# Automates cellulaires

Mémoire de 1$^{\text{ème}}$ année

Magistère de Mathématiques

UFR de Mathématique et d'Informatique
Université de Strasbourg

Thomas Saigre

sous la direction de Michel Coornaert

26 août 2018

# Table des matières

**1 Introduction** **3**

**2 Configurations et décalage** **4**
2.1 Définitions préliminaires . . . . 4
2.2 Action de décalage . . . . 4
2.3 Motif . . . . 4

**3 Automates cellulaires** **5**
3.1 Définition générale . . . . 5
3.2 Propriétés des automates cellulaires . . . . 6
3.3 Mémoire minimale d'un automate cellulaire . . . . 7
3.4 Automates cellulaires unidimensionnels . . . . 8
3.5 Configurations périodiques . . . . 10
3.6 Configurations finies . . . . 10

**4 Topologie sur l'ensemble des configurations** **11**
4.1 Topologie prodiscrète . . . . 11
4.2 Automate cellulaire réversible . . . . 12
4.3 Autre propriété topologique . . . . 12
4.4 Injectivité et surjectivité . . . . 12
4.5 Théorème du Jardin d'Éden . . . . 14
4.6 Cas unidimensionnel . . . . 16

**5 Graphes de de Bruijn** **18**
5.1 Graphe de de Bruijn . . . . 18
5.2 Graphe pair . . . . 20

**6 Test algorithmique** **22**
6.1 Graphe réduit . . . . 22
6.2 Parcours en profondeur . . . . 22
6.3 Résultats . . . . 25

**A Code OCaml** **26**
A.1 Fonctions auxiliaires . . . . 26
A.2 Graphes de de Bruijn . . . . 26
A.2.1 Graphe de de Bruijn . . . . 27
A.2.2 Représentation de de Bruijn d'un automate cellulaire . . . . 28
A.2.3 Graphe pair . . . . 28
A.2.4 Réduire le graphe pair . . . . 29
A.3 Injectivité et surjectivité . . . . 30

**B Langage des orphelins** **34**
B.1 Généralités sur les automates finis . . . . 34
B.2 AFD, AFDC et méthode des sous-ensembles . . . . 34
B.3 Retour aux automates cellulaires . . . . 35

**Références** **37**

# 1 Introduction

Les automates cellulaires ont été inventés dans les années 1940 par les mathématiciens Stanislaw Ulam et John von Neumann. Ulam étudiait la croissance des cristaux en la modélisant sur une grille. De son côté, von Neumann travaillait sur des systèmes auto-réplicatifs et avait des problèmes pour modéliser un robot capable de se copier lui-même. En s'inspirant des travaux de son collègue, von Neumann conçut le tout premier automate cellulaire, le « copieur et constructeur universel ». Dans les décennies qui ont suivies, la théorie des automates cellulaires s'est beaucoup développée et de nouveaux automates ont été inventés. Le plus connu est le Jeu de la Vie, imaginé par John Conway en 1970.

Les automates cellulaires permettent de simuler formellement des systèmes complexes. On démarre avec des cellules qui sont dans un état appartenant à un certain ensemble. L'état d'une cellule est changé par l'application d'une règle locale qui prend en compte l'état de la cellule ainsi que celui des cellules dans un voisinage fixe. On applique ce procédé à chaque cellule de manière parallèle. Le principe de fonctionnement des automates cellulaires est simple à comprendre, mais le comportement de l'état des cellules peut être beaucoup plus compliqué à étudier.

Une étude théorique des automates cellulaires fera l'objet d'une première partie. Ensuite, nous nous intéresserons à une étude topologique de l'ensemble des configurations de cellules. Ces parties utilisent principalement le livre de Tullio Ceccherini-Silberstein et Michel Coornaert [1] ainsi que le cours de Jarkko Kari [2], à propos de la théorie des automates cellulaires. Un théorème important, le théorème du Jardin d'Éden, sera démontré. Ce théorème est l'un des plus anciens résultats de la théorie des automates cellulaires et a été démontré pour la première fois dans les années 1960 par Moore et Myhill. Le théorème du Jardin d'Éden permettra de mettre en place un algorithme qui détermine la surjectivité et l'injectivité d'une certaine classe d'automates cellulaires : les automates unidimensionnels. Un premier algorithme avait été proposé par Amoroso et Patt en 1972 [3], montrant ainsi que la question de réversibilité pour les automates unidimensionnels était décidable, mais la procédure était assez coûteuse d'un point de vue informatique. L'algorithme que nous allons étudier est basé sur les graphes de de Bruijn et a été proposé par Sutner [4]. Nous nous intéresserons enfin à une implémentation de cet algorithme. Cette implémentation utilisera un parcours de graphe en profondeur. Grâce à cela, nous pourrons compter parmi certaines classes d'automates ceux qui vérifient les propriétés de surjectivité et d'injectivité.

# 2 Configurations et décalage

Pour définir de façon mathématique ce qu'est un automate cellulaire, nous allons avoir besoin d'introduire certaines notions. Cette section définit celles-ci et introduit l'action de décalage.

## 2.1 Définitions préliminaires

Pour définir ce qu'est un automate cellulaire, nous allons utiliser un groupe $G$ quelconque, qui sera appelé l'*univers*, et un ensemble quelconque $A$, *l'alphabet*. Les éléments de $A$ sont appelés les *états*. Ainsi, un élément $g$ de l'univers correspond à une cellule de l'automate et l'alphabet correspond au différents états que la cellule peut atteindre. L'ensemble $A^G$ des applications de $G$ dans $A$ est l'ensemble des *configurations* de l'automate. Ainsi, une configuration $x \in A^G$ est une application $x \colon G \to A$ qui associe à une cellule $g \in G$ son état dans $A$, on peut donc voir une configuration comme une photographie des états de toutes les cellules à un moment donné.

Dans la pratique, on prendra souvent $G = \mathbb{Z}^d$ (avec plus particulièrement $d = 1$ ou 2) et $A$ fini, mais ces définitions restent valables pour $G$ et $A$ quelconques.

## 2.2 Action de décalage

**Définition 2.1** (Opération de décalage)**.** Soit l'opération suivante :

$$\begin{array}{rcl} G \times A^G & \to & A^G \\ (g, x) & \mapsto & gx, \end{array}$$

où $gx$ est l'application de $G$ dans $A$ qui à $h \in G$ associe $x(g^{-1}h)$, pour $g \in G$ et $x \in A^G$. Cette opération est appelée opération de *décalage* sur $A^G$.

**Proposition 2.2.** *L'opération de décalage est une action à gauche de $G$ sur $A^G$. On l'appelle donc* action de décalage *sur $A^G$.*

***Preuve.*** Pour $g \in G$, on note $L_g \colon G \to G$ la multiplication à gauche par $g$ : $\forall h \in G, L_g(h) = gh$. Alors, pour $g \in G$ et $x \in A^G$, on a $gx = x \circ L_{g^{-1}}$. On a par ailleurs cette propriété immédiate :

$$\forall g_1, g_2 \in G, L_{g_1 g_2} = L_{g_1} \circ L_{g_2}$$

Vérifions les axiomes des actions pour cette opération :

- On note $e$ le neutre de $G$. Soit $x \in A^G$. On a alors $ex = x \circ L_{e^{-1}} = x \circ L_e = x$ car $L_e = \mathrm{Id}_G$.
- Soient $g_1, g_2 \in G$ et $x \in A^G$. Alors $(g_1 g_2)x = x \circ L_{(g_1 g_2)^{-1}} = x \circ L_{g_2^{-1} g_1^{-1}} = x \circ L_{g_2^{-1}} \circ L_{g_1^{-1}} = g_1(g_2 x)$ (par associativité de la composition).

■

Cette action va décaler la configuration sur le groupe $G$. Par exemple, en prenant $G = \mathbb{Z}$, pour $x \in A^{\mathbb{Z}}$ la configuration $2x$ est une application de $\mathbb{Z}$ dans $A$ qui à $g \in \mathbb{Z}$ associe $x(g - 2)$ (avec la notation additive). Cette action permet donc de « recentrer » la configuration autour d'une cellule bien précise.

## 2.3 Motif

On définit un *motif* sur le groupe $G$ et l'alphabet $A$ comme étant une application $p \colon D \to A$, où $D$ est un sous-ensemble de $G$, appelé le *support* de $p$. Si $D$ est fini, on dit que $p$ est un *motif fini*. On remarque qu'une configuration $x \in A^G$ est un motif de domaine $D = G$.

Si $g \in G$ et $p \colon D \to A$ est un motif, on définit le motif $p' = gp \colon D' \to A$ en prenant $D' = gD$ et $gp'(h) = p(g^{-1}h)$ pour tout $h \in D'$. On dit alors que $p$ et $p'$ sont des *copies translatées* l'une de l'autre.

Enfin, si $p_1 \colon D_1 \to A$ et $p_2 \colon D_2 \to A$ sont deux motifs, on dit que $p_1$ est un *sous-motif de* $p_2$ si $D_1 \subset D_2$ et si $\forall g \in D_1, p_1(g) = p_2(g)$. On dit par ailleurs que $p_1$ et $p_2$ sont *disjoints* si $D_1 \cap D_2 = \varnothing$.

# 3 Automates cellulaires

## 3.1 Définition générale

Soient $G$ un groupe et $A$ un alphabet.

**Définition 3.1.** On définit un automate cellulaire sur le groupe $G$ et l'alphabet $A$ comme étant une application $\tau \colon A^G \to A^G$ telle qu'il existe un sous-ensemble $M \subset G$ fini et une application $\mu \colon A^M \to A$ tels que $\forall x \in A^G, \forall g \in G$,

$$\tau(x)(g) = \mu\left((g^{-1}x)|_M\right),$$

où $(g^{-1}x)|_M$ est la restriction de l'application $g^{-1}x$ à $M$. L'ensemble $M$ est appelé *mémoire* et $\mu$ est la *règle locale.*

Par cette définition, on retrouve bien l'idée des automates cellulaires qui ne consiste qu'à regarder des cellules voisines à l'étape $n$ pour connaître l'état à l'étape $n+1$ : par l'action de décalage, on ne va regarder que dans la mémoire $M$, qui peut être vue comme une fenêtre ne montrant que les voisines d'une cellule, et cachant les autres qui ne servent pas pour mettre à jour son état.

Le fait que $M$ soit fini est très important : pour mettre à jour l'état d'une cellule, on va regarder les états de ses cellules voisines, mais si il y en a une infinité, un tel calcul sera impossible.

**Exemple 3.2** (Automate cellulaire associé au Jeu de la Vie)**.** Le Jeu de la Vie est un automate cellulaire imaginé par le mathématicien John Horton Conway en 1970. Prenons un plan infini de cases carrées qui représentent des cellules, dans lesquelles deux états sont possibles : vivante ou morte. Les règles d'interactions entre une cellule $c$ et ses huit voisines sont les suivantes :

— **Survie :** Une cellule vivante à la date $t$ survit à la date $t+1$ si et seulement si elle possède 2 ou 3 voisines vivantes à la date $t$ (Figure 1).

— **Mort :** Une cellule vivante à la date $t$ meurt à la date $t+1$ par isolement si et seulement si elle a au plus une voisine vivante à la date $t$ (Figure 2). Elle meurt par surpopulation si et seulement si elle a au moins quatre voisines vivantes à la date $t$ (Figure 3).

— **Naissance :** Une cellule morte à la date $t$ naît à la date $t+1$ si et seulement si exactement trois de ses voisines sont vivantes à la date $t$ (Figure 4).

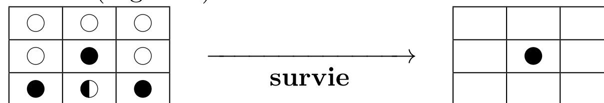



FIGURE 1 – Survie d'une cellule

FIGURE 2 – Mort par isolement d'une cellule

FIGURE 3 – Mort par surpopulation

FIGURE 4 – Naissance d'une cellule

On remarque que dans les schémas, on ne représente que les cellules voisines, car celles qui sont plus éloignées n'interviennent pas dans les règles. Cela correspond à cette fenêtre définie par la mémoire.

On prend ici :

- $G = \mathbb{Z}^2$, qui représente les cases de la grille.

- $M = \{-1, 0, 1\}^2 \subset G$ représentant les cellules voisines : si $c \in G$ est une cellule, ses voisines sont les $c + m$ pour $m \in M$.
- $A = \{0, 1\}$ correspond aux états d'une cellule : 1 si la cellule est vivante (●), 0 si elle est morte (○).

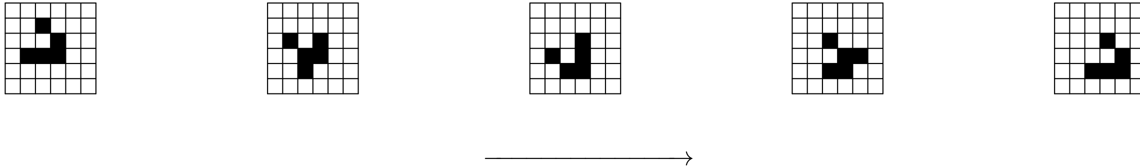
Une configuration est un élément $x \in A^G$ avec, pour toute cellule $c \in G$ : $x(c) = 1$ si $c$ est vivante et $x(c) = 0$ si $c$ est morte.

On pose $\mu \colon A^M \to A$ définie par :

$$\mu(y) = \begin{cases} 1 \text{ si} & \left| \begin{array}{l} \sum_{m \in M} y(m) = 3 \\ \text{ou} \\ \sum_{m \in M} y(m) = 4 \text{ et } y((0,0)) = 1 \end{array} \right. \\ 0 \text{ sinon} \end{cases}$$

pour $y \in A^M$. Alors il vient que l'application $\mu$ exprime bien les lois du Jeu de la Vie définies ci-dessus. L'automate cellulaire $\tau \colon A^G \to A^G$, avec $M$ comme mémoire et $\mu$ comme règle locale est l'automate cellulaire associé au Jeu de la Vie.

Pour un automate cellulaire $\tau$ donné, on va donc itérer plusieurs fois cette application pour toutes les cellules. Par exemple, à partir de la configuration suivante, pour le Jeu de la Vie, plusieurs itérations de l'automate cellulaire vont donner :



$\longrightarrow$

Pour un univers $G$ et un alphabet $A$ donnés, on note $\mathrm{CA}(G, A)$ l'ensemble des automates cellulaires sur le groupe $G$ avec l'alphabet $A$, c'est-à-dire l'ensemble des applications $\tau \colon A^G \to A^G$ vérifiant la définition 3.1.

## 3.2 Propriétés des automates cellulaires

**Définition 3.3.** Soit $E$ un ensemble muni d'une action d'un groupe $G$. Soit $f \colon E \to E$. On dit que $f$ est *$G$-équivariante* si $\forall g \in G, \forall x \in E, f(g \cdot x) = g \cdot f(x)$.

**Proposition 3.4** (Caractérisation des automates cellulaires)**.** *Soient $G$ un groupe et $A$ un ensemble quelconque. Soient $M$ un sous-ensemble fini de $G$ et $\mu$ une application de $A^M$ dans $A$. Les conditions suivantes sont équivalentes :*

1. *$\tau \in \mathrm{CA}(G, A)$ avec $M$ comme mémoire et $\mu$ comme règle locale.*
2. *L'application $\tau$ est $G$-équivariante et $\forall x \in A^G, \tau(x)(e_G) = \mu(x|_M)$ (ici, $e_G$ désigne le neutre de $G$).*

***Preuve.*** Supposons 1. Soient $x \in A^G, g, h \in G$. Alors :

$$\tau(gx)(h) = \mu((h^{-1}gx)|_M) = \mu(((g^{-1}h)^{-1}x)|_M) = \tau(x)(g^{-1}h) = g\tau(x)(h)$$

Donc $\tau(gx) = g\tau(x)$. D'après la formule de la définition 3.1, en prenant $g = e_G$, on a :

$$\forall x \in A^G, \tau(x)(e_G) = \mu((e_G^{-1}x)|_M) = \mu(x|_M)$$

Supposons maintenant 2. Soient $x \in A^G$ et $g \in G$. Par définition de l'action de décalage, on a $x(g) = g^{-1}x(e_G)$, donc en utilisant la $G$-équivariance de $\tau$, il vient que :

$$\tau(x)(g) = \tau(g^{-1}x)(e_G) = \mu((g^{-1}x)|_M)$$

$\tau$ vérifie bien l'axiome de la définition d'automate cellulaire, $\tau$ vérifie donc 1. ■

On remarque que : dire que $\tau$ est $G$-équivariante signifie qu'elle commute avec l'action de décalage.

**Lemme 3.5.** *Soient $\tau_1, \tau_2 \in \mathrm{CA}(G, A)$. Alors $\tau_1 \circ \tau_2 \in \mathrm{CA}(G, A)$, c'est-à-dire $\tau_1 \circ \tau_2$ est un automate cellulaire sur $G$ avec l'alphabet $A$. De plus si $M_1$, $M_2$ sont des mémoires de $\tau_1$ et $\tau_2$ respectivement, alors $M_1M_2 := \{m_1m_2 \mid m_1 \in M_1, m_2 \in M_2\}$ est une mémoire de $\tau_1 \circ \tau_2$.*

***Preuve.*** D'après la proposition précédente, $\tau_1$ et $\tau_2$ sont $G$-équivariants. Soient $g \in G$ et $x \in A^G$, alors :

$$\tau_1 \circ \tau_2(gx) = \tau_1(g^{-1}\tau_2(x)) = g(\tau_1 \circ \tau_2)(x)$$

Donc $\tau_1 \circ \tau_2$ est $G$-équivariant.

De plus, $\tau_1$ et $\tau_2$ vérifient $\forall x \in A^G, \tau_i(x)(e_G) = \mu(x|_{M_i})$. La relation de la définition d'automate cellulaire nous dit que $\forall g \in G, \forall x \in A^G, \tau_i(g)(x)$ ne dépend que de la restriction de $x$ sur $gM_i$ car $\forall m \in M_i, x(g^{-1}x)(m) = x(gm)$. Donc $\tau_1(\tau_2(x))(e_G)$ ne dépend que de la restriction de $\tau_2(x)$ sur $M_1$, et en raisonnant de même, $\forall m \in M_1$, l'élément $\tau_2(x)(m)$ ne dépend que de la restriction de $x$ sur $mM_2$. Ainsi, $\tau_1 \circ \tau_2(x)(e_G)$ ne dépend que de la restriction de $x$ sur $M_1M_2$.

D'après la proposition précédente, il vient que $\tau_1 \circ \tau_2 \in \mathrm{CA}(G, A)$. ■

Ce lemme montre que la composition définit une loi de composition interne sur $\mathrm{CA}(G, A)$.

