\quad \big[\exists f\ (d < f \leq e) \wedge \text{puiss}_b(f) \wedge \\
&\qquad ((x[f] = q_{\text{acc}}) \vee (x[f] = q_{\text{rej}}))\big], \\
\varphi_{\mathcal{M},w} &\coloneqq \exists x\ \text{historique}(x) \wedge \text{arrêt}(x).
\end{aligned}$$

Remarquons que le terme « $q_0 + \sum_{i=1}^{|w|} w_i \cdot b^i$ » semble utiliser l'exponentiation. Or, ce nombre ne dépend que du mot w, et peut ainsi être précalculé et figé dans la formule.

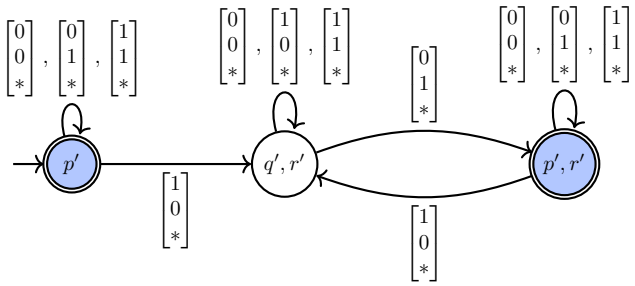
4.12) Rappelons que $\sqrt{2}$ est irrationnel. Ainsi, la formule $\exists x\ (x \cdot x = 2)$ est seulement vraie sur $\mathbb{R}$. Nous pourrions également prendre la formule

$$\forall y\ ((y > 0) \to (\exists x\ (x \cdot x = y))).$$

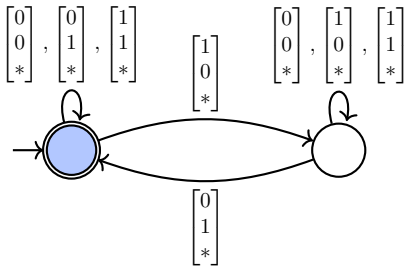
4.13) Rappelons que nous avons construit cet automate non déterministe pour la formule $\exists z\ (y = x + z)$:



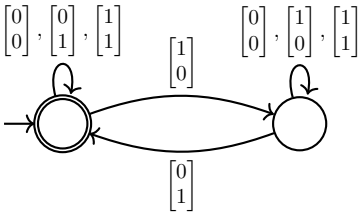
Déterminisons-le:



Bien que ce ne soit pas nécessaire, remarquons que les états colorés peuvent être contractés en un seul état, sans que cela change le langage accepté:



En omettant la (troisième) composante pour z, nous obtenons:



Rappelons que l'automate représente $[\![\exists z\ (y = x + z)]\!] = [\![x \leq y]\!]$.

4.16) ↑↓

a) Nout utilisons l'alphabet $\{0,1,\#\}$ où $\#$ est un délimiteur, et où la $i^{\text{ème}}$ composante indique en binaire le nombre de fois que le $i^{\text{ème}}$ nombre premier apparaît. Par exemple, $01\#0\#1$ code $20 = 2^2 \cdot 5^1$ et $1\#1\#1$ représente $30 = 2^1 \cdot 3^1 \cdot 5^1$. Nous étendons ce codage à $\mathbb{N}^V$ avec l'alphabet $\Sigma := \{0,1,\#\}^V$. Par exemple $(20,30)$ est représenté par

$$\begin{bmatrix} 01\#0\#1 \\ 10\#1\#1 \end{bmatrix}.$$

Chaque assignation a une infinité de codages: avec zéros non significatifs et $\#$ superflus. Par exemple, voici d'autres codages de $(20,30)$:

$$\begin{bmatrix} 010\#0\#1 \\ 100\#1\#1 \end{bmatrix}, \begin{bmatrix} 01\#0\#1\#\# \\ 10\#1\#1\#\# \end{bmatrix}, \begin{bmatrix} 010\#0\#1\#0\# \\ 100\#1\#1\#0\# \end{bmatrix}, \ldots$$