**Proposition 3.6.** *L'ensemble* $\mathrm{CA}(G, A)$ *est un monoïde pour la composition.*

***Preuve.*** La composition étant associative, il ne reste qu'à montrer l'existence d'un automate cellulaire $\mathrm{e}_{\mathrm{CA}(G,A)}$ neutre (c'est-à-dire tel que $\forall \tau \in \mathrm{CA}(G, A), \tau \circ \mathrm{e}_{\mathrm{CA}(G,A)} = \mathrm{e}_{\mathrm{CA}(G,A)} \circ \tau = \tau$). On pose $\mathrm{e}_{\mathrm{CA}(G,A)} = \mathrm{Id}_{A^G}$ (l'application identité sur $A^G$). Alors en posant $M := \{e_G\}$ et $\mu \colon A^M \to A$ définie par $\mu(y) := y(e_G)$, on voit bien que $\mathrm{Id}_{A^G} \in \mathrm{CA}(G, A)$. On a par ailleurs $\forall \tau \in (G, A), \tau \circ \mathrm{e}_{\mathrm{CA}(G,A)} = \mathrm{e}_{\mathrm{CA}(G,A)} \circ \tau = \tau$. ■

## 3.3 Mémoire minimale d'un automate cellulaire

Soit $\tau \in \mathrm{CA}(G, A)$, et notons $M$ une mémoire de $\tau$ et $\mu$ la règle locale associée à $M$. On remarque que si $M'$ est un ensemble fini tel que $M \subset M'$, alors $M'$ est aussi une mémoire associée à $\tau$, avec comme règle locale $\mu' \colon A^{M'} \to A$ où $\mu' = \mu \circ p$ où $p$ est la projection canonique de $A^{M'}$ vers $A^M$. Cela montre que la mémoire d'un automate cellulaire n'est pas unique. Plus précisément, on a :

**Lemme 3.7.** *Soit* $\tau \in \mathrm{CA}(G, A)$. *Si* $M_1$ *et* $M_2$ *sont des mémoires de* $\tau$, *alors* $M_1 \cap M_2$ *est aussi une mémoire de* $\tau$.

***Preuve.*** Soit $x \in A^G$. Montrons que $\tau(x)(e_G)$ ne dépend que de la restriction de $x$ à $M_1 \cap M_2$. Prenons une configuration $y \in A^G$ qui coïncide avec $x$ sur $M_1 \cap M_2$ : $x|_{M_1 \cap M_2} = y|_{M_1 \cap M_2}$. Soit maintenant une configuration $z$ qui coïncide avec $x$ sur $M_1$ et $y$ sur $M_2$ : $z|_{M_1} = x|_{M_1}$ et $z|_{M_2} = y|_{M_2}$ (on peut par exemple poser $z$ comme étant égale à $x$ sur $M_1$ et égale à $y$ sur $G \setminus M_1$). D'une part, comme $M_1$ est une mémoire de $\tau$, on a $\tau(x)(e_G) = \tau(z)(e_G)$ car $z$ et $x$ coïncident sur $M_1$. D'autre part, comme $M_2$ est aussi une mémoire, on a $\tau(y)(e_G) = \tau(z)(e_G)$, car $y$ et $z$ coïncident sur $M_2$. Il s'ensuit que $\tau(x)(e_G) = \tau(y)(e_G)$.

Ainsi, il existe une application $\mu \colon A^{M_1 \cap M_2} \to A$ telle que $\forall x \in A^G, \tau(x)(e_G) = \mu(x|_{M_1 \cap M_2})$. De plus, d'après la Proposition 3.4 (le sens direct), il vient que $\tau$ est $G$-équivariant, donc en appliquant à nouveau cette Proposition 3.4 (en utilisant le sens réciproque), il vient que $M_1 \cap M_2$ est une mémoire de $\tau$. ■

On peut par ailleurs établir ces deux résultats :

**Proposition 3.8.** *Soit* $\tau \in \mathrm{CA}(G, A)$. *Il existe une unique mémoire* $M_0 \subset G$ *de cardinal minimal. Plus précisément, si* $M$ *est un sous-ensemble fini de* $G$, *alors* $M$ *est une mémoire de* $\tau$ *si et seulement si* $M_0 \subset M$.

***Preuve.*** Soit $M_0$ une mémoire de cardinal minimal. On a vu précédemment que tout sous-ensemble de $G$ contenant $M_0$ est une mémoire de $\tau$. Inversement, si $M$ est une mémoire de $\tau$, d'après le lemme précédent, il vient que $M \cap M_0$ est une mémoire de $\tau$. On a de plus, $|M \cap M_0| \leqslant |M_0|$. Comme $M_0$ est une mémoire de cardinal minimal, il vient que $M \cap M_0 = M_0$, et alors $M \subset M_0$. En particulier, $M_0$ est l'unique mémoire de cardinal minimal. ■

**Proposition 3.9.** *Soit* $\tau \in \mathrm{CA}(G, A)$, *et soit* $M$ *une mémoire associée à* $\tau$. *Alors la règle locale* $\mu \colon A^M \to A$ *associée à l'automate cellulaire* $\tau$ *est unique.*

***Preuve.*** Soient $\mu_1 \colon A^M \to A$ et $\mu_2 \colon A^M \to A$ deux règles locales de $\tau$, on a alors d'après la définition 3.1 $\forall x \in A^G, \forall g \in G$ :

$$\tau(x)(g) = \mu_1(g^{-1}x|_M) = \mu_2(g^{-1}x|_M)$$

Soit $x \in A^M$, alors, en prenant $g = e_G$ dans l'égalité précédente, il vient que $\mu_1(x|_M) = \mu_2(x|_M)$ et comme $x$ est définie de $M$ dans $A$, il en résulte $\mu_1(x) = \mu_2(x)$. Donc $\mu_1 = \mu_2$.

■

**Lemme 3.10.** *Soit* $\tau \in \mathrm{CA}(G, A)$ *avec* $M$ *comme mémoire et soit* $g \in G$. *Alors* $\tau(x)(g)$ *ne dépend que de la restriction de* $x$ *à* $gM := \{gm \mid m \in M\}$.

***Preuve.*** Pour $m \in M$, on a $(g^{-1}x)(s) = xg(s)$, par définition de l'action de décalage. Or par définition des automates cellulaires, $\tau(x)(g) = \mu((g^{-1}x)|_M)$. D'où le résultat. ■

Prenons ici $G = \mathbb{Z}^d$. Pour un automate $\tau \in \mathrm{CA}(\mathbb{Z}^d, A)$, et pour $r \in \mathbb{N}^*$, la mémoire $M$ peut être donc complétée de telle sorte que l'ensemble

$$M_r^d := \left\{(k_1, \cdots, k_d) \in \mathbb{Z}^d \,\middle|\, |k_i| \leqslant r\right\}$$

soit une mémoire de cet automate. Cette mémoire contient $(2r+1)^d$ éléments et l'automate $\tau$ muni de cette mémoire est appelé *automate cellulaire de rayon $r$*.

## 3.4 Automates cellulaires unidimensionnels

Nous allons ici nous intéresser à un type particulier d'automates cellulaires : les automates cellulaires unidimensionnels. On prend ici $G = \mathbb{Z}$, les cellules peuvent donc être représentées sur une ligne infinie. La définition d'un automate unidimensionnel reste donc la même qu'un automate cellulaire quelconque, avec la seule contrainte $G = \mathbb{Z}$.

On va alors se fixer un entier $m > 0$ qui va correspondre à la taille de la mémoire utilisée dans la définition de l'automate. On voit ainsi qu'à $m$ fixé, si $A$ est fini, le nombre de règles locales est fini : il en existe très exactement $a^{a^m}$, où $a$ désigne le nombre de lettres de l'alphabet. En effet, sur les $m$ cases voisines, il y a $a^m$ possibilités de configurations de ces cases, et à chacune de ces configurations on peut associer $a$ états sortants possibles.

**Exemple 3.11** (Automates élémentaires)**.** On prend ici $M = \{-1, 0, 1\}$ comme mémoire. On a alors $m = 3$. Dans cet automate unidimensionnel, les cellules sont disposées sur une ligne et peuvent être vivantes ou mortes. On pose alors $A = \{0, 1\}$. On voit donc qu'il y a $2^{2^3} = 256$ règles possibles. Ces règles ont été étudiées dans les années 1980[5] par Steven Wolfram, qui leur a donné une numérotation : le code de Wolfram. À chacune des 256 règles locales, on va associer un entier compris ente 0 et 255 qui est obtenu par la méthode suivante : on écrit à la suite les résultats des transformations par la règle locale des 8 configurations (111, 110, . . . , 000) que l'on interprète comme un nombre en base 2. En convertissant ce nombre en base 10, on obtient le numéro de la règle. Prenons par exemple l'automate élémentaire défini par cette règle locale :

| $x(M)$ | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|---|
| $\mu(x)$ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Cela nous donne, en mettant les résultats bout à bout le nombre 01111110 en base 2, qui correspond en base 10 au nombre 126. Cet automate élémentaire correspond donc à la règle 126 du code de Wolfram.

Cette règle locale est en fait définie ainsi : pour une cellule $n \in \mathbb{Z}$, si $n$ est morte et ses deux voisines aussi, alors $n$ reste morte et si $n$ est vivante et ses deux voisines aussi, alors $n$ meurt. Dans le cas contraire ($n$ et ses deux voisines ne sont pas dans le même état) si $n$ est morte, alors elle naît, et si elle est vivante, alors elle survit. La Figure 5 schématise le fonctionnement de cette règle locale.

Figure 5 – Règle 126

Pour l'exemple, on va partir d'une configuration où une seule cellule est vivante et appliquer l'automate un certain nombre de fois. Comme les cellules peuvent êtres représentées sur une même ligne, on va représenter la configuration suivante en plaçant une nouvelle ligne en dessous de la première, et ainsi de suite. On obtient alors un diagramme espace-temps comme celui représenté Figure 6 pour la règle 126.

Avec d'autres règles, on peut avoir d'autres figures dont certaines peuvent paraître totalement aléatoires, comme c'est le cas, par exemple pour la règle 110. On a $\overline{110}^{10} = \overline{01101110}^{2}$, d'où la table de la règle locale :

| $x(M)$ | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|---|
| $\mu_{110}(x)$ | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

Le diagramme espace-temps est représenté Figure 7 à partir d'une configuration aléatoire.

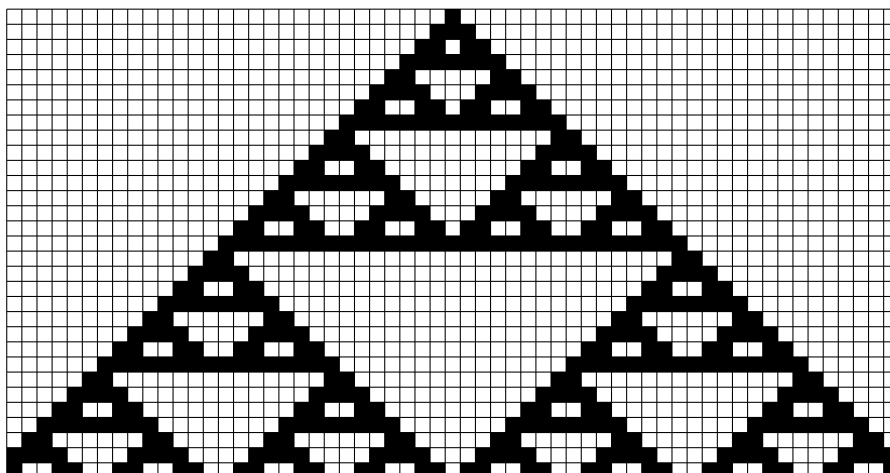


Figure 6 – La diagramme espace-temps pour la règle 126. On voit ici apparaître le triangle de Sierpiński

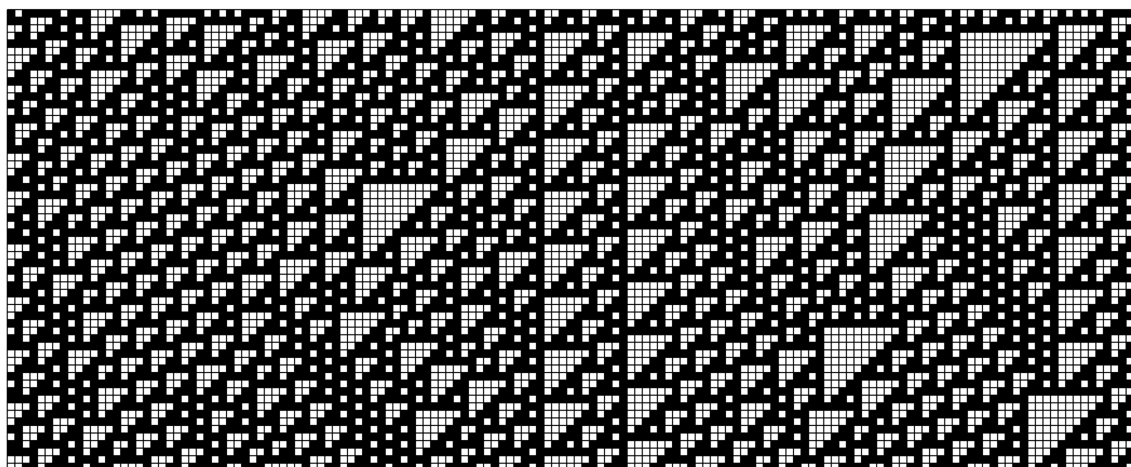


Figure 7 – Diagramme espace-temps pour la règle 110, à partir d'une configuration aléatoire

## 3.5 Configurations périodiques

**Définition 3.12** (Configuration périodique spatiale)**.** Soit $x \in A^G$ une configuration, et soit $r \in G, r \neq e_G$. On dit que $x$ est *$r$-périodique* si $\forall g \in G, x(g) = x(rg)$ (on note multiplicativement l'opération du groupe $G$).

En posant $t_r\colon A^G \to A^G$ le décalage de $r$, où $t_r(x) = r^{-1}x$, une configuration $x$ est $r$-périodique si et seulement si $x = t_r(x)$. On dit que $x$ est *périodique* si il existe $r \in G \setminus \{e_G\}$ tel que $x$ soit $r$-périodique. On pose alors $\mathcal{P} := \{\text{configurations périodiques}\}$.

**Proposition 3.13.** *Soit $\tau \in \mathrm{CA}(G, A)$, et soit $x \in A^G$, $r$-périodique. Alors $\tau(x)$ est aussi $r$-périodique.*

***Preuve.*** La configuration $x$ est $r$-périodique, donc $x = r^{-1}x$. D'après la Proposition 3.4, $\tau$ est $G$-équivariant, donc $\tau(x) = \tau(r^{-1}x) = r^{-1}\tau(x)$. La configuration $\tau(x)$ est donc $r$-périodique. ■

**Corollaire 3.14.** *Soit $\tau \in \mathrm{CA}(G, A)$, alors $\mathcal{P}$ est stable par $\tau$.*

Pour un automate cellulaire $\tau$, on peut donc considérer sa restriction aux configurations périodiques : $\tau_P\colon \mathcal{P} \to \mathcal{P}$, qui est bien définie.

## 3.6 Configurations finies

**Définition 3.15** (Support d'une configuration)**.** Soit $x \in A^G$ et soit $a \in A$. Le *$a$-support* de $x$ est l'ensemble

$$\mathrm{supp}_a(x) := \{g \in G \mid x(g) \neq a\},$$

c'est l'ensemble des cellules qui ne sont *pas* dans l'état $a$.

On dit qu'une configuration $x \in A^G$ est *$a$-finie* (pour un état $a \in A$) si $\mathrm{supp}_a(x)$ est fini, c'est-à-dire si toutes les cellules sauf un nombre fini sont dans l'état $a$. On pose $\mathcal{F}_a := \{x \in A^G \mid x \text{ est } a\text{-finie}\}$.

**Proposition 3.16.** *Soit $\tau \in \mathrm{CA}(G, A)$ et notons $\mu$ la règle locale associée à une mémoire $M$. Soit $a \in A$. Soit $x \in \mathcal{F}_a$ une configuration $a$-finie. Alors $\tau(x)$ est une configuration $t$-finie, où $t = \mu(u_a)$ (où $u_a \in A^M$ est un motif, dit* tranquille*, tel que toutes les cellules de $u_a$ soient dans l'état $a$).*

***Preuve.*** On pose $\Omega = \mathrm{supp}_a(x)$, qui est fini. Si $g \in G$ est tel que $gM \cap \Omega = \varnothing$, c'est-à-dire tel que le voisinage de $g$ ne rencontre pas $\Omega$, alors $\forall h \in gM, x(h) = a$ et donc $\tau(x)(g) = t$. Comme $\Omega$ est fini, il n'existe qu'un nombre fini de $g \in G$ tel que $gM \cap \Omega \neq \varnothing$. Ainsi, $\mathrm{supp}_t(\tau(x))$ est fini : $\tau(x)$ est $t$-finie. ■

On dit qu'une configuration $x \in A^G$ est *presque constante* si il existe $a \in A$ tel que $x$ soit $a$-finie. On définit alors l'ensemble $\mathcal{F}$ des configurations presque constantes. On a $\mathcal{F} = \bigcup_{a \in A} \mathcal{F}_a$. D'après la proposition précédente, $\mathcal{F}$ est stable par un automate cellulaire $\tau \in \mathrm{CA}(G, A)$. On peut donc définir sa restriction aux configurations presque constantes : $\tau_F\colon \mathcal{F} \to \mathcal{F}$.

# 4 Topologie sur l'ensemble des configurations

## 4.1 Topologie prodiscrète

**Définition 4.1** (Topologie discrète)**.** Pour un ensemble $X$ quelconque, l'ensemble des parties de $X$, noté $\mathcal{P}(X)$, définit une topologie sur $X$. Les parties de $X$ en forment les ouverts. Cette topologie est appelée *topologie discrète.* La topologie discrète est induite par la *distance discrète* : pour $x, y \in X, d(x,y) = \begin{cases} 0 \text{ si } x = y \\ 1 \text{ sinon} \end{cases}$.

Une configuration $x \in A^G$ peut être vue comme une famille indexée par $G$ : $x = (x(g))_{g\in G}$. Ainsi, on peut voir l'ensemble $A^G$ comme un produit :

$$A^G = \prod_{g\in G} A$$

En munissant chaque facteur $A$ de la topologie discrète et en considérant le produit de ces topologies, on munit $A^G$ d'une topologie, la *topologie prodiscrète.*

**Proposition 4.2.** *Soit $\tau \in \mathrm{CA}(G, A)$ ($A$ est un ensemble quelconque). Alors $\tau$ est une application continue.*

Pour la preuve de ce théorème, et certaines preuves qui vont suivre, on définit le voisinage d'une configuration $x \in A^G$ à partir d'un sous-ensemble fini $\Omega \subset G$ par :

$$V(x,\Omega) := \left\{ y \in A^G \,\middle|\, x|_\Omega = y|_\Omega \right\} \tag{1}$$

L'ensemble $V(x, \Omega)$ est l'ensemble des configurations $y$ qui coïncident avec $x$ sur la partie $\Omega$. Si $\Omega$ décrit l'ensemble des sous-ensembles finis de $G$, les $V(x, \Omega)$ forment une base de voisinage de $x$.

***Preuve.*** Soit $M$ la mémoire de $\tau$. Soit $x \in A^G$ et soit $W$ un voisinage de $\tau(x)$ dans $A^G$. Alors il est possible de trouver un ensemble $\Omega \subset G$, fini, tel que :

$$V(\tau(x), \Omega) \subset W$$

On pose $\Omega M := \{\omega m \,|\, \omega \in \Omega, m \in M\}$. Si $y \in A^G$ coïncide avec $x$ sur $\Omega M$, alors $\tau(x)$ et $\tau(y)$ coïncident sur $\Omega$, d'après le lemme 3.10. Ainsi, il vient que

$$\tau(V(x, \Omega M)) \subset V(\tau(x), \Omega) \subset W$$

L'application $\tau$ est continue. ■

On rappelle ici un résultat général de topologie :

**Théorème 4.3** (Tychonov)**.** *Un produit quelconque d'espaces compacts est compact.*

Quand $A$ est fini et donc compact, ce théorème donne de façon immédiate le corollaire suivant :

**Corollaire 4.4.** *Si $A$ est fini, $A^G$ muni de la topologie prodiscrète est compact.*

On avait donné dans la Proposition 3.4 une caractérisation générale des automates cellulaires. Cette caractérisation demandait de connaître à l'avance la mémoire et la règle locale. On va ici en donner une qui ne demande pas de les connaître, dans le cas où $A$ est fini.