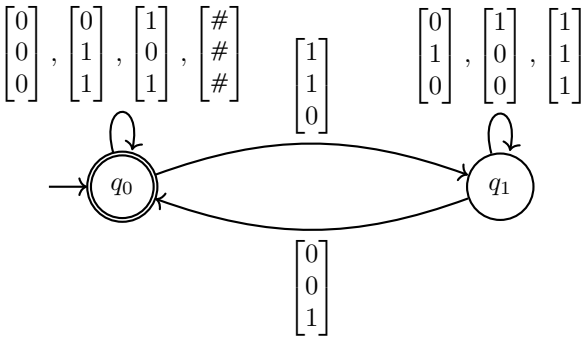
b) Considérons l'addition de multiensembles finis de $\mathbb{P}$. Par exemple, $\{2,2,5\} + \{2,3,5\} = \{2,2,2,3,5,5\}$, ce qui se code par

$$\begin{bmatrix} 01\#0\#10 \\ 10\#1\#10 \\ 11\#1\#01 \end{bmatrix}.$$

Indirectement, cela représente la multiplication d'entiers naturels; l'exemple ci-dessus représente

$$20 \cdot 30 = 2^2\ 3^0\ 5^1 \cdot 2^1\ 3^1\ 5^1 = 2^3\ 3^1\ 5^2 = 600.$$

Cet automate représente l'addition de multiensembles finis de $\mathbb{P}$:



La conjonction, la négation et la quantification existentielle s'implémentent essentiellement comme pour l'arithmétique de Presburger.

c) Comme pour l'arithmétique de Presburger, nous construisons récursivement un automate pour une formule donnée, puis nous vérifions si celui-ci accepte $(\{0,1\}^V \cup \{\#\}^V)^*$.

4.17) ↑↓

a) Nous utilisons d'abord ces règles afin d'amener les négations le plus à l'intérieur possible:

$$\neg\exists x\ \varphi \equiv \forall x\ \neg\varphi,$$
$$\neg\forall x\ \varphi \equiv \exists x\ \neg\varphi.$$

Nous utilisons ensuite ces règles pour amener les quantificateurs à gauche, où $Q \in \{\exists, \forall\}$ et $\circ \in \{\vee, \wedge\}$:

$$(Qx\ \varphi) \circ \psi \equiv Qx'\ (\varphi[x \mapsto x'] \circ \psi),$$
$$\varphi \circ (Qx\ \psi) \equiv Qx'\ (\varphi \circ \psi[x \mapsto x']).$$

Par exemple

$$(\exists x\ (x > y)) \wedge (x = y) \equiv \exists x'\ ((x' > y) \wedge (x = y)).$$

b) La clôture sous union est triviale puisque l'union de deux unions finies est une union finie. Puisque

$$X \cap Y = \overline{\overline{X} \cup \overline{Y}},$$

il suffit de démontrer la clôture sous complémentation. Soit $X \subseteq \mathbb{R}^V$ un ensemble semi-algébrique. Par définition,

$$X = \bigcup_{\ell=1}^{k} \left\{ \boldsymbol{x} \in \mathbb{R}^V : \bigwedge_{i=1}^{m_\ell} (p_{\ell,i}(\boldsymbol{x}) = 0) \wedge \bigwedge_{j=1}^{n_\ell} (q_{\ell,j}(\boldsymbol{x}) > 0) \right\}.$$

Ainsi,

$$\begin{aligned}\overline{X} &= \overline{\left\{ \boldsymbol{x} \in \mathbb{R}^V : \bigvee_{\ell=1}^{k} \left(\bigwedge_{i=1}^{m_\ell} (p_{\ell,i}(\boldsymbol{x}) = 0) \wedge \bigwedge_{j=1}^{n_\ell} (q_{\ell,j}(\boldsymbol{x}) > 0) \right) \right\}} \\ &= \Bigg\{ \boldsymbol{x} \in \mathbb{R}^V : \underbrace{\bigwedge_{\ell=1}^{k} \left(\bigvee_{i=1}^{m_\ell} (p_{\ell,i}(\boldsymbol{x}) \neq 0) \vee \bigvee_{j=1}^{n_\ell} (q_{\ell,j}(\boldsymbol{x}) \leq 0) \right)}_{\psi :=} \Bigg\}.\end{aligned}$$

Remarquons que

$$\psi \equiv \bigwedge_{\ell=1}^{k} \Bigg(\bigvee_{i=1}^{m_\ell} (p_{\ell,i}(\boldsymbol{x}) > 0 \vee -p_{\ell,i}(\boldsymbol{x}) > 0) \vee \bigvee_{j=1}^{n_\ell} (q_{\ell,j}(\boldsymbol{x}) = 0 \vee -q_{\ell,j}(\boldsymbol{x}) > 0) \Bigg).$$

Mettons ψ en forme normale disjonctive:

$$\psi \equiv \bigvee_{\ell=1}^{k'} \bigwedge_{i=1}^{m'} \psi_{\ell,i}.$$

Nous avons

$$\begin{aligned}\overline{X} &= \left\{\boldsymbol{x} \in \mathbb{R}^V : \psi(\boldsymbol{x})\right\} \\ &= \left\{\boldsymbol{x} \in \mathbb{R}^V : \bigvee_{\ell=1}^{k'} \bigwedge_{i=1}^{m'} \psi_{\ell,i}(\boldsymbol{x})\right\} \\ &= \bigcup_{\ell=1}^{k'} \left\{\boldsymbol{x} \in \mathbb{R}^V : \bigwedge_{i=1}^{m'} \psi_{\ell,i}(\boldsymbol{x})\right\}.\end{aligned}$$

Nous avons terminé puisque chaque $\psi_{\ell,i}$ est de la forme $r(\boldsymbol{x}) = 0$ ou $r(\boldsymbol{x}) > 0$, où r est un polynôme. □

c) Par hypothèse:

$$Y = \left\{\boldsymbol{x} \in \mathbb{R}^{V\setminus\{v\}} : \bigwedge_{i=1}^{m} (p_i(\boldsymbol{x}) = 0) \wedge \bigwedge_{j=1}^{n} (q_j(\boldsymbol{x}) > 0)\right\}.$$

Posons

$$X \coloneqq \{\boldsymbol{x} \in \mathbb{R}^V : \pi_v(\boldsymbol{x}) \in Y\}$$

Soit $p'_i \in \mathbb{R}[V]$ le polynôme identique à $p_i \in \mathbb{R}[V\setminus\{v\}]$ où la variable v n'est simplement pas utilisée. Définissons q'_j pareillement par rapport à q_j. L'ensemble X est semi-algébrique car

$$\begin{aligned}X &= \left\{\boldsymbol{x} \in \mathbb{R}^V : \bigwedge_{i=1}^{m} (p_i(\pi_v(\boldsymbol{x})) = 0) \wedge \bigwedge_{j=1}^{n} (q_j(\pi_v(\boldsymbol{x})) > 0)\right\} \\ &= \left\{\boldsymbol{x} \in \mathbb{R}^V : \bigwedge_{i=1}^{m} (p'_i(\boldsymbol{x}) = 0) \wedge \bigwedge_{j=1}^{n} (q'_j(\boldsymbol{x}) > 0)\right\}.\end{aligned}$$ □

d) Posons $Y \coloneqq \pi_x(X)$. Par définition, $(a,b,c) \in Y$ ssi il existe $x \in \mathbb{R}$ tel que $a^2x + bx + c = 0$. Lorsque $a > 0$, nous avons $(a,b,c) \in Y$ ssi $b^2 - 4ac \geq 0$, puisque les racines du polynôme sont

$$x \in \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Lorsque $a = 0$, nous avons $(a,b,c) \in Y$ ssi $b > 0$ ou $b = c = 0$. Ainsi,

$$\begin{aligned}&(a,b,c) \in Y \\ \iff\ &(a > 0 \wedge b^2 - 4ac = 0) \vee (a > 0 \wedge b^2 - 4ac > 0) \vee \\ &(a = 0 \wedge b > 0) \vee (a = 0 \wedge b = 0 \wedge c = 0).\end{aligned}$$

Par conséquent, $Y = \pi_x(X)$ est semi-algébrique comme attendu.

e) Considérons une formule $\varphi \in \mathsf{FO}(\mathbb{R}, \leq, +, \cdot)$. Nous pouvons considérer sans perte de généralité que φ est en forme prénexe:

$$\varphi = Q_1 v_1 \; Q_2 v_2 \; \cdots Q_k v_k \; \psi \text{ où } Q_i \in \{\exists, \forall\}.$$

Montrons que $[\![\psi]\!]$ est semi-algébrique par induction. Comme les deux seules opérations arithmétiques permises sont l'addition et la multiplication, si ψ est une formule atomique, alors elle est forcément de la forme $r(\boldsymbol{x}) \leq r'(\boldsymbol{x})$, où $r, r' \in \mathbb{R}[V]$ sont des polynômes. Dans ce cas, l'ensemble $[\![\psi]\!]$ est semi-algébrique puisque

$$[\![\psi]\!] = \left\{\boldsymbol{x} \in \mathbb{R}^V : (r(\boldsymbol{x}) - r'(\boldsymbol{x}) = 0) \vee (r'(\boldsymbol{x}) - r(\boldsymbol{x}) > 0)\right\}.$$