**Théorème 4.5** (Curtis–Hedlund–Lyndon)**.** *Soit $\tau\colon A^G \to A^G$, avec $A$ fini. Les conditions suivantes sont équivalentes :*

1. *$\tau \in \mathrm{CA}(G, A)$.*
2. *L'application $\tau$ est continue pour la topologie prodiscrète et $\tau$ est $G$-équivariante (cf Définition 3.3).*

***Preuve.*** Le fait que 1. implique 2. ne dépend pas de la finitude de $A$. C'est une conséquence des propositions 4.2 et 3.4.

Supposons 2. et montrons que $\tau$ est un automate cellulaire. Soit $\varphi\colon A^G \to A; x \mapsto \tau(x)(e_G)$. Cette fonction est continue, on peut alors trouver, pour tout $x \in A^G$ un sous-ensemble fini $\Omega_x \subset G$ tel que si $y \in A^G$ coïncide avec $x$ sur $\Omega_x$, c'est-à-dire si $y \in V(x, \Omega_x)$ (cf (1)), alors $\tau(x)(e_G) = \tau(y)(e_G)$. La famille $(V(x, \Omega_x))_{x\in A^G}$ forme un recouvrement ouvert de $A^G$. D'après le corollaire précédent, $A^G$ est compact, donc par propriété de Borel–Lebesgue, il existe un sous-recouvrement fini : $\exists F \subset A^G$ fini tel que $X = \bigcup_{x\in F} V(x, \Omega_x)$. On pose $M = \bigcup_{x\in F} \Omega_x$, et soient $y, z \in A^G$ deux configurations coïncidant sur $M$. Comme la famille $(V(x, \Omega_x))_{x\in F}$ forme un recouvrement de $A^G$, il existe $x_0 \in F$ tel que $y \in V(x_0, \Omega_{x_0})$ : $y|_{\Omega_{x_0}} = x_0|_{\Omega_{x_0}}$. De plus, $M \supset \Omega_{x_0}$, on a $y|_{\Omega_{x_0}} = z|_{\Omega_{x_0}}$ et par conséquent $\tau(y)(e_G) = \tau(x)(e_G) = \tau(z)(e_G)$. On en déduit qu'il existe une application $\mu\colon A^M \to A$ telle que $\forall x \in A^G, \tau(x)(e_G) = \mu(x|_M)$. Comme $\tau$ est $G$-équivariante, il vient d'après la caractérisation 3.4 que $\tau$ est un automate cellulaire avec $M$ comme mémoire et $\mu$ comme règle locale. ■

## 4.2 Automate cellulaire réversible

**Définition 4.6** (Automate réversible)**.** Soient $G$ un groupe et $A$ un ensemble quelconque. Un automate cellulaire $\tau \in \mathrm{CA}(G, A)$ est dit *réversible* si $\tau$ est une application bijective et $\tau^{-1} \colon A^G \to A^G$ est aussi un automate cellulaire.

Si l'automate cellulaire est réversible, il est alors possible d'aller « dans les deux sens » : à partir d'une configuration, il est possible de remonter le temps en utilisant son inverse $\tau^{-1}$. Un automate cellulaire est réversible si et seulement si il existe un automate cellulaire $\sigma$ tel que $\tau \circ \sigma = \sigma \circ \tau = \mathrm{Id}_{A^G}$. Ainsi, l'ensemble des automates cellulaires réversibles est un groupe contenant tous les éléments inversibles du monoïde $\mathrm{CA}(G, A)$.

Il est possible que $\tau$ soit bijective mais que $\tau^{-1}$ ne soit pas un automate cellulaire. Le résultat suivant montre que ce n'est pas le cas si $A$ est fini.

**Théorème 4.7.** *Soient $G$ un groupe et $A$ un ensemble fini. Si $\tau \in \mathrm{CA}(G, A)$ est bijectif, alors il est réversible.*

***Preuve.*** Soit $\tau \in \mathrm{CA}(G, A)$, que l'on suppose bijectif. D'après la Proposition 4.2 il vient que $\tau$ est continue. De plus $A^G$ est compact, il vient alors que $\tau^{-1}$ est continue.

Soient $x \in A^G$ et $g \in G$. On a $\tau^{-1}(gx) = \tau^{-1}(g\tau(\tau^{-1}(x))) = \tau^{-1}(\tau(g\tau^{-1}(x)))$ car $\tau$ est $G$-équivariant, et donc $\tau^{-1}(gx) = g\tau^{-1}(x)$ : $\tau^{-1}$ est $G$-équivariant. D'après le théorème de Curtis–Hedlund–Lyndon 4.5, il vient que $\tau^{-1} \in \mathrm{CA}(G, A)$, $\tau$ est donc réversible. ■

## 4.3 Autre propriété topologique

**Proposition 4.8.** *On prend ici $G = \mathbb{Z}^d$. Soient $x \in A^G$ et $a \in A$. Alors :*

1. *Il existe une suite $(f_n)_{n\in\mathbb{N}}$ de configurations $a$-finies telle que $\lim\limits_{n\to+\infty} f_n = x$.*
2. *Il existe une suite $(p_n)_{n\in\mathbb{N}}$ de configurations périodiques telle que $\lim\limits_{n\to+\infty} p_n = x$.*

*Autrement dit, les espaces $\mathcal{F}_a$ et $\mathcal{P}$ sont denses dans $A^G$.*

***Preuve.*** Soit $(z_i)_{i\in\mathbb{N}}$ une énumération de $\mathbb{Z}^d$ et pour $n \in \mathbb{N}$, soit $x \in A^G$ définie par $\forall i \in \mathbb{N}, x_n(z_i) = \begin{cases} x(z_i) \text{ si } i \leqslant n \\ a \text{ si } i > n \end{cases}$. Alors, $\forall n \in \mathbb{N}, x_n$ est une configuration $a$-finie et il est clair que $\lim_{n\to+\infty} x_n = x$.

Pour $n \in \mathbb{N}$, notons $D_n = [\![-n, n]\!]^d$. On pose alors $p_n$ telle que $p_n$ coïncide avec $x$ sur $D_n$ et qui soit périodique de période $2n + 1$. La configuration $p_n$ est donc périodique et $\lim_{n\to+\infty} p_n = x$. ■

## 4.4 Injectivité et surjectivité des automates cellulaires

Un automate cellulaire peut vérifier des propriétés d'injectivité et de surjectivité, comme application de $A^G$ dans $A^G$. Ainsi, un automate cellulaire $\tau$ est injectif si $\forall x \in A^G$, $x$ a au plus un antécédent par l'application $\tau$. Il est surjectif si $\forall x \in A^G, \tau^{-1}(\{x\}) = \{y \in A^G \mid x = \tau(y)\} \neq \varnothing$. Si $\tau$ est un automate non surjectif, les configurations qui n'ont pas de pré-image par l'application $\tau$ sont appelées des *configurations de Jardin d'Éden*.

La proposition suivante donne des propriétés d'injectivité et de surjectivité d'un automate cellulaire $\tau$ et de ses restrictions $\tau_P$ et $\tau_F$ (définies aux sections 3.5 et 3.6), lorsque $G = \mathbb{Z}^d$ et $A$ est fini.

**Proposition 4.9.** *Soit $\tau \in \mathrm{CA}(\mathbb{Z}^d, A)$, où $A$ est fini. On a alors :*

1. *Si $\tau$ est injectif, alors $\tau_F$ et $\tau_P$ sont aussi injectifs.*
2. *Si $\tau_F$ ou $\tau_P$ est surjectif, alors $\tau$ est surjectif.*
3. *Si $\tau_P$ est injectif, alors $\tau_P$ est surjectif.*
4. *Si $\tau_P$ est injectif, alors $\tau$ est surjectif.*

***Preuve.*** La première propriété est évidente, par définition de l'injectivité.

Montrons 2. Supposons que $\tau_F$ est surjectif : toute configuration presque constante a au moins un antécédent. Soit $x \in A^G$ une configuration et montrons que $x$ possède une pré-image. D'après la Proposition 4.8, il existe une suite $(x_n)_n$ de configurations $a$-finies (on prend un état $a \in A$ quelconque) qui converge vers $x$. Comme $\tau_F$ est surjectif, $\forall n \in \mathbb{N}$ il existe $c_n \in A^G$ telle que $\tau(c_n) = x_n$. Comme $A^G$ est compact (car $A$ est fini), il existe une suite-extraite de $(c_n)_n$ qui converge : $c_{\varphi(n)} \underset{n\to+\infty}{\longrightarrow} c \in A^G$. Par continuité de $\tau$, il vient que $\tau(c_{\varphi(n)}) \underset{n\to+\infty}{\longrightarrow} \tau(c)$. Par ailleurs, on a

$$\lim_{n\to+\infty} \tau(c_{\varphi(n)}) = \lim_{n\to+\infty} x_{\varphi(n)} = \lim_{n\to+\infty} x_n = x$$

par unicité de la limite. On a $x = \tau(c)$ : $\tau$ est donc surjectif.

De la même manière, en utilisant la densité de $\mathcal{P}$, on montre que si $\tau_P$ est surjectif, alors $\tau$ l'est.

Supposons maintenant que $\tau_P$ est injectif et montrons que $\tau_p$ est surjectif. Soit $x \in \mathcal{P}$. Par définition des configurations périodiques sur $\mathbb{Z}^d$, il existe $k_1, \ldots, k_d$ tels que pour $i = 1, \ldots, d$, on ait $t_{e_i}^k(x) = x$, où $e_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ est un vecteur de la « base canonique » de $\mathbb{Z}^d$. Soit $\mathcal{K} \subset \mathcal{P}$ l'ensemble des configurations vérifiant ces égalités pour les $k_i$ fixés par $x$ (en particulier, $x \in \mathcal{K}$). Cet ensemble $\mathcal{K}$ est fini (plus précisément, on a $|\mathcal{K}| = |A|^{k_1 \cdots k_d}$), et il ne contient que des configurations périodiques. Comme $\tau$ commute avec l'action de décalage, il vient que $\tau$ commute avec les $t_{e_i}$, on a alors $\tau(\mathcal{K}) \subset \mathcal{K}$. Par hypothèse, $\tau_P$ est injective, alors la restriction $\tau|_{\mathcal{K}}$ l'est aussi. Le cardinal de $\mathcal{K}$ étant fini, il en découle que $\tau(\mathcal{K}) = \mathcal{K}$. Cela montre que si $y \in \mathcal{K}$, alors $y$ a un antécédent dans $\mathcal{K}$ par $\tau$ : en particulier, $x$ à une pré-image périodique. Cela montre bien que $\tau_P$ est surjectif.

Le quatrième point est une conséquence immédiate des points 2. et 3. ■

**Corollaire 4.10.** *Si $G = \mathbb{Z}^d$ et si $A$ est fini, tout automate cellulaire $\tau \in \mathrm{CA}(G, A)$ injectif est surjectif. D'après le Théorème 4.7, l'injectivité est équivalente à la réversibilité.*

***Preuve.*** C'est une conséquence immédiate de la proposition précédente : $\tau$ injectif $\Rightarrow$ $\tau_P$ injective (1.) $\Rightarrow$ $\tau$ surjectif (4.). ■

La réciproque est fausse. Le contre-exemple suivant montre que $\tau$ surjectif n'implique pas $\tau$ injectif.

**Exemple 4.11.** On se place sur $G = \mathbb{Z}$, avec $A = \{0, 1\}$. Soit $\tau \in \mathrm{CA}(G, A)$ l'automate cellulaire défini par

$$\forall x \in A^G, \forall n \in \mathbb{Z}, \tau(x)(n) = x(n) + x(n+1) \mod 2$$

C'est un automate unidimensionnel avec $M = \{0, 1\}$ comme mémoire.

On remarque que la règle locale correspond au *ou exclusif*, ou XOR. La table suivante donne sa définition :

| $x(M)$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| $\mu(x)$ | 0 | 1 | 1 | 0 |

On remarque par ailleurs qu'en prenant une mémoire plus grande, cet automate est un automate élémentaire : en prenant $M = \{-1, 0, 1\}$, cela correspond à la règle 102 (selon la numérotation de Wolfram). En effet, si l'on effectue la somme modulo 2 sur les deux dernières cellules du motif, on a la table suivante :

| $x(M)$ | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|---|
| $\mu(x)$ | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

Montrons que $\tau$ est surjectif mais pas injectif :

Soit une configuration $x \in \{0, 1\}^{\mathbb{Z}}$. Construisons par récurrence une pré-image $\tilde{x}$ de $x$, de la manière suivante :

- On pose $\tilde{x}(0) = 0$.
- Si $n < 0$, on pose $\tilde{x}(n) = x(n) + \tilde{x}(n+1) \mod 2$.
- Si $n > 0$, on pose $\tilde{x}(n) = x(n-1) + \tilde{x}(n-1) \mod 2$.

Avec une telle construction, on a $\tau(\tilde{x}) = x$, ce qui prouve bien que $\tau$ est surjectif. On remarque qu'en prenant $\tilde{x}(0)$ égal à 0 ou 1, on obtient les deux seules configurations ayant pour image $x$.

Soient maintenant les configurations constantes $x_0$ et $x_1$ respectivement dans l'état 0 et l'état 1. Alors $\tau(x_0) = x_0$ et $\tau(x_1) = x_0$, ce qui montre que $\tau$ n'est pas injectif.

Cet automate cellulaire montre aussi que la réciproque du deuxième point de la Proposition 4.9 est fausse : $\tau$ surjectif n'implique pas $\tau_F$ surjectif. En effet, soit la configuration constante égale à 1 : $x = \cdots 11111111 \cdots$. Alors, d'après ce qui précède, les deux pré-images de $x$ sont les deux configurations $x_1 = \cdots 01010101 \cdots$ et $x_2 = \cdots 10101010 \cdots$ qui ne sont pas des configurations presque constantes.

**Corollaire 4.12.** *Soit $\tau \in \mathrm{CA}(\mathbb{Z}^d, A)$ où $A$ est fini. Si $\tau$ est injectif, alors $\tau_F$ est surjectif.*

***Preuve.*** En utilisant le corollaire précédent, il vient que $\tau$ est bijectif. Il est donc réversible, d'après le Théorème 4.7. Si $q$ est un état tranquille de $\tau$, alors c'est aussi un état tranquille de $\tau^{-1}$, c'est-à-dire que la règle locale de $\tau^{-1}$ envoie $(q, \cdots, q)$ sur $q$. Par conséquent, si $x$ est une configuration $q$-finie, $e := \tau^{-1}(x)$ l'est aussi. Donc $\tau(e) = x$, $x$ a une pré-image $q$-finie par $\tau$ : $\tau_F$ est bien surjectif. ■

**Définition 4.13** (Automate pré-injectif)**.** Soient $x_1, x_2 \in A^G$. Soit l'ensemble

$$\mathit{diff}(x_1, x_2) := \{g \in G \,|\, x_1(g) \neq x_2(g)\},$$

des cellules où $x_1$ et $x_2$ diffèrent. On dit que ces configurations sont *presque égales* si l'ensemble $\mathit{diff}(x_1, x_2)$ est fini. Cette définition de presque égalité définit une relation d'équivalence sur $A^G$.

Un automate cellulaire $\tau \in \mathrm{CA}(G, A)$ est dit *pré-injectif* s'il vérifie la propriété suivante : si $x_1$ et $x_2$ sont des configurations presque égales telles que $x_1 \neq x_2$, alors $\tau(x_1) \neq \tau(x_2)$.

On a clairement l'implication suivante : si $\tau$ est injectif, alors $\tau$ est pré-injectif.

## 4.5 Théorème du Jardin d'Éden

Ce théorème a tout d'abord été démontré dans les années 1960 par Moore et Myhill pour $G = \mathbb{Z}^2$, puis a été étendu à certains types de groupes (les groupes moyennables [6]). Ici nous allons le démontrer sur $G = \mathbb{Z}^d$.

**Théorème 4.14** (Jardin d'Éden). *Soit $\tau \in \mathrm{CA}(G, A)$, où $G = \mathbb{Z}^d$ et $A$ est fini. L'automate cellulaire $\tau$ possède des configurations de Jardin d'Éden si et seulement si il existe deux configurations presque constantes différentes qui ont la même image. En d'autres termes :*

$$\tau \text{ est surjectif} \Leftrightarrow \tau \text{ est pré-injectif}$$

Pour montrer ce théorème, nous allons établir quelques résultats et définitions préliminaires. La Proposition 4.16 établira le sens réciproque et la Proposition 4.19 montrera le sens direct.

On avait défini en section 2.3 ce qu'étaient les motifs sur les configurations. Soit $\tau \in \mathrm{CA}(G, A)$ dont on note $M$ une mémoire et $\mu$ sa règle locale. Soit $p \colon D \subset G \to A$ un motif et soit $D' \subset G$ tel que $M(D') \subset D$ : tous les voisins des cellules de $D'$ sont dans $D$. Une application de la règle locale sur le motif $p$ détermine des nouveaux états pour les cellules du domaine $D'$ : on obtient un nouveau motif $p' \colon D' \to A$ tel que $\forall c \in D', p'(c) = \mu(p(c))$. L'application $p \mapsto p'$ est notée $\tau^{D \to D'}$.

Un motif sans pré-image par $\tau$ est appelé *orphelin*. En d'autres termes, le motif $p'$ est un orphelin si et seulement si pour tout motif $p \colon M(D') \to A, \tau^{D \to D'}(p) \neq p'$.

Clairement, toute configuration qui contient une copie translatée d'un motif orphelin est une configuration de Jardin d'Éden. L'inverse est vrai comme le dit la proposition suivante :

**Proposition 4.15.** *Toute configuration de Jardin d'Éden a un sous-motif qui est orphelin. Ainsi, $\tau \in \mathrm{CA}(\mathbb{Z}^d, A)$ n'est pas surjectif si et seulement si il existe un motif orphelin.*

***Preuve.*** Soit $x \in A^{\mathbb{Z}^d}$ une configuration de Jardin d'Éden, et supposons qu'aucun de ses sous-motifs ne soit orphelin. Soit $(z_i)_{i \in \mathbb{N}}$ une énumération de $\mathbb{Z}^d$, et pour $j \in \mathbb{N}$, posons $D_j := \{z_1, \cdots, z_j\}$. Comme le sous-motif $p_j$ avec $D_j$ comme support n'est pas orphelin, il existe une configuration $x_j \in A^{\mathbb{Z}^d}$ telle que $\tau(x_j)$ coïncide avec $x$ sur $D_j$. Cela implique que $\tau(x_j) \underset{j \to +\infty}{\longrightarrow} x$. Par compacité de l'espace des configurations, la suite $(x_n)_n$ possède une sous-suite convergente $(x_{\varphi(n)})_n$ qui converge vers une configuration $e \in A^{\mathbb{Z}^d}$. Par continuité de $\tau$, la suite $\big(\tau(x_{\varphi(n)})\big)_n$ converge vers $\tau(e)$. D'autre part, on a

$$\lim_{j \to +\infty} \tau(x_{\varphi(j)}) = \lim_{i \to +\infty} \tau(x_i) = x,$$

donc $x = \tau(e)$, par unicité de la limite, ce qui signifie que $x$ n'est pas une configuration de Jardin d'Éden. ■

Pour les propositions à venir, on se fixe un $\tau \in \mathrm{CA}(G, A)$, où $G = \mathbb{Z}^d$ et $A$ est fini.