Pour le cas général, considérons $\psi = \psi_1 \wedge \psi_2$. Par induction, chaque $[\![\psi_i]\!]$ est semi-algébrique. Puisque $[\![\psi]\!] = [\![\psi_1]\!] \cap [\![\psi_2]\!]$ et que les ensemble semi-algébriques sont clos par intersection, l'ensemble $[\![\psi]\!]$ est semi-algébrique. Le cas de la disjonction est similaire avec l'union, et pareillement pour la négation avec la complémentation.

Montrons maintenant que $[\![\varphi]\!]$ est semi-algébrique. Nous procédons par induction sur k. Si $k = 0$, alors l'affirmation est triviale. Supposons que $k > 0$. Par hypothèse d'induction, $[\![Q_2 v_2 \; \cdots Q_k v_k \; \psi]\!]$ est un ensemble semi-algébrique X. Considérons d'abord le cas où $Q_1 = \exists$. Par le théorème de Tarski–Seidenberg, $\pi_{v_1}(X)$ est semi-algébrique. Remarquons que

$$\begin{aligned}[\![\varphi]\!] &= \{\boldsymbol{x} \in \mathbb{R}^V : \exists \alpha \in \mathbb{R} \text{ t.q. } \boldsymbol{x}[v_1 \mapsto \alpha] \in X\} \\ &= \{\boldsymbol{x} \in \mathbb{R}^V : \pi_{v_1}(\boldsymbol{x}) \in \pi_{v_1}(X)\}.\end{aligned}$$

Par le point 4.17)c, cet ensemble est semi-algébrique.

Considérons maintenant le cas où $Q_1 = \forall$. Remarquons que

$$\begin{aligned}[\![\varphi]\!] &= [\![\neg\exists v_1 \neg Q_2 v_2 \; \cdots Q_k v_k \; \psi]\!] \\ &= \overline{[\![\exists v_1 \neg Q_2 v_2 \; \cdots Q_k v_k \; \psi]\!]} \\ &= \overline{\{\boldsymbol{x} \in \mathbb{R}^V : \pi_{v_1}(\boldsymbol{x}) \in \pi_{v_1}(\overline{X})\}}.\end{aligned}$$

Comme les ensembles semi-algébriques sont clos sous complémentation, l'ensemble ci-dessus est semi-algébrique. □

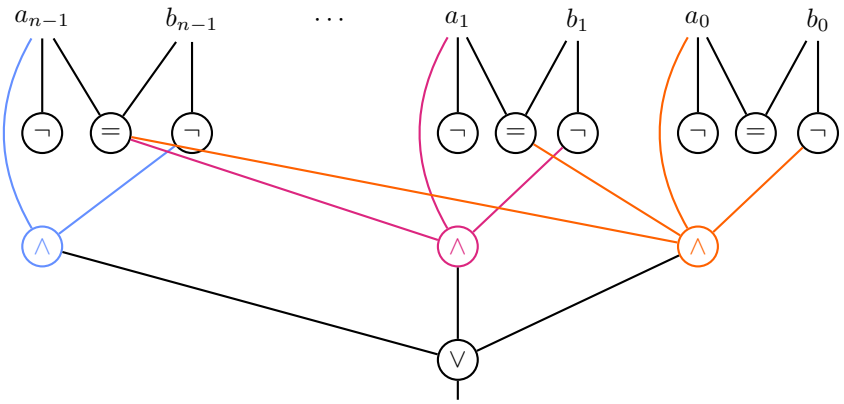
f) Étant donné une formule $\varphi \in \mathsf{FO}(\mathbb{R}, \leq, +, \cdot)$ sans variable libre, on calcule une représentation semi-algébrique de $[\![\varphi]\!]$. Comme φ n'a pas de variable libre, nous avons forcément $[\![\varphi]\!] = \mathbb{R}^V$ (lorsque $\varphi \equiv \texttt{vrai}$) ou $[\![\varphi]\!] = \emptyset$ (lorsque $\varphi \equiv \texttt{faux}$). Ainsi,

$$\varphi \equiv \texttt{vrai} \iff [\![\varphi]\!] = \mathbb{R}^V \iff \mathbf{0} \in [\![\varphi]\!].$$

Il suffit donc d'évaluer les polynômes de la représentation de $[\![\varphi]\!]$ avec toutes les variables à zéro, et de vérifier si l'une des conjonctions évalue à `vrai`.