**Proposition 4.16.** *Si $\tau$ n'est pas surjectif, alors $\tau$ n'est pas pré-injectif.*

Pour la démonstration de cette proposition, ainsi que pour la Proposition 4.19, on utilise la notion d'hypercube sur $\mathbb{Z}^d$ :

**Définition 4.17** (Hypercube). Soit $n \in \mathbb{N}^*$ et soient $(k_1, \cdots, k_d) \in \mathbb{Z}^d$. L'hypercube de taille $n^d$ déterminé par les coins $(k_1, \cdots, k_d)$ est défini par le domaine :

$$D := \big\{(x_1, \cdots, x_d) \in \mathbb{Z}^d \,\big|\, k_i \leqslant x_i < k_i + n \text{ pour } i = 1, \cdots, d\big\}$$

**Lemme 4.18.** *Pour tous $d, n, a, r \in \mathbb{N}^*$, il existe $k \in \mathbb{N}$ de sorte que :*

$$\left(a^{n^d} - 1\right)^{k^d} < a^{(kn-2r)^d}$$

***Preuve.*** En passant au logarithme en base $a$ qui est une fonction strictement croissante, l'inégalité à démontrer est équivalente à :

$$\log_a\left(\left(a^{n^d} - 1\right)^{n^d}\right) < (kn - 2r)^d \Leftrightarrow \log_a\left(a^{n^d} - 1\right) < \left(k - \frac{2r}{n}\right)^d$$

Par ailleurs, on a $\log_a\left(a^{n^d} - 1\right) < \log_a\left(a^{n^d}\right) = n^d = \lim_{k \to +\infty}\left(n - \frac{2r}{k}\right)^d$. On en déduit qu'il existe $k \in \mathbb{N}^*$ tel que l'inégalité soit vérifiée. ■

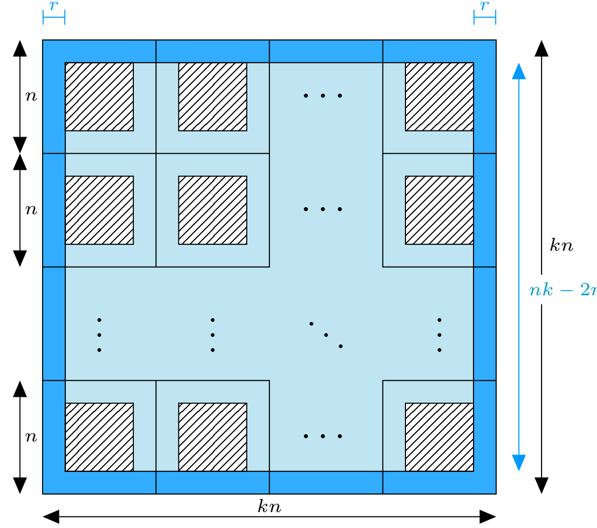



FIGURE 8 – Découpage d'un hypercube de taille $(kn)^d$ en $k^d$ hypercubes de taille $n^d$ (sur le schéma $d = 2$).

On peut maintenant retourner à la démonstration de la Proposition 4.16 :

***Preuve.*** Supposons que $\tau$ n'est pas surjectif et fixons-nous un état $q \in A$. Montrons qu'il existe deux configurations $q$-finies distinctes $x_1, x_2$ telles que $\tau(x_1) = \tau(x_2)$. On pose $t = \mu(q, \cdots, q)$ et $a = |A|$. Prenons $r$ suffisamment grand de sorte de $\tau$ soit un automate cellulaire de rayon $r$.

D'après la Proposition 4.15, il existe un motif fini $p$ tel que toute configuration contenant $p$ soit un Jardin d'Éden. On peut compléter $p$ avec des copies de l'état $t$ de sorte que le domaine de $p$ soit un hypercube de taille $n^d$ pour un certain entier strictement positif $n$.

Soit $k \in \mathbb{N}^*$ arbitraire et considérons un hypercube C de taille $(kn)^d$. On peut alors partitionner $C$ en $k^d$ hypercubes deux-à-deux disjoints de taille $n^d$ (voir sur la Figure 8 dans le cas $d = 2$)

Soit $C'$ l'hypercube de taille $(kn - 2r)^d$ centré dans $C$ (en bleu clair sur la Figure 8) et soit $K := \{x \in A^{\mathbb{Z}^d} \,|\, \text{supp}_q(x) \subset C'\}$ l'ensemble des configurations $q$-finies telles que les cellules qui ne sont pas dans l'état $q$ soient dans $C'$. Il y a $a^{|C'|}$ éléments dans $K$.

Pour tout $x \in K$, le $t$-support de $\tau(x)$ est inclus dans $C$. De plus, $\tau(x)$ ne peut contenir le motif $p$ dans aucun des $k^d$ sous-hypercubes de taille $n^d$, car $p$ est un motif orphelin. Cela signifie qu'il y a au plus $\left(a^{n^d} - 1\right)^{k^d}$ configurations possibles pour $\tau(x)$. Mais d'après le Lemme 4.18, pour un certain $k$, $(a^{n^d} - 1)^{k^d} < a^{(kn-2r)^d} = a^{|C'|} = |K|$. Donc il existe $x_1, x_2 \in K$ telles que $x_1 \neq x_2$ alors que $\tau(x_1) = \tau(x_2)$.

■

**Proposition 4.19.** *Si $\tau$ n'est pas pré-injectif, alors $\tau$ a des configurations de Jardin d'Éden.*

***Preuve.*** Soient $x_1, x_2 \in A^{\mathbb{Z}^d}$ presque égales et supposons que $x_1 \neq x_2$ mais $\tau(x_1) = \tau(x_2)$. Soit $r$ suffisamment grand pour que $\tau$ soit un automate de rayon $\frac{r}{2}$. Prenons $n$ suffisamment grand de sorte qu'il y ait un hypercube $D'$ de taille $(n - 2r)^d$ contenant toutes les cellules où $x_1$ et $x_2$ diffèrent ($n$ existe bien car $x_1$ et $x_2$ sont presque égales) : $\mathit{diff}(x_1, x_2) \subset D'$. Soit $D$ l'hypercube de taille $n^d$ autour de $D'$ concentrique avec $D'$ et soient $p_1 : D \to A$ et $p_2 : D \to A$ les sous-motifs de $x_1$ et $x_2$ respectivement avec $D$ comme support.

Dans toute configuration $x$ qui a le motif $p_1$, on peut remplacer $p_1$ par $p_2$ sans changer $\tau(x)$. En effet, les cellules $c \in \mathbb{Z}^d$ qui ne contiennent par des éléments de $\mathit{diff}(x_1, x_2)$ ne voient aucun changement dans leur voisinage. Ces cellules $c$ dont le voisinage contient des éléments de $\mathit{diff}(x_1, x_2)$ sont à une distance supérieure ou égale à $\frac{r}{2}$ de $D'$ alors leur voisinage est entièrement contenu dans $D$. Par conséquent, $\tau(x)(c) = \tau(x_1)(c) = \tau(x_2)(c) = \tau(x')(c)$ où $x'$ est la configuration obtenue en remplaçant $p_1$ par $p_2$ dans $x$.

Comme dans la preuve de la Proposition 4.16, soit $k \in \mathbb{N}^*$ fixé arbitrairement et soit $C$ un hypercube de taille $(kn)^d$ constitué de $k^d$ sous-hypercubes de taille $n^d$ deux-à-deux disjoints. Soit $C'$ l'hypercube de taille $(kn - 2r)^d$ centré dans $C$. Si $\tau$ est surjectif, alors tous les motifs de domaine $C'$ ont une pré-image de domaine $C$ et d'après ce qui précède, il y a une pré-image là où aucun des $k^d$ sous-hypercubes de taille $n^d$ ne contient une copie de $p_1$. Mais il y a seulement $\left(a^{|D|} - 1\right)^{k^d} = \left(a^{n^d} - 1\right)^{k^d}$ motifs avec $C$ comme domaine, tandis qu'il y a $a^{(kn-2r)^d}$ motifs de domaine $C'$. Il suit du Lemme 4.18 que des motifs n'ont pas de pré-image. L'automate $\tau$ a donc des configurations de Jardin d'Éden. ■

**Corollaire 4.20.** *Si $\tau$ est surjectif, alors $\tau_F$ est injectif.*

***Preuve.*** C'est une conséquence immédiate du théorème précédent, et du fait que si deux configurations $x$ et $y$ sont $a$-finies (pour un certain $a \in A$), alors elles sont presque égales. ■

**Exemple 4.21.** Le Jeu de la Vie (voir Exemple 3.2) n'est pas surjectif : soient $x_1$ la configuration où toutes les cellules sont mortes et soit $x_2$ une configuration où exactement une cellule est vivante. Alors $x_1$ et $x_2$ ont toutes deux la même image par cet automate : $x_1$. Ainsi, par la Proposition 4.15, il existe des motifs orphelins. Le plus petit orphelin trouvé de nos jours a un domaine de taille 92 (voir Figure 9) et a été découvert en 2011 par M.Heule, C.Hartman, K.Kwekkeboom et A.Noels [7].

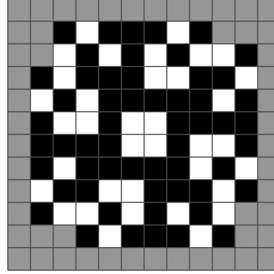


Figure 9 – Les cellules noires sont vivantes et les grises ne sont pas dans le domaine

## 4.6 Cas unidimensionnel

Dans cette partie on prend $G = \mathbb{Z}$ et on va donner des propriétés qui seront utiles pour montrer des résultats dans le paragraphe 5.2.

**Définition 4.22** (Configurations séparées et asymptotiques)**.** Soient $x$ et $y$ deux configurations de $A^{\mathbb{Z}}$ et soit $n \in \mathbb{N}^*$. On dit que :

- Elles sont *positivement asymptotiques* si $\exists i_0 \in \mathbb{Z}, \forall i \geqslant i_0$, $x(i) = y(i)$. Autrement dit, pour $i$ suffisamment grand, $x(i) = y(i)$.
- Elles sont *négativement asymptotiques* si $\exists i_0 \in \mathbb{Z}, \forall i \leqslant i_0$, $x(i) = y(i)$. Autrement dit, pour $i$ suffisamment petit, $x(i) = y(i)$.
- Elles sont *positivement $n$-séparées* (respectivement *négativement $n$-séparées*) si pour $i \in \mathbb{Z}$ suffisamment grand (respectivement suffisamment petit), il existe $j \in [\![i, i+n-1]\!]$ tel que $x(j) \neq y(j)$.
- Elles sont *totalement $n$-séparées* si pour tout $i \in \mathbb{Z}$, il existe $j \in [\![i, i+n-1]\!]$ tel que $x(j) \neq y(j)$.

**Proposition 4.23.** *Soit $A$ un ensemble fini et soit $\tau \in \mathrm{CA}(\mathbb{Z}, A)$ un automate cellulaire surjectif, et supposons que la mémoire est formée de $m$ entiers consécutifs, avec $m \geqslant 2$. Soient $x, y \in A^{\mathbb{Z}}$ telles que $x \neq y$ et $\tau(x) = \tau(y)$. Alors exactement une des trois conditions suivantes est vérifiée :*

1. *Les configurations $x$ et $y$ sont négativement asymptotiques et positivement $(m-1)$-séparées,*
2. *Les configurations $x$ et $y$ sont positivement asymptotiques et négativement $(m-1)$-séparées,*
3. *Les configurations $x$ et $y$ sont toutes deux positivement et négativement $(m-1)$-séparées.*

***Preuve.*** Les conditions 1. à 3. s'excluent deux-à-deux. Il suffit donc de montrer qu'au moins une des trois est vraie pour $x$ et $y$.

Montrons d'abord que si $x(n) \neq y(n)$, alors il ne peut pas y avoir de segment de longueur $m-1$ des deux côtés de $n$ où $x$ et $y$ coïncident. Supposons le contraire : il existe $k_1 < n$ et $k_2 > n$ tels que, pour $i$ dans $[\![k_1 - (m-1), k_1]\!]$ et dans $[\![k_2, k_2 + (m-1)]\!]$, on ait $x(i) = y(i)$. On peut alors remplacer dans la configuration $x$ les états des cellules $k_1, \ldots, k_2$ par les états de $y$ sans affecter $\tau(x)$. D'après le Théorème du Jardin d'Éden 4.14, comme $\tau$ est surjectif, il vient que $\tau$ est pré-injectif. On obtient une contradiction car la configuration $x'$ ainsi obtenue diffère de $x$ sur un nombre fini de cellules, $x$ et $x'$ sont donc presque égales avec $\tau(x) = \tau(x')$.

Il vient alors que $x$ et $y$ sont positivement asymptotiques ou bien positivement $(m-1)$-séparées. Sinon, il y aurait un segment de longueur $(m-1)$ sur lequel $x$ et $y$ coïncideraient, une cellule $n$ telle que $x(n) \neq y(n)$ et un autre segment de longueur $(m-1)$ où $x$ et $y$ coïncideraient à nouveau. Un raisonnement symétrique montre que $x$ et $y$ doivent être négativement asymptotiques ou négativement $(m-1)$-séparées.

De plus, $x$ et $y$ ne peuvent pas être toutes les deux positivement et négativement asymptotiques, ce serait encore une contradiction au Théorème du Jardin d'Éden. ■

**Proposition 4.24.** *Soit $\tau \in \mathrm{CA}(\mathbb{Z}, A)$, où $A$ est un ensemble fini. Si $\tau_P$ est injectif, alors $\tau$ est injectif.*

***Preuve.*** Prenons comme mémoire de $\tau$ un segment formé de $m$ entiers consécutifs. Si $m = 1$, alors la mémoire est réduite à un seul élément et $\tau$ est trivialement réversible (et en particulier injectif). Si $m \geqslant 2$, supposons alors par l'absurde que $\tau$ ne soit pas injectif : $\exists x, y \in A^{\mathbb{Z}}$ distinctes telles que $\tau(x) = \tau(y)$. Comme $\tau_P$ est injectif, il vient d'après la Proposition 4.9 2. que $\tau$ est surjectif. Ainsi, d'après la Proposition 4.23, $x$ et $y$ sont positivement ou négativement $(m-1)$-séparées. Ces deux cas sont symétriques, supposons que $x$ et $y$ sont positivement $(m-1)$-séparées.

Il y a un nombre fini de motifs différents de longueurs $m-1$ dans $x$ et $y$, alors il existe deux nombres positifs $k_1$, $k_2$ (grands) tels que dans $x$ et $y$ à la fois, les chaînes de longueurs $m-1$ commençant en position $k_1$ et $k_2$ soient identiques. Plus précisément :

$$\exists k_2 \geqslant k_1 + m, \forall i \in [\![0, m-1[\![, x(k_1 + i) = x(k_2 + i) \text{ et } y(k_1 + i) = y(k_2 + i)$$

Comme $x$ et $y$ sont positivement $(m-1)$-séparées, on prend $k_1$ suffisamment grand de sorte que $x(k_1+i) \neq y(k_1+i)$ pour $0 \leqslant i < m-1$.

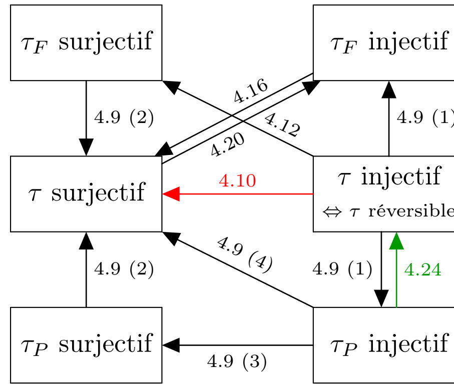
Considérons les configurations périodiques $x_p$ et $y_p$ qui sont invariantes par décalage de $k_2 - k_1$ cellules et qui coïncident avec $x$ et $y$ respectivement sur les cellules $k_1, k_1+1, \cdots, k_2-1$ : pour $n \in \mathbb{N}$ et $i \in [\![k_1, k_2[\![$ :

$$x_p\left(i + n(k_2 - k_1)\right) = x(i)$$
$$y_p\left(i + n(k_2 - k_1)\right) = y(i)$$

Alors, pour tout $j \in \mathbb{Z}$, il existe $i$ dans $[\![k_1, k_2[\![$ tel que les segments de longueur $m$ dans $x_p$ et $y_p$ commençant à la position $j$ sont les mêmes que les segments de longueur $m$ dans $x$ et $y$ commençant en position $i$. Comme $m$ est la taille de la mémoire de $\tau$ et $\tau(x) = \tau(y)$, il vient que $\tau(x_p) = \tau(y_p)$. Par ailleurs, $x_p \neq y_p$, alors $\tau_P$ n'est pas injectif, ce qui est une contradiction.

■

La figure suivante récapitule toutes les différentes propriétés qui ont été démontrées précédemment concernant l'injectivité et la surjectivité d'un automate cellulaire $\tau \in \mathrm{CA}(\mathbb{Z}^d, A)$ (la flèche verte correspond à un résultat seulement valable sur $\mathbb{Z}^1$ uniquement, et la flèche rouge est un résultat important) :



# 5 Graphes de de Bruijn

Dans cette section, on ne va s'intéresser qu'aux automates cellulaires unidimensionnels définis en section 3.4. On va voir une autre représentation de ces automates : les graphes étiquetés orientés.

**Définition 5.1** (Graphe orienté)**.** Un *graphe orienté* $G$ est un quadruplet $G = (V, E, q, t)$, où $V$ et $E$ sont des ensembles finis et $q, t \colon E \to V$ des applications de $E$ dans $V$. L'ensemble $V$ est appelé l'ensemble des *sommets* de $G$ et l'ensemble $E$ est appelé l'ensemble des *arêtes* de $G$. Si $e \in E$ est une arête de $G$ telle que $q(e) = v_1$ et $t(e) = v_2$, on dit que l'arête $e$ part du sommet $v_1$ pour aller au sommet $v_2$. On dit alors que $v_1$ est la *queue* de $e$ et que $v_2$ en est la *tête*. Il est possible que la tête et la queue d'une arête soient un seul et même sommet du graphe, on parle alors de *boucle*. Dans la suite, par abus de notation, on notera un graphe sous la forme d'un couple : $G = (V, E)$.

Si $G = (V, E)$ est un graphe, un *chemin de longueur* $k$ de $G$ est une suite $(e_1, \cdots, e_k)$ d'arêtes telles que $\forall i \in [\![1, k-1]\!]$, $t(e_i) = q(e_{i+1})$. On définit de même un *chemin bi-infini* comme étant une suite $(e_i)_{i\in\mathbb{Z}}$ d'arêtes telle que $\forall i \in \mathbb{Z}, t(e_i) = q(e_{i+1})$. Un *cycle de longueur* $k$ est un chemin de longueur $k \geqslant 1$ vérifiant $t(p_k) = q(p_1)$.

Un graphe orienté est dit *étiqueté* si il est muni d'une application *étiquette* $\lambda \colon E \to A$, où $A$ est un ensemble. L'étiquette d'un chemin $(e_i)_i$ (fini ou non), parfois appelée *mot*, est alors la suite $(\lambda(e_i))_i$ des étiquettes de chacune des arêtes du chemin.

On peut représenter de manière visuelle un graphe : tous les sommets sont représentés sur le plan et une arête (orientée) est représentée par une flèche allant de sa queue vers sa tête.