Chapitre 5

5.1)



Ici, une porte de la forme $x = y$ s'implémente avec $(x \wedge y) \vee (\neg x \wedge \neg y)$.

5.2) Il suffit d'utiliser le circuit de l'exercice 5.2) afin d'identifier le plus grand nombre, puis de copier les bits du plus grand nombre en sortie.

5.5) On donne une réduction à partir de CVP. Chaque porte est convertie comme suit:

Porte	$x_i = w_i$	$y = \neg x$	$z = x \vee y$	$z = x \wedge y$
Contraintes	$x_i = w_i$	$0 \leq y \leq 1$ $y = 1 - x$	$0 \leq z \leq 1$ $x \leq z$ $y \leq z$ $z \leq x + y$	$0 \leq z \leq 1$ $z \leq x$ $z \leq y$ $x + y - 1 \leq z$

On ajoute la contrainte $s = 1$ où s est la sortie du circuit. Le système possède une solution ssi le circuit évalue à vrai.

Notons que techniquement toutes les contraintes doivent être de la forme « $\leq$ ». Or, on peut aisément les écrire sous cette forme car $p = q$ est équivalent à $\{p \leq q, p \geq q\}$, et $p \geq q$ est équivalent à $-p \leq -q$.

5.6) Nous donnons une réduction à partir de CVP. Nous remplaçons chaque porte x par x et $\overline{x}$ comme suit:

Avant	**Après**	
$x_i = w_i$	$x_i = w_i$	$\overline{x_i} = \neg w_i$
$w = u \wedge v$	$w = u \wedge v$	$\overline{w} = \overline{u} \vee \overline{v}$
$w = u \vee v$	$w = u \vee v$	$\overline{w} = \overline{u} \wedge \overline{v}$
$w = \neg u$	$w = \overline{u}$	$\overline{w} = \overline{u}$

5.8) Considérons la variante de la P-complétude. Soit $B \in \mathsf{P}$ où $B \notin \{\emptyset, \Sigma^*\}$. Montrons que B est P-difficile. Soit $A \in \mathsf{P}$. Nous définissons une réduction f qui, sur entrée w, détermine en temps polynomial si $w \in A$, et retourne un mot de B ou $\overline{B}$ selon le résultat. Clairement, $A \leq_m^{\text{poly}} B$. Ainsi, B est P-complet. Par conséquent, essentiellement tous les langages de P sont P-complets.

Démonstrations

Proposition 4. *Soit* $u \coloneqq \mathrm{casc}(\bigcirc_{1\leq i\leq n} x_i)$ *où* $\circ \in \{\vee, \wedge\}$. *Nous avons* $\mathrm{taille}(u) = n-1$ *et* $\mathrm{prof}(u) \leq \lceil \log n \rceil$.

Démonstration. Nous procédons par induction sur le nombre de termes. La proposition est trivialement vraie lorsque $n \leq 2$. Pour $n > 2$, nous avons:

$$\begin{aligned}
&\mathrm{taille}(\mathrm{casc}(\bigcirc_{1\leq i\leq n} x_i)) \\
&= \mathrm{taille}(\mathrm{casc}(\bigcirc_{1\leq i\leq n\div 2} x_i)) + \mathrm{taille}(\mathrm{casc}(\bigcirc_{n\div 2<i\leq n} x_i)) + 1 && \text{(par déf.)} \\
&= (\lfloor n/2 \rfloor - 1) + (\lceil n/2 \rceil - 1) + 1 && \text{(par ind.)} \\
&= n - 1.
\end{aligned}$$

De plus,

$$\begin{aligned}
&\mathrm{prof}(\mathrm{casc}(\bigcirc_{1\leq i\leq n} x_i)) \\
&= \max\left(\mathrm{prof}(\mathrm{casc}(\bigcirc_{1\leq i\leq n\div 2} x_i)), \mathrm{prof}(\mathrm{casc}(\bigcirc_{n\div 2<i\leq n} x_i))\right) + 1 && \text{(par déf.)} \\
&\leq \max(\lceil \log \lfloor n/2 \rfloor \rceil, \lceil \log \lceil n/2 \rceil \rceil) + 1 && \text{(par ind.)} \\
&= \lceil \log \lceil n/2 \rceil \rceil + 1 \\
&= \lceil \log \lceil n/2 \rceil + 1 \rceil \\
&= \lceil \log \lceil n/2 \rceil + \log 2 \rceil \\
&= \lceil \log 2 \lceil n/2 \rceil \rceil \\
&= \lceil \log n \rceil. && \square
\end{aligned}$$

Proposition 5. *Nous avons* $\mathrm{prof}(r_i) = 2i + 1$ *et* $\mathrm{prof}(c_i) = \max(2i + 2, 1)$ *pour tout* $0 \leq i < n$.