**Exemple 5.2.** On prend ici $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ et $E = \{(v_0, v_1), (v_0, v_3), (v_1, v_4), (v_2, v_5), (v_3, v_1), (v_4, v_3), (v_5, v_5)\}$. On attribut à chacune de ces arêtes une étiquette dans l'ensemble $\{a, b, c\}$. Le graphe $G = (V, E)$ obtenu est représenté Figure 10. On voit alors, par exemple, que la suite de sommets $(v_0, v_1, v_4, v_3, v_1)$ est un chemin dans ce graphe, d'étiquette $aacb$.

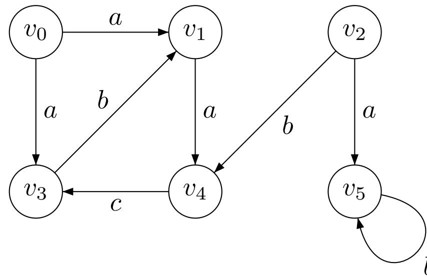



FIGURE 10 – Un exemple de graphe orienté étiqueté

## 5.1 Graphe de de Bruijn

On va ici s'intéresser à une certaine catégorie de graphes, qui tiennent leur nom du mathématicien néerlandais Nicolaas Govert de Bruijn qui les a décrits en 1946.

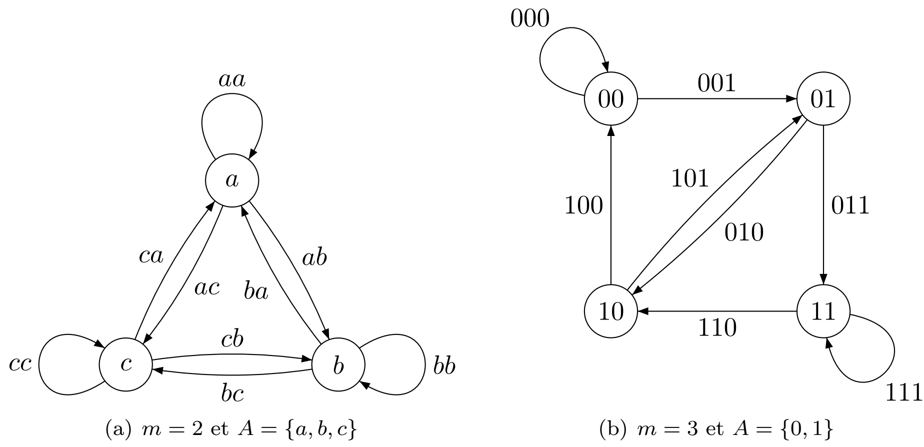
**Définition 5.3** (Graphe de de Bruijn)**.** Soit $A$ un ensemble fini et soit $m \in \mathbb{N}^*$. On définit le *graphe de de Bruijn* de taille $m$ sur l'alphabet $A$ comme étant le graphe orienté tel que :

- L'ensemble des sommets est $V = A^{m-1}$.
- L'ensemble des arêtes est $E = A^m$. Une arête $(a_1, a_2, \cdots, a_m)$ sera notée $a_1 a_2 \cdots a_m$.
- Pour toute arête de la forme $a_1 a_2 \cdots a_m \in E$, on a $q(a_1 a_2 \cdots a_m) = a_1 a_2 \cdots a_{m-1}$ et $t(a_1 a_2 \cdots a_m) = a_2 a_3 \cdots a_m$.

En d'autres termes, dans un tel graphe, quel que soit le mot $u \in A^{m-2}$ et quelles que soient les lettres $s, t \in A$, il existe une arête du sommet $su$ au sommet $ut$.

**Exemple 5.4.** Prenons par exemple $m = 2$ et $A = \{a, b, c\}$. Le graphe de de Bruijn sera donc le graphe $G = (V, E)$ où $V = A^1$ et $E = A^2$. Ce graphe a donc 3 sommets et 6 arêtes (voir Figure 11 (a)). On a aussi représenté sur la Figure 11 (b) le graphe de taille $m = 3$ sur l'alphabet $A = \{0, 1\}$. C'est un graphe à 4 sommets et 8 arêtes.

On remarque que dans un tel graphe, il y a $|A|^{m-1}$ sommets et $|A|^m$ arêtes. De plus, chaque sommet à un degré entrant et un degré sortant valant $|A|$, où le degré entrant (respectivement sortant) est le nombre d'arêtes qui arrivent sur le sommet (respectivement qui partent du sommet).




(a) $m = 2$ et $A = \{a, b, c\}$ (b) $m = 3$ et $A = \{0, 1\}$

FIGURE 11 – Exemples de graphes de de Bruijn

**Définition 5.5.** Soit $\tau \in \mathrm{CA}(\mathbb{Z}, A)$ (où $A$ est fini). On suppose que la mémoire est constituée de $m$ entiers consécutifs. On définit la *représentation de de Bruijn* de l'automate $\tau$ comme étant le graphe de de Bruijn de taille $m$ sur l'alphabet $A$, où chaque arête $e \in A^m$ est marquée par $\mu(e) \in A$ ($\mu$ désigne la règle locale de l'automate $\tau$).

Ainsi, quelle que soit la configuration $x \in A^{\mathbb{Z}}$, il existe un chemin bi-infini $p\colon \mathbb{Z} \to E$ tel que $\forall i \in \mathbb{Z}, p(i) = x(i)x(i+1)\cdots x(i+m-1)$ : les arêtes parcourues par $p$ correspondent à celles vues par la « fenêtre de taille $m$ » sur $x$. La correspondance entre $x$ et $p$ est une bijection.

La théorie des automates finis permet de mettre en place une méthode pour déterminer l'ensemble des motifs orphelins pour un automate cellulaire qui n'est pas surjectif (voir en Annexe B).

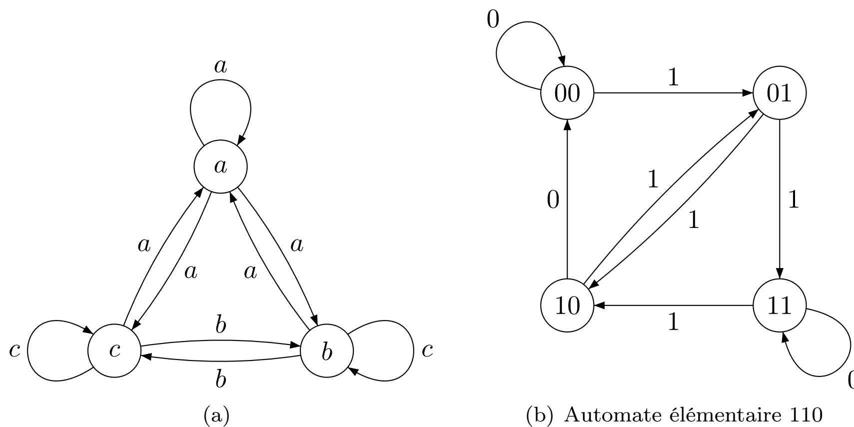
Reprenons les cas particuliers de l'Exemple 5.4. Dans le premier cas, on prend comme règle locale l'application définie ainsi :

| $x(M)$ | $aa$ | $ab$ | $ac$ | $ba$ | $bb$ | $bc$ | $ca$ | $cb$ | $cc$ |
|---|---|---|---|---|---|---|---|---|---|
| $\mu(x)$ | $a$ | $a$ | $a$ | $a$ | $c$ | $b$ | $a$ | $b$ | $c$ |

Pour le second exemple, prenons l'automate élémentaire numéro 110 défini en section 3.4 dont on rappelle la définition de la règle locale $\mu_{110}$ :

| $x(M)$ | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|---|
| $\mu_{110}(x)$ | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

Les graphes de ces deux automates sont représentés Figure 12.




(a) (b) Automate élémentaire 110

FIGURE 12 – Représentation de de Bruijn de deux automates cellulaires

Ainsi, quel que soit le mot $a_1 a_2 \cdots a_m \in A^m$, le graphe ainsi obtenu possède une arête marquée par $\mu(a_1 \cdots a_m)$ allant du sommet $a_1 \cdots a_{m-1}$ au sommet $a_2 \cdots a_m$. Il contient toutes les informations à propos de la règle locale de l'automate. On remarque par ailleurs que la représentation de de Bruijn d'un automate cellulaire $\tau$ ne

prend pas en compte la position du voisinage mais seulement sa taille. Ainsi si $\sigma$ désigne le décalage de $r = 1$ ($\sigma\colon c \mapsto c + 1$), le graphe de l'automate $\tau$ sera le même que le graphe de l'automate $\tau \circ \sigma^k$, pour tout $k \in \mathbb{Z}$. Ceci ne pose pas de problème pour la question d'injectivité et de surjectivité de l'automate $\tau$ car ces propriétés ne sont pas affectées par l'action de décalage.

Soit $p$ un chemin bi-infini dans la représentation de de Bruijn d'un automate cellulaire $\tau$. À partir de ce chemin, on peut définir deux applications de $\mathbb{Z}$ dans $A$, c'est à dire deux configurations de $A^{\mathbb{Z}}$ :

— La suite $c_p$ obtenue en lisant le premier symbole du nom des sommets parcourus par $p$.

— La suite $f_p$ obtenue en lisant l'étiquette des arêtes de $p$.

Les étiquettes étant des résultats de la règle locale de l'automate, il est possible de trouver un entier $k \in \mathbb{Z}$ tel que $\sigma^k(f_p) = \tau(c_p)$, la valeur de $k$ dépendant de la position du voisinage dans l'automate cellulaire.

**Définition 5.6** (Diamant dans un graphe)**.** Soit $G$ un graphe étiqueté. On dit que $G$ possède un *diamant* si il existe deux chemins finis distincts ayant la même étiquette, qui commencent et se terminent sur le même sommet.

**Proposition 5.7.** *Soit $\tau \in \mathrm{CA}(\mathbb{Z}, A)$. Il est maintenant possible d'interpréter les propriétés de surjectivité et d'injectivité de $\tau$ grâce à sa représentation de de Bruijn :*

- *L'automate $\tau$ est injectif si et seulement si deux chemins bi-infinis différents ont toujours des étiquettes différentes.*
- *L'automate est surjectif si et seulement si pour toute configuration $x \in A^{\mathbb{Z}}$, il existe un chemin bi-infini étiqueté par $x$. Ceci est équivalent à dire que le graphe ne possède pas de diamant.*
- *Un orphelin est un mot sur l'alphabet $A$ qui n'est l'étiquette d'aucun chemin du graphe.*

***Preuve.*** Le premier et le troisième points sont évidents.

Montrons le deuxième point. Supposons que la représentation de de Bruijn de $\tau$ possède un diamant. Soient $x_1$ et $x_2$ les deux configurations qui correspondent aux deux chemins de ce diamant. Alors $x_1$ et $x_2$ sont distinctes, presque égales et ont la même image par $\tau$. Il vient alors que $\tau$ n'est pas pré-injectif, ce qui d'après le Théorème du Jardin d'Éden 4.14 est équivalent à dire que $\tau$ n'est pas surjectif. Inversement, si $\tau$ n'est pas surjectif, alors $\tau$ n'est pas pré-injectif et donc il existe deux configurations presque égales et distinctes $x_1$, $x_2$ telles que $\tau(x_1) = \tau(x_2)$. Ces deux configurations correspondent alors à deux chemins finis dans la représentation de de Bruijn qui forment un diamant. ∎

## 5.2 Graphe pair

Pour tester si un automate cellulaire est injectif ou surjectif, on va utiliser un type de graphe particulier : le graphe pair.

**Définition 5.8** (Graphe pair)**.** Soit $\tau \in \mathrm{CA}(\mathbb{Z}, A)$ et notons $G = (V, E)$ sa représentation de de Bruijn. Le *graphe pair* formé à partir de $G$ est le graphe $\tilde{G} = (\tilde{V}, \tilde{E}, \tilde{q}, \tilde{t})$ où :

— L'ensemble des sommets est défini par $\tilde{V} = V \times V$,

— Il y a une arête d'étiquette $a \in A$ du sommets $(u_1, u_2) \in V \times V$ vers $(v_1, v_2) \in V \times V$ si et seulement si il y a deux arêtes, toutes deux d'étiquette $a$, l'une allant de $u_1$ vers $v_1$ et l'autre de $u_2$ vers $v_2$ dans la représentation de de Bruijn de $\tau$.

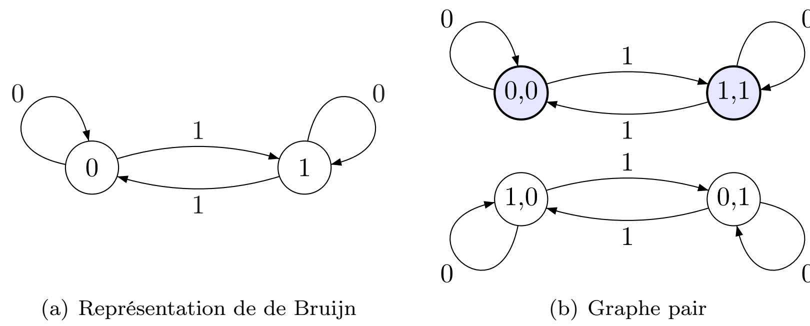
Un chemin bi-infini $p$ dans le graphe pair correspond à deux chemins dans la représentation de de Bruijn originale, obtenus en lisant la première ou la seconde composante du sommet dans le graphe pair. Ces deux chemins ont la même étiquette, donc correspondent à deux configurations $x_1^p$ et $x_2^p$ ayant la même image par $\tau$. L'application $p \mapsto (x_1^p, x_2^p)$ est une bijection entre les chemins bi-infinis $p$ et les configurations qui vérifient $\tau(x_1^p) = \tau(x_2^p)$.

Soit $\Delta := \{(v, v) \mid v \in V\}$ l'ensemble des sommets du graphe pair dont les deux composantes sont égales. On dit que $\Delta$ est l'ensemble des *sommets diagonaux*. Si $p$ est un chemin bi-infini qui ne passe uniquement que par des sommet de $\Delta$, alors on a $x_1^p = x_2^p$ : $p$ ne donne pas deux configurations différentes ayant la même image. Seuls les chemins ayant un sommet en dehors de $\Delta$ donnent de telles configurations.

**Exemple 5.9.** Reprenons l'exemple 4.11 du XOR décrit précédemment. Sa représentation de de Bruijn est donnée sur la Figure 13 (a) et son graphe pair est représenté sur la Figure 13 (b), les sommets diagonaux sont bleutés.

**Théorème 5.10.** *Soit $\tau \in \mathrm{CA}(\mathbb{Z}, A)$. Alors $\tau$ est :*

1. *Non injectif si et seulement si son graphe pair possède un cycle qui contient un sommet en dehors de $\Delta$.*
2. *Non surjectif si et seulement si son graphe pair possède un cycle qui contient un sommet de $\Delta$ et un sommet en dehors de $\Delta$.*




(a) Représentation de de Bruijn (b) Graphe pair

FIGURE 13 – Automate cellulaire XOR

***Preuve.*** 1. Supposons que $\tau$ ne soit pas injectif. D'après la Proposition 4.24, il existe deux configurations périodiques $x, y \in A^{\mathbb{Z}}$ distinctes qui ont la même image par $\tau$. Le chemin y correspondant dans le graphe pair est un cycle qui contient un sommet en dehors de $\Delta$. Inversement, si un tel cycle existe, alors les deux configurations, distinctes, y correspondant ont la même image par $\tau$ : $\tau$ n'est donc pas injectif.

2. Soit $a \in A$ un état arbitraire. Si $\tau$ n'est pas injectif, alors $\tau_F$ n'est pas injectif, par la Proposition 4.9 2. Il existe alors deux configurations $x$, et $y$ $a$-finies différentes qui ont la même image par $\tau$. Le chemin y correspondant dans le graphe pair consiste en une boucle à l'intérieur de $\Delta$, correspondant aux cellules dans l'état $a$ à « moins l'infini », suivie par un cycle allant en dehors de $\Delta$ (pour les cellules du $a$-support) et retournant à la même boucle à l'intérieur de $\Delta$, pour les cellules à « plus infini ». Ainsi, il existe un cycle parcourant des sommets en dehors de $\Delta$. Inversement, si un tel cycle existe, la représentation de de Bruijn de $\tau$ possède un diamant. D'après la Proposition 5.7 : $\tau$ n'est pas surjectif. ■

Dans l'exemple précédent du XOR, en appliquant ce théorème, on voit que le cycle $\big((1,0),(0,1),(1,0)\big)$ contient des sommets non diagonaux, l'automate n'est donc pas injectif. Par ailleurs, tous les cycles de ce graphe contiennent soit toujours de sommets de $\Delta$, soit jamais, cet automate est donc surjectif. Ceci correspond bien à ce que l'on avait montré dans l'Exemple 4.11.

**Exemple 5.11.** Revenons à la règle 110 définie précédemment et générons son graphe pair (voir Figure 14). Cette fois-ci, le graphe est un petit peu plus gros, car on a ici $m = 3$ comme taille de la mémoire. Par ce graphe, on voit que l'automate $\tau_{110}$ n'est pas surjectif : le cycle $\big((01,01),(11,10),(10,01),(01,11),(10,10),(01,01)\big)$ (représenté en rouge sur la figure) contient des sommets de $\Delta$ et plusieurs sommets en dehors de $\Delta$. Ce même cycle nous montre que $\tau_{110}$ n'est pas injectif non plus (le contraire aurait d'ailleurs été impossible d'après la Proposition 4.10).

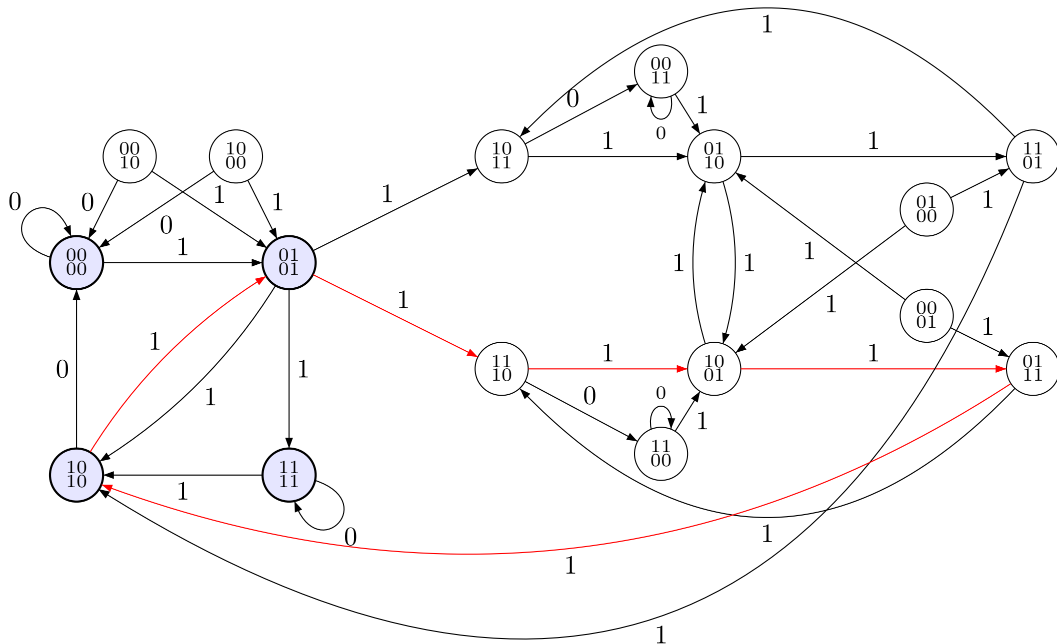


FIGURE 14 – Graphe pair de l'automate élémentaire 110

# 6 Test algorithmique

Le Théorème 5.10 nous donne une méthode algorithmique qui permet de tester si un automate cellulaire unidimensionnel est injectif ou surjectif. En effet, si on arrive à trouver un cycle dans le graphe pair passant ou non par des sommets diagonaux, on pourra conclure quant à la surjectivité ou l'injectivité de l'automate cellulaire. On rappelle que, par le Théorème 4.10, l'injectivité et la réversiblité sont deux notions équivalentes. Dans cette section, nous allons nous intéresser à un algorithme qui va répondre à cette question. L'algorithme que nous allons étudier, sera quadratique en la taille du graphe de de Bruijn[4]. Jarkko Kari a par ailleurs montré que pour des automates cellulaires sur $\mathbb{Z}^2$, la question de réversibilité est indécidable [8].