Démonstration. Nous procédons par induction sur i. Le cas de base est trivial.

Soit $i \geq 1$. Nous avons

$$\begin{aligned}
\text{prof}(r_i) &= \text{prof}((a_i \wedge b_i) \vee ((a_i \vee b_i) \wedge r_{i-1})) \\
&= \max(\text{prof}(a_i \wedge b_i), \text{prof}((a_i \vee b_i) \wedge r_{i-1})) + 1 \\
&= \max(1, \max(\text{prof}(a_i \vee b_i), \text{prof}(r_{i-1})) + 1) + 1 \\
&= \max(1, \max(1, \text{prof}(r_{i-1})) + 1) + 1 \\
&= \max(2, 3, \text{prof}(r_{i-1}) + 2) \\
&= \max(2, 3, 2(i-1) + 3) && \text{(par induction)} \\
&= 2(i-1) + 3 && \text{(car } i \geq 1\text{)} \\
&= 2i + 1.
\end{aligned}$$

Afin de compléter la preuve, remarquons que $\text{prof}(c_0) = 1$, et que, pour tout $i > 0$, nous avons

$$\text{prof}(c_i) = \max(1, \text{prof}(r_i)) + 1 = \max(1, 2i+1) + 1 = \max(2, 2i+2) = 2i+2. \quad \square$$

Proposition 6. *Nous avons $r_i = r'_i$ pour tout $0 \leq i < n$, et $c_i = c'_i$ pour tout $0 \leq i \leq n$.*

Démonstration. Il suffit de montrer que $r_i = r'_i$, puisque $c_i = c'_i$ en découle directement. Nous avons trivialement $r_0 = r'_0$. Soit $i > 0$. Nous avons

$$\begin{aligned}
& r_i \\
&= (a_i \wedge b_i) \vee ((a_i \vee b_i) \wedge r_{i-1}) \\
&= (a_i \wedge b_i) \vee ((a_i \vee b_i) \wedge r'_{i-1}) && \text{(par ind.)} \\
&= (a_i \wedge b_i) \vee \left((a_i \vee b_i) \wedge \bigvee_{i-1 \geq j \geq 0} \left((a_j \wedge b_j) \wedge \bigwedge_{i-1 \geq k > j} (a_k \vee b_k) \right) \right) && \text{(déf. de } r'_{i-1}\text{)} \\
&= (a_i \wedge b_i) \vee \bigvee_{i-1 \geq j \geq 0} \left((a_j \wedge b_j) \wedge \bigwedge_{i \geq k > j} (a_k \vee b_k) \right) && \text{(par distrib.)} \\
&= \bigvee_{i \geq j \geq 0} \left((a_j \wedge b_j) \wedge \bigwedge_{i \geq k > j} (a_k \vee b_k) \right) \\
&= r'_i && \text{(déf. de } r'_i\text{).} \quad \square
\end{aligned}$$

Théorème 4. $\text{ADD} \in \text{FAC}^0$.

Démonstration. Nous analysons plus rigoureusement la profondeur du circuit. Les autres détails ont déjà été prouvés. Il est clair que $\text{prof}(r_0) = 1$. Pour $i > 0$,

nous avons

$$\begin{aligned}
&\text{prof}(r'_i)\\
&= \text{prof}(\bigvee_{i\geq j\geq 0}((a_j \wedge b_j) \wedge \bigwedge_{i\geq k>j}(a_k \vee b_k)))\\
&= \max\left\{\text{prof}((a_j \wedge b_j) \wedge \bigwedge_{i\geq k>j}(a_k \vee b_k)) + 1 : i \geq j \geq 0\right\} + 1\\
&= \max\left\{\max(1, \max\left\{\text{prof}(a_k \vee b_k) : i \geq k > j\right\} + 1) + 1 : i \geq j \geq 0\right\} + 1\\
&= \max\left\{\max(1, \max\left\{1 : i \geq k > j\right\} + 1) + 1 : i \geq j \geq 0\right\} + 1\\
&= 4.
\end{aligned}$$