## 6.1 Graphe réduit

On remarque que dans le théorème 5.10, on est seulement intéressé par des cycles qui contiennent des sommets en dehors de l'ensemble des sommets diagonaux $\Delta$. On peut alors fusionner tous ces sommets diagonaux en un seul sommet, que l'on notera $\Delta$ dans la suite. Le graphe ainsi obtenu est appelé *graphe réduit*.

Sur la Figure 15 est représenté le graphe réduit obtenu à partir du graphe pair de l'automate élémentaire 110 vu dans l'Exemple 5.11.

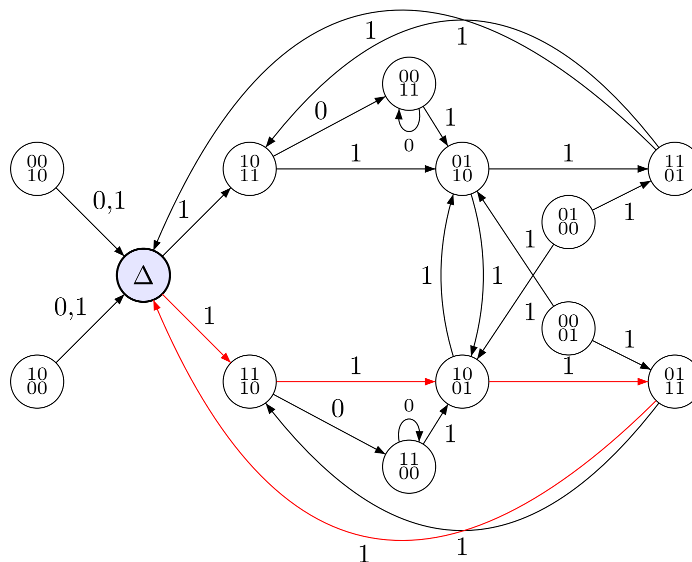


FIGURE 15 – Le graphe réduit de la règle 110 (l'arête avec deux étiquettes correspond en fait à deux arêtes)

Le Théorème 5.10 peut alors se réécrire ainsi :

**Théorème 6.1.** *Soit* $\tau \in \mathrm{CA}(\mathbb{Z}, A)$. *Alors* $\tau$ *est :*

- *injectif si et seulement si il n'y a pas de cycle dans le graphe réduit,*
- *surjectif si et seulement si il n'y a pas de cycle passant par* $\Delta$.

Dans la Figure 15, on a représenté en rouge le même cycle que dans l'Exemple 5.11 qui montrait la non-surjectivité de l'automate.

## 6.2 Parcours en profondeur

Un algorithme efficace pour trouver les cycles dans un graphe est l'algorithme du *parcours en profondeur* (ou *depth-first search* en anglais, abrégé en *dfs*)[9].

Soit $G = (V, E)$ un graphe. À chaque sommet $v \in V$, on associe une couleur $v_{\text{couleur}}$ parmi les trois : blanc, gris ou noir. Initialement, tous les sommets sont blancs, c'est-à-dire qu'ils n'ont pas encore été visités. Le principe du *dfs* est qu'il va d'abord parcourir les sommets les plus profonds possibles en premier. Cela signifie que lorsqu'un sommet $v$ est rencontré, il va le colorier en gris, puis va aller visiter tous les sommets adjacents à $v$, c'est-à-dire tous les sommets $w$ tels que $(v, w)$ soit une arête du graphe. Une fois tous ces sommets visités, le sommet $v$ sera colorié en noir. Tant que tous les sommets ne sont pas noirs, c'est qu'ils n'ont pas été tous visités, l'algorithme continue de parcourir ces sommets. De plus, deux marqueurs temporels permettent de savoir à quel instant dans l'exécution de l'algorithme un sommet $v$ a été visité puis quitté : on note $v_d$ la date où le sommet $v$ a

été découvert pour la première fois et $v_f$ l'étape où on a fini de parcourir les sommets adjacents à $v$. Ces deux marqueurs sont des entiers compris entre 1 et $2|V|$. Ces indicateurs ne nous serviront pas dans notre cas pour trouver les cycles, mais ils permettent de connaître le comportement du graphe dans d'autres situations. On a $\forall v \in V, v_d < v_f$, et le sommet $v$ est blanc avant $v_d$, est gris entre $v_d$ et $v_f$, puis est noir après.

Enfin, pour un sommet $v$, on note $v_\pi$ le *prédécesseur* de $v$ dans le parcours, c'est-à-dire le sommet $u$ à partir duquel on est arrivé sur $v$. Si $v$ n'a pas de prédécesseur car on a commencé la fonction dessus, on pose $v_\pi = \varnothing$.

Voici le pseudo-code de l'algorithme *dfs*, qui est un algorithme récursif :

```
1   DFS(G) :
2       Pour chaque sommet v ∈ V :
3           v_couleur ← Blanc
4           v_π ← ∅
5       t ← 0
6       Pour chaque sommet v ∈ V :
7           Si v_couleur = Blanc :
8               DFS_visite(G, v)

9   DFS_visite(G, v) :
10      incrémenter t
11      v_d ← t (v vient juste d'être découvert)
12      v_couleur = Gris
13      Pour chaque voisin Blanc w de v :
14          w_π ← v
15          DFS_visite(G, w)
16      v_couleur ← Noir (on a fini d'explorer les voisins de v)
17      incrémenter t
18      u_f ← t
```

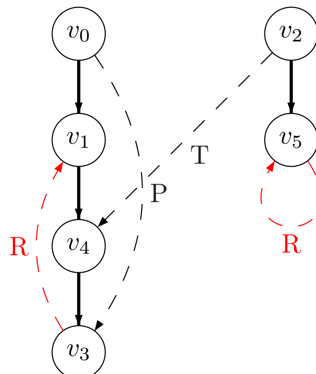
On voit qu'après exécution de l'algorithme, tous les sommets sont coloriés en noir.

Le résultat obtenu correspond alors à une forêt, dite *couvrante* du graphe, qui contient des arbres tels que les racines de ces arbres sont les sommets qui n'ont pas de prédécesseurs. On peut alors classer les arêtes du graphe en quatre catégories :

1. Les *arêtes des arbres* : une arête $(u, v)$ est dans un des arbres si $v$ a été découvert pour la première fois en explorant l'arête $(u, v)$.
2. Les *rétro-arêtes* : une arête $(u, v)$ est une rétro-arête si $v$ est ancêtre de $u$ dans l'un des arbres de la forêt couvrante.
3. Les *pro-arêtes* : une arête $(u, v)$ est une pro-arête si elle n'est pas une arête de l'arbre contenant $u$, mais connecte $u$ à un descendant $v$.
4. Les *arêtes-transversales* : ce sont les autres arêtes. Elles peuvent aller d'un arbre à un autre, ou bien relier deux sommets d'un même arbre tant que l'un n'est pas ancêtre de l'autre.

**Exemple 6.2.** On représente sur la Figure 16 l'exécution du *dfs* pour le graphe $G$ de l'exemple 5.2. Pour simplifier la lecture, les étiquettes des sommets et des arêtes ne sont pas représentées. Les arêtes épaisses correspondent aux arêtes des arbres couvrants, et celles en pointillés représentent les autres arêtes. La lettre à côté de l'arête correspond aux type d'arête (R pour rétro-arête, P pour pro-arête et T pour arête transversale).

La forêt couvrante obtenue est alors cet arbre :




L'algorithme *dfs* donne assez d'informations pour classifier les arêtes quand on les rencontre. Plus précisément, on a :

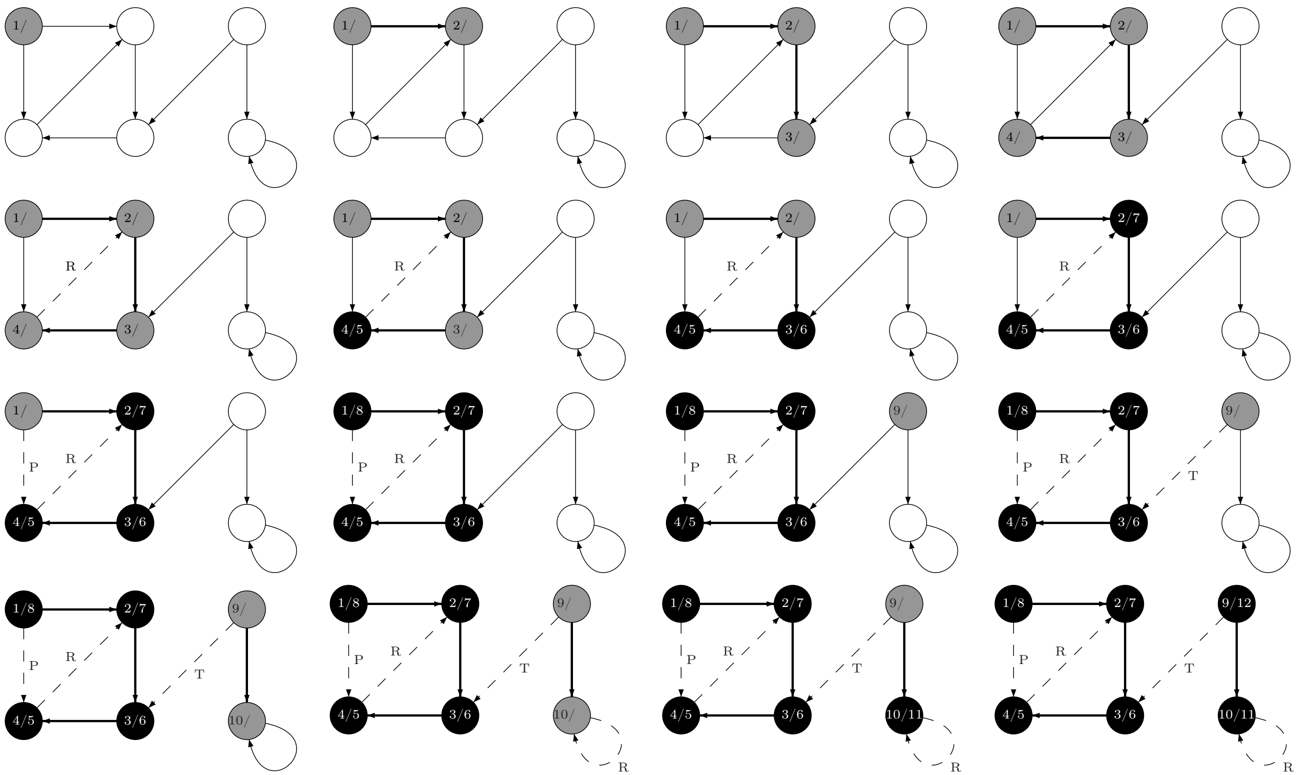



FIGURE 16 – Exécution du parcours en profondeur

**Proposition 6.3.** *Soit $G = (V, E)$ un graphe. Lorsque l'on explore une arête $(u, v) \in E$, la couleur de $v$ nous donne une indication de la catégorie de l'arête :*

- *Si $v$ est* blanc*, c'est une arête de l'arbre couvrant.*
- *Si $v$ est* gris*, c'est une rétro-arête.*
- *Si $v$ est* noir*, c'est une pro-arête ou bien une arête transversale.*

***Preuve.*** 1. est immédiat par la description de l'algorithme.

2. les sommets gris forment une chaîne de descendants correspondant à la pile d'exécution de la fonction récursive DFS_visite. Le nombre de sommets gris est un de plus que la profondeur [1] dans la forêt couvrante du sommet le plus récemment découvert. L'exploration continue ensuite par le sommet le plus profond, alors un sommet qui atteint un autre sommet gris a atteint un ancêtre : c'est bien une rétro-arête.

3. est le cas où les arêtes ne sont ni dans le cas 1. ni dans le cas 2. ■

À chaque rétro-arête, on peut associer un cycle dans le graphe $G$, que l'on créé en reliant les extrémités $u$ et $v$ de la rétro-arête par l'unique chemin dans la forêt couvrante du graphe reliant $u$ à $v$. Ainsi, en effectuant un parcourt en profondeur d'un graphe, il est possible de voir si il possède un cycle.

Dans la pratique, on n'effectuera pas le parcours entier dans tous les cas : dès que l'on trouve une rétro-arête, c'est qu'il y a un cycle, donc on s'arrête là ! Dans le cas de graphes réduits, pour savoir si le cycle passe par $\Delta$, il suffit que la rétro-arête arrive sur le sommet $\Delta$. En effet, dans l'implémentation, on fera en sorte que le sommet $\Delta$ soit le premier à être visité dans le *dfs*, il sera donc toujours une racine d'un des arbres de la forêt couvrante.

**Proposition 6.4** (complexité du *dfs*)**.** *Soit $G = (V, E)$ un graphe. On pose $n := |V|$ et $m := |E|$. Si l'on ne s'arrête par dès que l'on a trouvé un cycle, la complexité du parcours en profondeur est un $O(n + m)$.*

***Preuve.*** Reprenons le pseudo-code ci-dessus. Les lignes 2 à 4 et 6 à 8 sont en $O(n)$ car on visite tous les sommets, sans compter les appels de la fonction DFS_visite. Cette fonction est appelée exactement $n$ fois, car les sommets sur lesquels ont fait l'appel doivent être blancs et sont colorés en gris dès le début de l'appel (ligne 12). Pendant l'exécution de DFS_visite$(G, v)$, la boucle **pour** allant des lignes 13 à 15 est exécutée exactement $|\text{Adj}(v)|$, ce qui correspond au nombre de sommets atteignables depuis le sommet $v$ (c'est-à-dire son degré sortant). Par ailleurs, dans un graphe quelconque, on a $\sum_{v \in V} |\text{Adj}(v)| = m$. Le coût total des lignes 6 à 8 est donc un $O(m)$. Ainsi, la complexité est bien un $O(n + m)$. ■

1. la *profondeur* d'un sommet dans un arbre correspond au nombre d'arêtes qui le sépare de la racine

Le parcours en profondeur est donc linéaire en la taille du graphe. Dans notre cas, l'algorithme qui détermine si un automate unidimensionnel est surjectif est donc quadratique en la taille de la mémoire car la taille du graphe pair est comme le carré de la taille de la mémoire.

## 6.3 Résultats

Il est donc possible de tester si un automate cellulaire unidimensionnel donné est injectif, surjectif ou non, uniquement à partir de sa règle locale et de la taille $m$ de la mémoire. J'ai codé en langage OCaml des fonctions qui, à partir de la règle locale d'un automate cellulaire unidimensionnel, renvoient les réponses à ces questions. Le code est donné en Annexe A. Par exemple, en exécutant cette fonction pour les 256 automates élémentaires, on peut compter le nombre d'automates injectifs et le nombre d'automates surjectifs. On trouve qu'il y a 6 automates élémentaires injectifs (et donc réversibles) et 30 surjectifs. Il y en a donc 24 qui sont surjectifs et non-injectifs et 226 qui sont non-surjectifs (et donc non-injectifs). Les résultats sont donnés dans le tableau suivant :

| | Injectif | Non-injectif | $\sum$ |
|---|---|---|---|
| Surjectif | 6 | 24 | 30 |
| Non-surjectif | 0 | 226 | 226 |
| $\sum$ | 6 | 250 | 256 |

Ici, les 6 automates réversibles sont les automates « trivialement réversibles », c'est-à-dire ceux dont la mémoire peut être ramenée à un seul élément : on ne regarde l'état que d'une seule cellule pour aller à l'état suivant.

Prenons maintenant $m = 4$. Cette fois-ci, il y en a 8 qui sont trivialement réversibles. En exécutant la fonction pour les $2^{2^4} = 65\,536$ automates possibles, on voit qu'il y a d'autres automates réversibles. C'est le cas par exemple de l'automate 54 000 (en reprenant la notation de Wolfram). On peut synthétiser les résultats dans ce tableau :

| | Injectif | Non-injectif | $\sum$ |
|---|---|---|---|
| Surjectif | 16 | 566 | 582 |
| Non-surjectif | 0 | 64 594 | 64 594 |
| $\sum$ | 16 | 65 520 | 65 536 |

Prenons maintenant $m = 2$, $A = \{0, 1, 2\}$. On a ici un alphabet à trois éléments, ce qui fait donc $3^{3^2} = 19\,683$ automates cellulaires possibles. On effectuant les recherches, on trouve les résultats suivantes :

| | Injectif | Non-injectif | $\sum$ |
|---|---|---|---|
| Surjectif | 48 | 372 | 420 |
| Non-surjectif | 0 | 19 263 | 19 263 |
| $\sum$ | 48 | 19 635 | 19 683 |

Ici, il y a 12 automates dont la mémoire peut être réduite à un seul élément. Il y a donc d'autres automates cellulaires réversibles, comme par exemple la règle numéro 17 563 (en adaptant la notation de Wolfram en base 3). Cet automate a comme règle locale celle décrite dans le tableau ci-dessous. Enfin, on a représenté en Figure 17 le diagramme espace-temps de cet automate.

| $x(M)$ | 22 | 21 | 20 | 12 | 11 | 10 | 02 | 01 | 00 |
|---|---|---|---|---|---|---|---|---|---|
| $\mu_{17563}(x)$ | 2 | 2 | 0 | 0 | 0 | 2 | 1 | 1 | 1 |

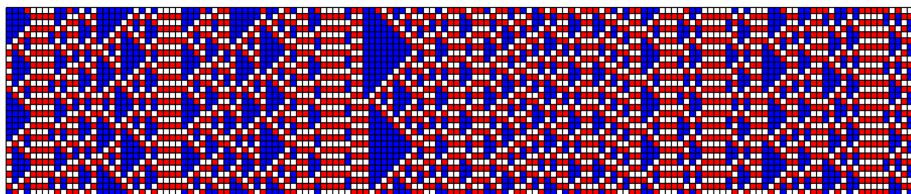


FIGURE 17 – Diagramme espace-temps de la règle 17 563 avec $m = 2$ et $A = \{0, 1, 2\}$

# A Code OCaml

Pour implémenter l'algorithme qui détermine si un automate cellulaire unidimensionnel est injectif ou surjectif, nous allons utiliser la structure de donnée `'a graphe` qui est définie ainsi :

```
type 'a graphe = {
    v : 'a array;
    e : (char * int) list array;
}
```

Le champ `g.v` est un tableau contenant les étiquettes de type `'a` des $n$ sommets du graphe et le champ `g.e` est un tableau de $n$ éléments contenant les listes d'adjacence des sommets : ainsi pour $i \in [\![0, n-1]\!]$, la case `g.e.(i)` contient la liste des couples `(c,arr)` où `arr` est le numéro du sommet d'arrivée et `c` est l'étiquette de l'arête (forcément de type `char`).

## A.1 Fonctions auxiliaires

Ces différentes fonctions servent dans l'exécution des fonctions plus compliquées. Leur code n'est pas donné dans ce document car elles sont assez simple à comprendre.

`val ajouter_dans_tableau : 'a array ref -> 'a -> unit`
L'appel `ajouter_dans_tableau refTab e` va modifier le contenu de `refTab` en y ajoutant, à la fin, l'élément `e`.

`val ajouter_dans_liste : 'a list -> 'a -> 'a list`
L'appel `ajouter_dans_liste l e` renvoie une nouvelle liste contenant `e` pointant vers la liste `l`.

`val ppi : 'a array -> 'a -> int`
L'appel `ppi tab e` renvoie le plus petit indice `i` tel que `tab.(i)=e`. Si l'élément `e` n'est pas dans le tableau, la fonction revoie `-1`.

`val produit_cartesien : 'a array -> 'b array -> ('a * 'b) array`
L'appel `produit_cartesien tabA tabB` renvoie le tableau des couples `(a,b)` pour `a` dans `tabA` et `b` dans `tabB`.

`val produit_cartesien_diagonal : 'a array -> ('a * 'a) array`
L'appel `produit_cartesien_diagonal t` renvoie le tableau des couples `(a,b)` pour `a` et `b` dans `t` de sorte que `a`$\neq$`b`.

`val est_dans_liste : 'a list -> 'a -> bool`
L'appel `est_dans_liste l e` renvoie `true` si `e` est dans la liste `l` et `false` sinon.

`val sont_dans_liste : 'a list -> 'a -> 'a -> bool`
L'appel `sont_dans_liste l e1 e2` renvoie `true` si `e1` et `e2` sont tous les deux dans la liste `l` et `false` sinon.

`val nombre_sommet : 'a graphe -> int`
Renvoie le nombre de sommets du graphe donné en entrée.

## A.2 Graphes de de Bruijn

Les graphes de de Bruijn sont obtenus à partir de deux paramètres : la taille $m$ du graphe et l'alphabet $A$. On utilise ensuite une fonction $\mu$ qui correspond à la règle locale pour obtenir la représentation de de Bruijn d'un automate cellulaire. Les règles locales seront toujours de signature `string -> char`. C'est pour cela que les graphes de de Bruijn sont toujours de type `string graphe`, tout comme les représentations de de Bruijn.

`val nouveau_mot : string -> char -> string`
L'appel `nouveau_mot mot l` renvoie l'étiquette d'arrivée du sommet `mot` d'étiquette `l` dans le graphe de de Bruijn.

`val etiquette_arete : string -> char -> string`
L'appel `etiquette_arete mot l` renvoie le caractère `l` concaténé à la chaîne `mot`.

```
let nouveau_mot mot l = (String.sub mot 1 (String.length mot - 1)) ^ (Char.
    escaped l)
let etiquette_arete mot l = mot ^ (Char.escaped l)
```

### A.2.1 Graphe de de Bruijn

`val sommets : int -> char array -> string array`
L'appel `sommets m alp` renvoie le tableau formé de tous les mots de longueur `m` sur l'alphabet `alp`.

`val ajouter_arete : ('a * 'b) list array -> int -> 'a -> 'b -> unit`
L'appel `ajouter_arete es dep eti arr` ajoute l'arête allant du sommet `dep` au sommet `arr` d'étiquette `i` dans le tableau `es` des listes d'adjacence du graphe.

`val aretes : char array -> string array -> (char * int) list array`
L'appel `aretes alp vs` renvoie le tableau d'adjacence du graphe de de Bruijn sur les sommets de `vs` avec l'alphabet `alp`.

`val creer_graphe_de_de_bruijn : int -> char array -> string graphe`
L'appel `creer_graphe_de_de_bruijn m a` renvoie le graphe de de Bruijn de taille `m` (les mots sont de taille $m-1$) sur l'alphabet `a`.

```
let rec sommets m alp =
  if m = 0 then [||]
  else if m = 1 then Array.map Char.escaped alp
  else (
    let vs = sommets (m-1) alp in
    let n = Array.length vs in
    let vs' = ref [||] in
    for i = 0 to n-1 do
      for j = 0 to (Array.length alp)-1 do
        ajouter_dans_tableau vs' ((Char.escaped alp.(j))^vs.(i));
      done
    done;
    !vs'
  )

let ajouter_arete es dep eti arr =
  es.(dep) <- ajouter_dans_liste es.(dep) (eti,arr)

let aretes alp vs =
  let n = Array.length vs and
      m = Array.length alp in
  let es = Array.make n [] in
  for i = 0 to n-1 do
    for a = 0 to m-1 do
      let mot = nouveau_mot vs.(i) alp.(a) in
      let ind = ppi vs mot in
      ajouter_arete es i alp.(a) ind;
    done
  done;
  es

let creer_graphe_de_de_bruijn m a =
  let vs = sommets (m-1) a in
  let es = aretes a vs in
  {v= vs; e = es}
```

Lignes 32 à 35 : fonction principale qui crée le graphe de de Bruijn de taille `m` sur l'alphabet `a`. Les mots, qui seront étiquettes des sommets de ce graphe, auront une longueur `m-1`. La fonction utilise les fonctions `sommets` et `aretes` pour remplir convenablement les champs du graphe.

Lignes 1 à 14 : fonction récursive qui renvoie un tableau contenant tous les mots de taille `m` sur l'alphabet `a`. Pour les construire, la fonction va, à partir des mots de taille inférieure, concaténer tous les caractères de `a` à tous ces mots.

Lignes 19 à 30 : fonction qui va créer le tableau d'adjacence du graphe de de Bruijn et le remplir avec les bonnes arêtes dans les bonnes cases, en utilisant la fonction `ajouter_arete` définie lignes 16 et 17, qui ne fait qu'utiliser une fonction auxiliaire.

### A.2.2 Représentation de de Bruijn d'un automate cellulaire

Pour avoir la représentation de de Bruijn d'un automate cellulaire, il suffit de prendre le graphe de de Bruijn de la taille adéquate et d'appliquer à toutes les arêtes la règle locale $\mu$.

`val aretes_automate : (char * 'a) list -> (string -> 'b) -> string -> ('b * 'a) list`

L'appel `aretes_automate l mu dep` renvoie la liste des arêtes du graphe de de Bruijn de l'automate partant du sommet `dep` par la fonction `mu`. La liste `l` contient les arêtes du graphe de de Bruijn initial.

`val creer_representation_automate : int -> char array -> (string -> char) -> string graphe`

L'appel `creer_representation_automate m a mu` renvoie la représentation de de Bruijn de l'automate cellulaire unidimensionnel de taille `m` sur l'alphabet `a` avec `mu` comme règle locale.

```
1  let rec aretes_automate l mu dep =
2    match l with
3    | [] -> []
4    | (c,arr)::q -> (mu (etiquette_arete dep c), arr)::(aretes_automate q mu
       dep)
5
6  let creer_representation_automate m a mu =
7    let g = creer_graphe_de_de_bruijn m a in
8    let n = Array.length g.v in
9    let es = Array.make n [] in
10   for i = 0 to n-1 do
11     es.(i) <- aretes_automate g.e.(i) mu g.v.(i);
12   done;
13   {v = g.v; e = es}
```

Lignes 6 à 13 : fonction principale qui crée la représentation de de Bruijn sur l'alphabet `a` de l'automate cellulaire dont on a donné la règle locale `mu` et la taille de la mémoire `m`. Elle commence par créer le graphe de de Bruijn de la bonne taille puis parcours les listes d'adjacence via la fonction auxiliaire récursive `aretes_automate` pour y appliquer `mu`, en utilisant une fonction auxiliaire.

### A.2.3 Graphe pair

Chaque sommet du graphe pair correspond à un couple de sommets du graphe initial. Tous les graphes pairs seront donc de type `(string * string) graphe`. On notera que la fonction `creer_graphe_pair_automate` va renvoyer un couple de type `(string * string) graphe * string array`. Cela facilitera l'exécution de la fonction de réduction de ce graphe pair dans la suite.

`val sont_aretes : int -> 'a -> int -> 'a -> 'b -> ('b * 'a) list array -> bool`

L'appel `sont_aretes dep1 arr1 dep2 arr2 a es` renvoie `true` si et seulement si les arêtes `(dep1,a,arr1)` [2] et `(dep2,a,arr2)` sont toutes les deux des arêtes de `es`.

`val aretes_paires : ('a * 'a) array -> 'b array -> ('b * int) list array -> 'a array -> ('b * int) list array`

L'appel `aretes_paires vs esinit alp vsinit` renvoie le tableau des listes d'adjacence contenant toutes les arêtes du graphe pair de sommets `vs` sur l'alphabet `alp` à partir des sommets `vsinit` et des arêtes `esinit`.

2. cela correspond à l'arête d'étiquette `a` partant du sommet `dep1` et allant au sommet `arr1`

`val creer_graphe_pair_automate : int -> char array -> (string -> char) -> (string * string) graphe * string array`

L'appel `creer_graphe_pair_automate m a mu` renvoie le graphe pair associé à l'automate de taille `m` sur l'alphabet `alp` dont la règle locale est `mu`, ainsi que le tableau contenant les sommets de la représentation de de Bruijn de l'automate.

```
1  let sont_aretes dep1 arr1 dep2 arr2 alp es =
2    if dep1 = dep2 then
3      sont_dans_liste es.(dep1) (alp,arr1) (alp,arr2)
4    else
5      (est_dans_liste es.(dep1) (alp,arr1))&&(est_dans_liste es.(dep2) (alp,
          arr2))
6
7  let aretes_paires vs alp esinit vsinit =
8    let n = Array.length vs in
9    let edges = Array.make n [] in
10   for i = 0 to n-1 do
11     for j = 0 to n-1 do
12       match vs.(i),vs.(j) with (u1,u2),(v1,v2) ->
13         let dep1 = ppi vsinit u1 and
14             dep2 = ppi vsinit u2 and
15             arr1 = ppi vsinit v1 and
16             arr2 = ppi vsinit v2 in
17         for a = 0 to (Array.length alp)-1 do
18           if sont_aretes dep1 arr1 dep2 arr2 alp.(a) esinit then
19             edges.(i) <- ajouter_dans_liste edges.(i) (alp.(a),j);
                 (* l'arete va de (u1,u2) vers (v1,v2) *)
20       done
21     done
22   done;
23   edges
24
25
26 let creer_graphe_pair_automate m alp mu =
27   let g = creer_representation_automate m alp mu in
28   let vs = produit_cartesien g.v g.v in
29   let es = aretes_paires vs alp g.e g.v in
30   {v = vs; e = es}, g.v
```

Lignes 26 à 30 : fonction principale qui crée le graphe pair associé à l'automate de taille `m` sur l'alphabet `alp` dont la règle locale est `mu`. Elle commence par créer la représentation de de Bruijn de l'automate, puis prend le produit cartésien des sommets de ce graphe pour enfin remplir le tableau d'adjacence, via la fonction `aretes_paires`.

Lignes 7 à 23 : fonction créant le tableau des listes d'adjacences, suivant la définition du graphe pair. Pour chaque couple de sommets du graphe pair $(i, j) = ((u_1, v_1), (u_2, v_2))$ et pour chaque lettre $a \in$ `alp`, elle va appeler la fonction `sont_aretes` pour voir si les arêtes $(u_1, a, v_1)$ et $(u_2, a, v_2)$ sont dans le graphe `g`. Les appels de la fonction `ppi` permettent de savoir à quels sommets du graphe initial `g` correspondent les sommets `u1`, `u2`, `v1`, `v2`.

Lignes 1 à 5 : Si les deux sommets de départ `dep1` et `dep2` sont les mêmes, il est inutile de parcourir deux fois la même liste (qui correspond à la liste d'adjacence dans le graphe initial), c'est pour cela que dans ce cas, on fait appel à la fonction `sont_dans_liste`. Dans l'autre cas, on est obligé de parcourir deux listes différentes, d'où les deux appels de la fonction `est_dans_liste`.

### A.2.4 Réduire le graphe pair

Cet algorithme de réduction va, à partir d'un graphe pair donné et de l'ensemble de sommets initiaux, renvoyer le graphe réduit, où tous les sommets diagonaux sont réunis dans un unique sommet $\Delta$. Comme on traite ici les graphes pairs obtenus à partir de représentations de de Bruijn d'automates cellulaires, les graphes à réduire sont toujours de type `(string * string) graphe`.

`val reduire_graphe : (string * string) graphe -> string array -> (string * string) graphe`

Fonction exécutant la réduction du graphe donné en entrée.

`val reduire_aretes : ('a * 'a) graphe -> ('a * 'a) array -> (char * int) list array`

Renvoie le tableau des listes correspondant aux sommets dans le graphe pair. Utilise une fonction auxiliaire récursive `aretes_reduites_aux : ('a * 'a) graphe -> ('a * 'a) array -> ('b * int) list -> bool -> ('b * int) list`.

```
1  let rec aretes_reduites_aux g vs es crt =
2    match es with
3    | [] -> []
4    | (a,arr)::es' -> if ((fst g.v.(arr))=(snd g.v.(arr)) && not crt) then
5                        (a,0)::(aretes_reduites_aux g vs es' true)
6                      else (
7                        let sArr = ppi vs g.v.(arr) in
8                        if not (sArr = -1) then
9                          (a,sArr)::(aretes_reduites_aux g vs es' crt)
10                       else (aretes_reduites_aux g vs es' crt)
11                     )
12
13 let reduire_aretes g vs =
14   let es = Array.make (Array.length vs) [] in
15   let nb = nombre_sommets g in
16   for i = 0 to nb-1 do
17     if (fst g.v.(i)) = (snd g.v.(i)) then (* sommet diagonal *)
18       es.(0) <- es.(0) @ (aretes_reduites_aux g vs g.e.(i) true)
19     else (
20       let ind = ppi vs g.v.(i) in
21       if not (ind = -1) then
22         es.(ind) <- aretes_reduites_aux g vs g.e.(i) false
23     )
24   done;
25   es
26
27
28 let reduire_graphe g sinit =
29   let vs = Array.append [|"Delta",""|] (produit_cartesien_diag sinit) in
30   let es = reduire_aretes g vs in
31   {v = vs; e = es}
```

Lignes 28 à 31 : fonction principale de réduction d'un graphe `g`. Le sommet d'étiquette `("Delta","")` correspond au sommet diagonal et est à la position `0` dans le tableau des sommets du graphe réduit. Les sommets suivants sont obtenus en faisant un produit cartésien sur les sommets initiaux `sinit` du graphe dans lequel les sommets diagonaux n'apparaissent pas (fonction `produit_cartesien_diagonal`).

Lignes 13 à 25 : cette fonction commence par créer le tableau des listes d'adjacence, initialement vide, et va le remplir au fur et à mesure en parcourant les sommets du graphe pair `g`. Si le sommet de `g` est diagonal, la fonction concatène, via l'opérateur `@`, la liste des arêtes de ce sommet à celle des arêtes de sommets diagonaux déjà traités. Dans l'autre cas, elle recherche à quel sommet du graphe réduit ce sommet correspond via la fonction `ppi` (nb : cet indice ne vaut jamais `0` car cela correspond à la position du sommet diagonal) et affecte directement la liste à la case du tableau d'adjacence. Cette liste est obtenue par la fonction auxiliaire `aretes_reduites_aux`.

Lignes 1 à 11 : fonction auxiliaire récursive qui parcourt la liste des sommets du graphe `g` et renvoie la liste de ces arêtes pour le graphe réduit. Le booléen `crt` permet de vérifier si on a déjà inséré le sommet diagonal dans la liste pour éviter de l'insérer plusieurs fois (c'est aussi pour cela que l'on appelle cette fonction directement avec `crt = true` dans la fonction précédente dans le cas où on est sur un sommet diagonal). De la même manière que dans la fonction précédente, la fonction va recherche le bon sommet correspondant dans le graphe réduit.

## A.3 Injectivité et surjectivité

Pour répondre à la question de l'injectivité et de la surjectivité de l'automate cellulaire, il faut déterminer si le graphe pair possède ou non des cycles. En effectuant un parcours en profondeur (*dfs*), on va pouvoir répondre à cette question. Pour l'algorithme, on définit le type `couleur` qui nous permettra de dire si l'on est déjà passé par un sommet durant l'exécution de la fonction.

`type couleur = Blanc | Gris | Noir`

`val voisin_blanc_ou_gris : ('a * int) list -> couleur array -> bool ref -> int`

L'appel `voisin_blanc_ou_gris vs couleurs crt` renvoie l'indice du premier sommet `Blanc` dans la liste `vs`. Si aucun sommet n'est `Blanc`, la fonction renvoie `-1`. Si un des sommets de la liste `vs` est `Gris`, le booléen `!crt` prend comme valeur `true` et la fonction renvoie `-1` (pour la recherche de rétro-arête dans l'exécution du *dfs*).

`val voisin_blanc_ou_gris_s0 : ('a * int) list -> couleur array -> bool ref -> bool ref -> int -> int`

L'appel `voisin_blanc_ou_gris_s0 vs couleurs crt crt0 s0` renvoie l'indice du premier sommet `Blanc` dans la liste `vs`. Si aucun sommet n'est `Blanc`, la fonction renvoie `-1`. Si un des sommets de la liste `vs` est `Gris` et ce sommet vaut `s0`, les booléens `!crt0` et `!crt` prennent comme valeur `true` et la fonction renvoie `-1` (pour la recherche de rétro-arête dans l'exécution du *dfs*). Si un sommet `Gris` qui n'est pas `s0` est trouvé, alors le booléen `!crt` devient `false` (car on a trouvé une rétro-arête).

`val contient_retro_arete : 'a graphe -> bool`

L'appel `contient_retro_arete g` renvoie `true` si et seulement si le graphe `g` contient une rétro-arête. Cette fonction utilise une fonction auxiliaire récursive `dfs_visite : 'a graphe -> couleur array -> bool ref -> int -> unit`.

`val contient_retro_arete_delta : 'a graphe -> bool * bool`

L'appel `contient_retro_arete_delta g` renvoie un couple de booléens `(b0,b)` où `b0` vaut `true` si et seulement si le graphe `g` possède une rétro-arête qui passe par le sommet `0` et `b` vaut `true` si et seulement si `g` possède une rétro-arête quelconque (si `b0 = true`, alors `b = true`). Cette fonction utilise une fonction auxiliaire récursive `dfs_visite_s0 : 'a graphe -> couleur array -> bool ref -> bool ref -> int -> unit`.

`val est_injectif : int -> char array -> (string -> char) -> bool`

L'appel `est_injectif m a mu` renvoie `true` si et seulement si l'automate cellulaire sur l'alphabet `a` de mémoire de taille `m` avec `mu` comme règle locale est injectif.

`val est_surjectif : int -> char array -> (string -> char) -> bool`

L'appel `est_surjectif m a mu` renvoie `true` si et seulement si l'automate cellulaire sur l'alphabet `a` de mémoire de taille `m` avec `mu` comme règle locale est surjectif.

`val est_reversible : int -> char array -> (string -> char) -> bool * bool`

L'appel `est_reversible m a mu` renvoie un couple de booléens `(b1, b2)` tels que `b1 = true` si et seulement si l'automate cellulaire sur l'alphabet `a` de mémoire de taille `m` avec `mu` comme règle locale est surjectif et `b2 = true` si cet automate est injectif.