□

Lemme 1. *Soit $G = (V, E)$ un graphe dirigé. S'il existe un chemin de u vers v dans G, alors il existe un chemin de u vers v de longueur au plus $2^{\lceil \log |V| \rceil}$.*

Démonstration. Soit $u \xrightarrow{e_1 e_2 \cdots e_k} v$ un chemin. Si $k \leq |V|$, alors nous avons fini car $|V| = 2^{\log |V|} \leq 2^{\lceil \log |V| \rceil}$.

Autrement, par le principe du pigeonnier, au moins un sommet w est répété sur le chemin. Plus précisément, il existe $i < j$ tels que

$$u \xrightarrow{e_1 \cdots e_i} w \xrightarrow{e_{i+1} \cdots e_j} w \xrightarrow{e_{j+1} \cdots e_k} v.$$

Ainsi, nous pouvons raccourcir le chemin en retirant le cycle: $u \xrightarrow{e_1 \cdots e_i\, e_{j+1} \cdots e_k} v$. En répétant ce processus, on obtient un chemin sans répétition et ainsi de longueur au plus $|V|$. □

Lemme 3. *Soit φ une formule en 2-FNC satisfaite par une assignation* val$\colon X \cup \overline{X} \to \{\texttt{faux}, \texttt{vrai}\}$. *Si $\ell \to^* \ell'$ dans G_φ et* val$(\ell) = \texttt{vrai}$*, alors* val$(\ell') = \texttt{vrai}$.

Démonstration. Montrons que $\ell \to^n \ell'$ et val$(\ell) = \texttt{vrai}$ implique val$(\ell') = \texttt{vrai}$ par induction sur n. Si $\ell \to^0 \ell'$, alors $\ell = \ell'$ et ainsi val(ℓ') = val(ℓ). Soit $n > 0$. Supposons que l'hypothèse d'induction soit satisfaite pour $n-1$. Puisque $\ell \to^n \ell'$, il existe un littéral g tel que $\ell \to^{n-1} g \to \ell'$. Comme val$(\ell) = \texttt{vrai}$, par hypothèse d'induction, nous avons val$(g) = \texttt{vrai}$. De plus, comme $(g, \ell') \in E$, la formule φ contient la clause $\neg g \vee \ell'$. Puisque val$(g) = \texttt{vrai}$ et comme φ est satisfaite, nous avons forcément val$(\ell') = \texttt{vrai}$. □

Lemme 4. *Soit φ une formule en 2-FNC. Tous les littéraux ℓ et ℓ' de G_φ satisfont $\ell \to^* \ell'$ ssi $\neg\ell' \to^* \neg\ell$.*