---

```
1  let rec voisin_blanc_ou_gris vs couleurs crt =
2    match vs with
3    | [] -> -1
4    | (_,t)::vs' -> match couleurs.(t) with
5              | Blanc -> t
6              | Gris -> crt := true;-1
7              | _ -> voisin_blanc_ou_gris vs' couleurs crt
8
9  let rec dfs_visite g couleurs crt s =
10   couleurs.(s) <- Gris;
11   let v = ref (voisin_blanc_ou_gris g.e.(s) couleurs crt) in
12   while !v <> -1 do
13     dfs_visite g couleurs crt !v;
14     v := (voisin_blanc_ou_gris g.e.(s) couleurs crt)
15   done;
16   couleurs.(s) <- Noir
17
18 let contient_retro_arete g =
19   let n = nombre_sommets g in
20   let couleurs = Array.make n Blanc in
```

```
21   let s = ref 0 and
22       ret = ref false in
23   while (not !ret && !s <n) do
24     if couleurs.(!s) = Blanc then
25       dfs_visite g couleurs ret !s;
26     incr s;
27   done;
28   !ret
```

Lignes 18 à 28 : cette fonction initialise le parcours en profondeur (cf section 6.2) du graphe puis appelle la fonction auxiliaire `dfs_visite` pour visiter les sommets. Tant que l'on n'a pas trouvé de rétro-arête, c'est-à-dire tant que le booléen `!ret` vaut `false`, on continue à parcourir les sommets du graphe. Au sortir de la fonction, on renvoie ce booléen.

Lignes 9 à 16 : cette fonction exécute la visite du graphe en le parcourant en profondeur. Elle commence par colorer le sommet `s` en `Gris`. Via la fonction `voisin_blanc_ou_gris`, elle parcourt la liste d'adjacence de `s`. Si l'on trouve un sommet `Blanc`, alors on rappelle la fonction à partir de ce sommet. Si l'on trouve un sommet `Gris`, alors on a trouvé une rétro-arête et on s'arrête là ! Enfin, le sommet `s` est colorié en `Noir`.

Lignes 1 à 7 : la fonction parcourt récursivement la liste `vs` à la recherche d'un sommet `Blanc` ou `Gris`. Si un sommet `Blanc` est trouvé (ligne 5), alors le numéro de ce sommet est renvoyé. Si un sommet `Gris` est trouvé (ligne 6), alors une rétro-arête est trouvée et le booléen `!crt` prend `true` comme valeur. `-1` est renvoyé pour sortir de la boucle `while` de la fonction `dfs_visite`, car on a trouvé une rétro-arête : il n'est donc plus nécessaire de continuer le parcours du graphe.

```
1  let rec voisin_blanc_ou_gris_s0 vs couleurs crt crt0 s0 =
2    match vs with
3    | [] -> -1
4    | (_,t)::vs' -> match couleurs.(t) with
5                | Blanc -> t
6                | Gris when t = s0 -> crt := true; crt0 := true; -1
7                | Gris when t <> s0 -> crt :=  true; voisin_blanc_ou_gris_s0 vs'
                   couleurs crt crt0 s0
8                | _ -> voisin_blanc_ou_gris_s0 vs' couleurs crt crt0 s0
9
10 let rec dfs_visite_s0 g couleurs crt crt0 s =
11   couleurs.(s) <- Gris;
12   let v = ref (voisin_blanc_ou_gris_s0 g.e.(s) couleurs crt crt0 0) in
13   while !v <> -1 do
14     dfs_visite_s0 g couleurs crt crt0 !v;
15     v := (voisin_blanc_ou_gris_s0 g.e.(s) couleurs crt crt0 0)
16   done;
17   couleurs.(s) <- Noir
18
19 let contient_retro_arete_delta g =  (* le sommet diagonal est le sommet 0 *)
20   let n = nombre_sommets g in
21   let couleurs = Array.make n Blanc in
22   let ret0 = ref false and
23       ret = ref false in
24   dfs_visite_s0 g couleurs ret ret0 0;
25   let s = ref 1 in
26   while (not !ret && !s <n) do
27     if couleurs.(!s) = Blanc then
28       dfs_visite g couleurs ret !s;
29     incr s;
30   done;
31   !ret0, !ret
```

Lignes 19 à 31 : le principe de fonctionnement de cette fonction est le même que celui de la fonction `contient_retro_arete`. La différence est qu'ici, on commence par parcourir le sommet diagonal $\Delta$ (qui correspond au sommet `0` du graphe réduit `g`). Le booléen `!ret0` donne une indication si une rétro-arête arrive sur le sommet `0`. Le booléen `!ret` dit si il y a une rétro-arête quelconque dans le graphe. Ensuite, la fonction termine le parcours en profondeur sur les autres sommets du graphe.

Lignes 1 à 8 : sur le même principe que la fonction `voisin_blanc_ou_gris`, cette fonction parcourt la liste d'adjacence et renvoie la valeur du premier sommet `Blanc` de la liste. Dans le même temps, elle modifie les booléens `!crt0` et `!crt` de sorte que :

— `!crt0` vaut `true` si une rétro-arête arrivant sur le sommet `s0` est trouvée. Dans ce cas, la fonction renvoie `-1` pour sortir de la boucle `while` dans la fonction `dfs_visite`.

— `!crt` vaut `true` si une rétro-arête quelconque est trouvée. Dans ce cas, on continue à parcourir la liste `vs` pour trouver éventuellement une rétro-arête arrivant sur `s0`.

Dans la pratique, comme on veut savoir si la rétro-arête arrive sur $\Delta$, on appellera toujours cette fonction en prenant `s0 = 0`.

```
1  let est_injectif m a mu =
2    let g,vs = creer_graphe_pair_automate m a mu in
3    let gr = reduire_graphe g vs in
4    not (contient_retro_arete gr)
5
6  let est_surjectif m a mu =
7    let g,vs = creer_graphe_pair_automate m a mu in
8    let gr = reduire_graphe g vs in
9    let bol,_ = contient_retro_arete_delta gr in
10   not bol
11
12 let est_reversible m a mu =
13   let g,vs = creer_graphe_pair_automate m a mu in
14   let gr = reduire_graphe g vs in
15   let surj,inj = contient_retro_arete_delta gr in
16   not surj, not inj
```

Ces trois fonctions créent le graphe pair de l'automate, puis le réduisent et enfin utilisent les fonctions précédentes pour rechercher si il y a des cycles passant ou non par le sommet diagonal. Elles déterminent donc si l'automate dont la règle locale est donnée en entrée est injectif ou surjectif, en utilisant les résultats du Théorème 5.10.

# B Langage des orphelins

Les représentations de de Bruijn d'un automate cellulaire unidimensionnel non-surjectif permettent de déterminer l'ensemble des orphelins, c'est-à-dire des motifs qui n'ont pas de pré-image par cet automate cellulaire. Cet algorithme utilise la théorie des automates finis [10]. Nous allons d'abord voir quelques généralités de cette théorie pour ensuite étudier cet algorithme, qui utilise la méthode des sous-ensembles.

## B.1 Généralités sur les automates finis

**Définition B.1.** On définit un *automate fini non déterministe* (abrégé en AFN) comme étant un quintuplet $\mathcal{A} = \langle Q, A, E, I, T\rangle$ où :

- $Q$ est un ensemble fini, l'ensemble des *états*.
- $A$ est un ensemble quelconque, l'*alphabet*.
- $I$ est un sous-ensemble de $Q$, l'ensemble des *états initiaux*.
- $T$ est un sous-ensemble de $Q$, l'ensemble des *états finals* (ou *accepteurs*).
- $E$ est un sous-ensemble de $Q \times A \times Q$, l'ensemble des *transitions*.

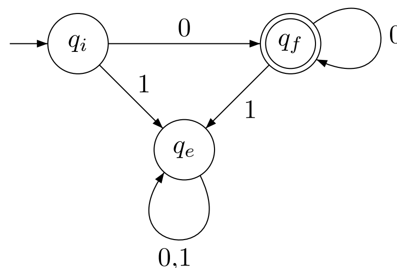
Une transition est un triplet $(p, a, q) \in Q \times A \times Q$, notée $p \xrightarrow{a} q$. La lettre $a$ est l'étiquette de la transition, $p$ est l'état d'origine et $q$ est l'extrémité de la transition.

On représente un automate fini sous la forme d'un graphe orienté étiqueté. Les sommets du graphe sont les états de l'ensemble $Q$. Si $(p, a, q) \in E$, on trace une arête de $p$ vers $q$, d'étiquette $a$. Enfin, pour représenter les états initiaux, on fait pointer une flèche de l'extérieur du graphe vers ces états. De même, un état final sera doublement entouré (voir l'Exemple B.2 ci-dessous).

Une suite finie d'éléments de $A$ est appelée un *mot*, et le nombre d'éléments de cette suite en est sa longueur. On définit aussi un mot de longueur nulle : $\varepsilon$. Enfin, $A^* = \bigcup_{k=0}^{+\infty} A^k$, est la réunion sur l'entier $k$ de tous les mots de longueur $k$ (parfois, cette réunion est appelée « somme » $A^* = \sum_{k=0}^{+\infty} A^k$). C'est l'ensemble de tous les mots sur l'alphabet $A$.

Un sous-ensemble $L \subset A^*$ est appelé *langage*. On dit qu'un mot $w \in A^*$ est reconnu par un automate fini $\mathcal{A}$ si il existe un chemin étiqueté par $w$ qui commence sur un état initial et se termine sur un état final. Pour un automate fini $\mathcal{A}$ , on note $\mathcal{L}(\mathcal{A})$ le langage des mots reconnus par $\mathcal{A}$. On dit que deux automates finis $\mathcal{A}$ et $\mathcal{A}'$ sont équivalents s'ils reconnaissent le même langage.

**Exemple B.2.** Voici un exemple d'automate sur l'alphabet $A = \{0, 1\}$. On pose $Q = \{q_i, q_f, q_e\}$, $I = \{q_i\}$, $T = \{q_f\}$ et $E = \{q_i \xrightarrow{0} q_f, q_f \xrightarrow{1} q_e, q_i \xrightarrow{1} q_e, q_e \xrightarrow{0} q_e, q_e \xrightarrow{1} q_e, q_f \xrightarrow{0} q_f\}$.




On a $\mathcal{L}(\mathcal{A}) = \{0, 00, 000, \cdots\}$, puisque dès que l'on a la lettre 1 dans le mot, on se retrouve dans l'état $q_e$ que l'on ne peut pas quitter. Cet automate reconnaît donc les mots non-vides ne contenant pas la lettre 1.

## B.2 AFD, AFDC et méthode des sous-ensembles

**Définition B.3** (AFD,AFDC)**.** Soit $\mathcal{A} = \langle Q, A, E, I, T\rangle$ un automate fini. On dit que $\mathcal{A}$ est *déterministe* (AFD) lorsque :

- L'ensemble des sommets initiaux $I$ ne contient qu'un seul élément.
- Pour tout état $q \in Q$ et pour toute lettre $a \in A$, il existe au plus un état $q' \in Q$ tel que $(q, a, q') \in E$.

Si l'automate vérifie $\forall q \in Q, \forall a \in A$, il existe un unique état $q' \in Q$ tel que $(q, a, q') \in E$, on dit que $\mathcal{A}$ est *déterministe complet* (AFDC).

L'automate fini donné dans l'Exemple B.2 est un AFDC.

Dans un AFD, il n'y a qu'un seul chemin possible depuis chaque état avec chaque lettre. Il est clair que pour tout mot $w \in A^*$, il y a au plus un chemin qui commence à l'état initial (et exactement un si l'automate est complet). Le mot $w$ est accepté si et seulement si le dernier état de ce chemin est un état final.

Soit $\mathcal{A}$ un AFD. Alors $\mathcal{A}$ est équivalent à un AFDC. En effet, il suffit de créer un nouvel état « évier », non-final, vers lequel on ajoute toutes les flèches manquantes pour satisfaire la définition d'AFDC. Ce nouvel automate construit reconnaît le même langage que $\mathcal{A}$.

Soit $\mathcal{A} = \langle Q, A, E, I, T \rangle$ un automate fini. Nous allons voir qu'il est possible de le déterminiser, c'est-à-dire de créer un AFDC équivalent. Cette méthode s'appelle la construction par sous-ensembles. Le nouvel automate est $\mathcal{A}' = \langle Q', A, E', I', T' \rangle$, où :

- $Q' = \mathcal{P}(Q)$ (l'ensemble des parties de $Q$). $\mathcal{A}'$ possède donc $2^{|Q|}$ états,
- $I' = \{I\}$,
- $T' = \{P \in Q' \mid P \cap T \neq \varnothing\}$,
- Pour toute partie $X$ de $Q$ et toute lettre $a \in A$, la transition de $X$ avec la lettre $a$ va vers l'état $\{q \in Q \mid \exists x \in X \text{ tel que } x \xrightarrow{a} q\}$.

On voit que dans $\mathcal{A}'$, le dernier état d'un chemin qui commence au sommet $X \in \mathcal{P}(Q)$ et étiqueté par le mot $w \in A^*$ consiste en tous les états $q \in Q$ tels qu'il existe un chemin depuis un élément de $X$ vers $q$ étiqueté par $w$. Ainsi $\mathcal{A}$ et $\mathcal{A}'$ reconnaissent le même langage.

Dans la pratique, on ne garde pas les $2^{|Q|}$ états dans l'automate déterminisé car certains états ne sont pas atteints. On construit $\mathcal{A}'$ au fur et à mesure : on part de l'unique état initial $I' \in \mathcal{P}(Q)$, puis on ajoute au fur et à mesure les états quand ils sont atteints (voir dans l'Exemple B.4 ci-dessous).

Enfin, si un AFDC $\mathcal{A} = \langle Q, A, E, I, T \rangle$ sur un alphabet $A$ reconnaît un langage $L$, alors il est possible de créer un automate fini sur ce même alphabet $A$ qui reconnaisse le complémentaire $A^* \setminus L$. Il suffit de considérer l'automate $\mathcal{A}' = \langle Q, A, E, I, Q \setminus T \rangle$, où les états finals de $\mathcal{A}$ deviennent non-finals dans $\mathcal{A}'$ et inversement. Cet automate est dit *complémentaire* de $\mathcal{A}$.

## B.3 Retour aux automates cellulaires

Soit $A$ un ensemble fini. Soit $\tau \in \mathrm{CA}(\mathbb{Z}, A)$ un automate cellulaire unidimensionnel et soit $G_\tau = (V, E)$ sa représentation de de Bruijn. Soit l'automate $\mathcal{A}_\tau = \langle V, A, E, V, V \rangle$, un automate où les sommets du graphe $G_\tau$ sont les états, tous initiaux et finals. On obtient alors un AFN qui accepte tous les mots qui ne sont par orphelins (on a $\mathcal{L}(\mathcal{A}) = \tau({}^\omega A^\omega)$, où ${}^\omega A^\omega$ désigne l'ensemble de tous les mots bi-infinis sur l'alphabet $A$). On peut alors déterminiser cet automate et passer au complémentaire pour avoir un AFDC qui reconnaisse le langage des mots orphelins par cet automate cellulaire.

**Exemple B.4.** Reprenons l'exemple de la règle 110 (voir p.19). On a représenté Figure 18 l'automate obtenu à partir de la représentation de de Bruijn de l'automate cellulaire $\tau_{110}$. Tous les états de cet AFN sont initiaux et accepteurs.

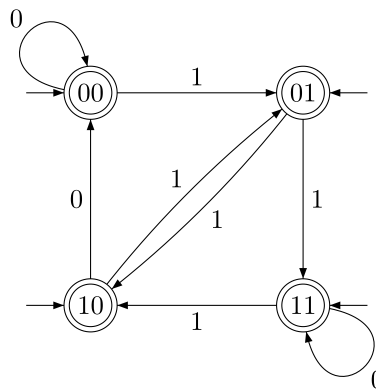



FIGURE 18 – Automate $\mathcal{A}_{110}$

Pour déterminiser cet automate, on part de l'état initial $\{00, 01, 10, 11\}$. Avec la lettre 0, on peut atteindre l'état $\{00, 11\}$ (car $00 \xrightarrow{0} 00$ et $11 \xrightarrow{0} 11$ sont des transitions de $\mathcal{A}_{110}$) et avec la lettre 1, on peut atteindre $\{01, 10, 11\}$. Puis on recommence à partir de ces états jusqu'à ce que l'on ait tout fait (au plus $2^4 = 16$ états possibles). Le résultat est un AFDC qui reconnaît tous les mots qui ne sont pas orphelins par l'automate $\tau_{110}$. La représentation du complémentaire de cet automate est donnée en Figure 19. L'état $\{00, 01, 10, 11\}$ est l'état initial et $\varnothing$ est le seul état final.

Avec cet automate fini, on a le langage des motifs qui sont orphelins par l'automate cellulaire $\tau_{110}$. Ainsi, on voit que le plus petit motif orphelin pour la règle est 01010 (qui peut être déterminé en effectuant un parcours en largeur du graphe). On observe aussi que seul les mots contenant les motifs 010 sont acceptés par cet AFDC, ce qui nous permet de déduire que toute configuration qui ne possède pas de 1 isolé a une pré-image par $\tau_{110}$.

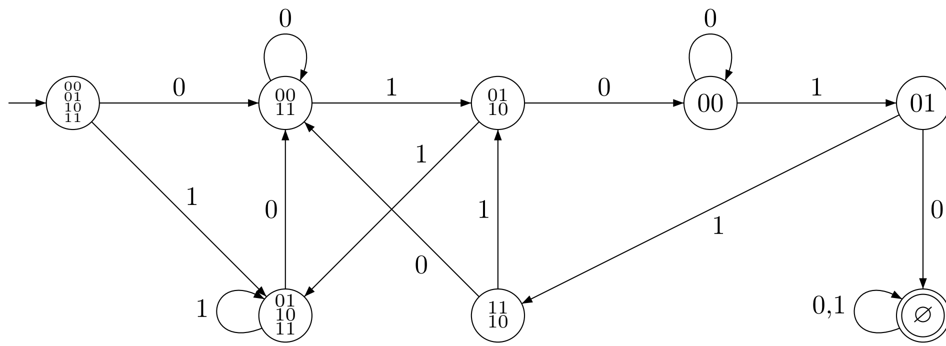



Figure 19 – AFDC $\mathcal{A}'_{110}$

# Références

[1] Tullio CECCHERINI-SILBERSTEIN et Michel COORNAERT : *Cellular Automata and Groups*, chapitre 1 : Cellular Automata. Springer, 2010.

[2] Jarkko KARI : Cellular automata, 2016.

[3] S. AMOROSO et Y.N. PATT : Decision Procedures for Surjectivity and Injectivity of Parallel Maps for Tesselation Structures. *Journal of Computers and System Science*, 6:448–464, 1972.

[4] Klaus SUTNER : De Bruijn Graph and Linear Cellular Automata. *Complex Systems*, 5 (1):19–30, 1991.

[5] WIKIPEDIA : Elementary cellular automaton. `https://en.wikipedia.org/wiki/Elementary_cellular_automaton`, 2018. Consulté en févirer 2018.

[6] Tullio CECCHERINI-SILBERSTEIN et Michel COORNAERT : *Cellular Automata and Groups*, chapitre 5 : The Garden of Eden Theorem. Springer, 2010.

[7] Christiann HARTMAN, Marijin J. H. HEULE, Kees KWEKKEBOOM et Alain NOELS : Symmetry in Gardens of Eden. *The Electronic Journal of Combinatorics*, 20 (3), 2013. Paper #P16.

[8] Jarkko KARI : Reversibility of 2D cellular automata is undecidable. *Physica D : Nonlinear Phenomena*, 45:379–385, 1990.

[9] Thomas H. CORMEN, Charles E. LEISERSON, Ronald R. RIVEST et Clifford STEIN : *Introduction to Algorithms*, chapitre 22 : Elementary Graph Algorithms. Massachusetts Institute of Technology, troisième édition, 2009.

[10] Marc LORENZI : Option Informatique, classes de MP et MP*, 2015. Chapitre 7 : Automates.