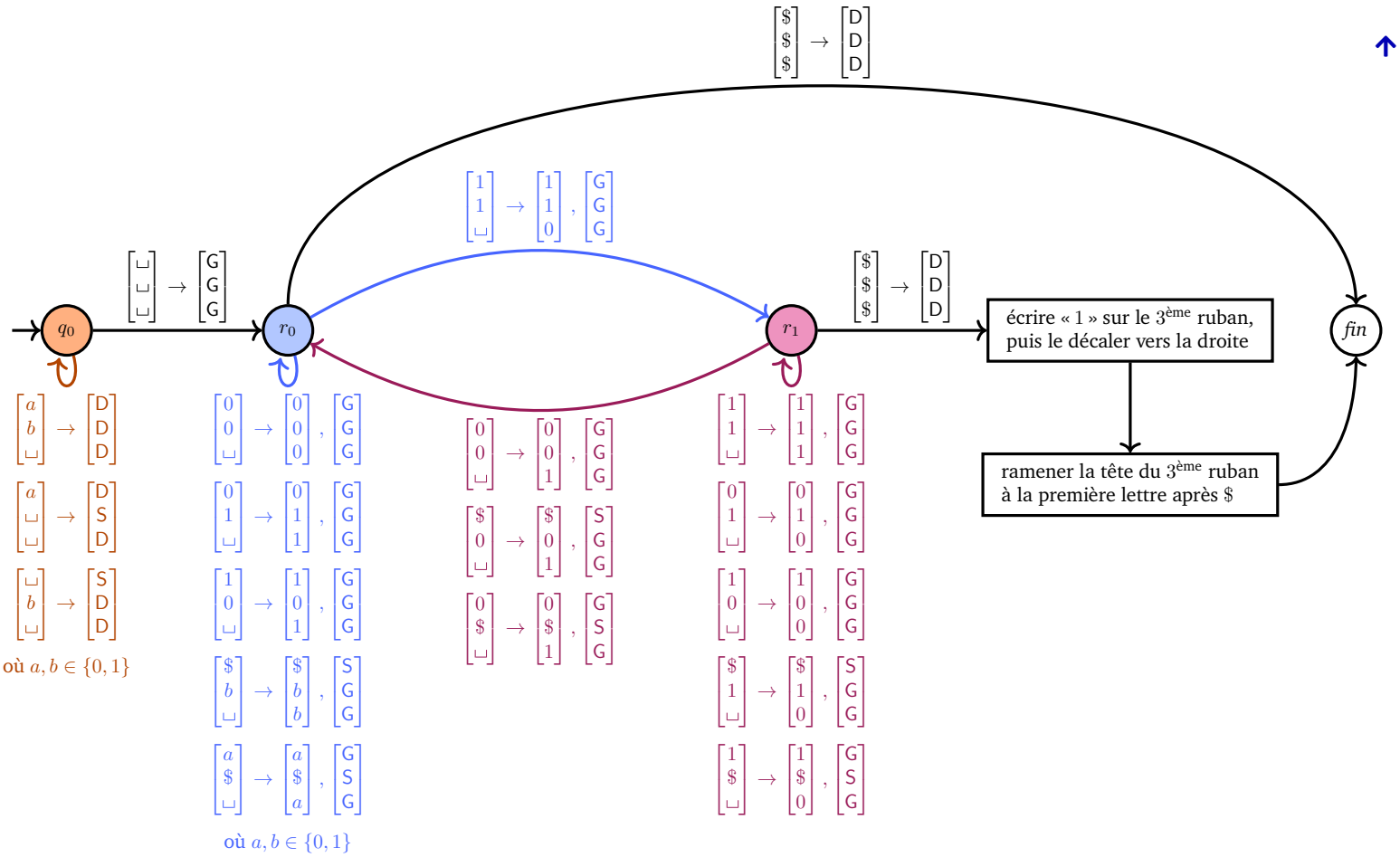
Démonstration. Montrons $\ell \to^n \ell'$ ssi $\ell' \to^n \ell$ par induction sur n. Pour $n = 0$, $\ell \to^0 \ell'$ ssi $\ell' = \ell$ ssi $\ell' \to^0 \ell$. Soit $n > 0$. Supposons que l'hypothèse d'induction est satisfaite pour $n - 1$. Nous avons:

$$
\begin{aligned}
& \ell \to^n \ell' \\
\iff & \exists g \in X \cup \overline{X} : \ell \to^{n-1} g \text{ et } g \to \ell' \\
\iff & \exists g \in X \cup \overline{X} : \neg g \to^{n-1} \neg \ell \text{ et } g \to \ell' \qquad \text{(par hypothèse d'induction)} \\
\iff & \exists g \in X \cup \overline{X} : \neg g \to^{n-1} \neg \ell \text{ et } \neg \ell' \to \neg g \ ((g, \ell') \in E) \text{ ssi } (\neg \ell', \neg g) \in E) \\
\iff & \exists g \in X \cup \overline{X} : \neg \ell' \to \neg g \text{ et } \neg g \to^{n-1} \neg \ell \\
\iff & \ell' \to^n \ell.
\end{aligned}
$$

□

Pré-condition: les trois têtes sont à droite de \$, et les trois rubans sont de la forme $\begin{bmatrix} \$x\sqcup\cdots \\ \$y\sqcup\cdots \\ \$\sqcup\cdots \end{bmatrix}$ où $x, y \in \{0,1\}^+$

Post-condition: les trois têtes sont à gauche, les deux premiers rubans n'ont pas changé, et le troisième ruban contient la somme de x et y en considérant leurs bits de poids faible comme étant à droite



Bibliographie

[Haa18] Christoph Haase : A survival guide to Presburger arithmetic. *ACM SIGLOG News*, 5(3):67–82, 2018.

[Sip06] Michael Sipser : *Introduction to the theory of computation*. Thomson Course Technology, 2ème édition, 2006.