Synchronisation et Automates Cellulaires: La Ligne de Fusiliers

Jean-Baptiste Yunès

► To cite this version:

Jean-Baptiste Yunès. Synchronisation et Automates Cellulaires: La Ligne de Fusiliers. Modélisation et simulation. Université Paris-Diderot - Paris VII, 1993. Français. ⟨NNT : ⟩. ⟨tel-00139049⟩

HAL Id: tel-00139049
https://theses.hal.science/tel-00139049v1

Submitted on 29 Mar 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Thèse
nouveau régime

présentée à

l'Université Paris 7

pour obtenir le titre de

Docteur de l'Université Paris 7

Spécialité

Informatique

par

Jean-Baptiste Yunès

Sujet de la thèse

Synchronisation et Automates Cellulaires : La ligne de fusiliers

Soutenue le 17 février 1993 à 16 heures
en salle 107, couloir 55-56 (salle dite du "sous-marin")
devant le jury composé de Messieurs :

Maurice	Nivat	(Président)
Serge	Grigorieff	(Directeur)
Jean-Paul	Delahaye	(Rapporteur)
Patrick	Greussay	(Rapporteur)
Jacques	Mazoyer	(Rapporteur)
Paul	Feautrier	
Jean-Marie	Rifflet	

Introduction

Cinquante ans seulement nous séparent des premiers pas de la construction de calculateurs numériques électroniques. Et depuis la seconde guerre mondiale la plupart des machines construites l'ont été sur le modèle dit *séquentiel* défini par J. von Neumann : une mémoire et un processeur. Pourtant force est de constater l'essor grandissant de nouveaux types d'ordinateurs : les machines dites *parallèles* qui possèdent plusieurs processeurs et une ou plusieurs mémoires. Nous disons *nouveaux types* bien que l'on ne doive jamais oublier que J. von Neumann proposa déjà à son époque des architectures parallèles revenues depuis quelques temps au goût du jour. Il existe aujourd'hui une grande diversité de parallélismes : `MIMD`, `SIMD`, vectoriel, massif, synchrone, asynchrone, etc, pour lesquels beaucoup de travaux restent à faire. Ce travail ne concerne qu'un type particulier de parallélisme : les automates cellulaires qu'avait inventé von Neumann à la fin des années 40.

Bien que mû par d'autres motivations que von Neumann qui cherchait à modéliser des phénomènes comme l'auto-reproduction dans des machines de structures particulières, nous pensons que l'automate cellulaire est un modèle sinon d'avenir, au moins intéressant du point de vue des problèmes qu'il pose concernant notre vision des algorithmes. L'automate cellulaire présente, en effet, des atouts non négligeables : premièrement, le nombre gigantesque de processeurs que l'on s'autorise laisse à penser qu'il ne s'agit plus seulement d'un saut quantitatif, deuxièmement, son fonctionnement synchrone nous permet de *visualiser* simplement la progression des calculs dans la structure.

Comme le *Jeu de la vie* inventé par J. Conway, le problème de la *synchronisation d'une ligne de fusiliers* est un prétexte bien agréable pour aborder la programmation d'ordinateurs massivement parallèles à fonctionnement synchrone. Ainsi notre quête d'automates très particuliers capables de synchroniser une ligne finie de cellules nous permet d'appréhender une nouvelle sorte de programmation : l'algorithmique géométrique dans laquelle les algorithmes ne seraient plus des objets statiques figés dans un langage puis éxécutés par des machines mais des objets dynamiques évoluants dans des espaces possédant une géométrie discrète.

Cette thèse étudie un problème ouvert concernant l'existence ou non d'un automate synchronisant dont l'ensemble des états ne contiendrait que 5 éléments. Deux approches sont décrites dans ce travail :

- Une première (descendante et constructive), montre comment après avoir construit de façon *brutale* un automate réalisant certaines fonctions avec 13 états, il est possible de diminuer la cardinalité de l'ensemble des états par des codages successifs. Ainsi

nous montrons comment fabriquer des automates synchronisants selon la méthode dite de Minsky. Un premier automate est fabriqué qui synchronise en temps $3n + 2\theta_n \log n + C$ $(0 \leq \theta_n \leq 1)$ avec 7 états. Un deuxième automate réalise le problème de la synchronisation en temps $3n - 2\theta'_n \log n + C$ $(0 \leq \theta'_n \leq 1)$ avec 7 états. Les deux solutions utilisent des signaux construits sur deux cellules voisines (épaisseur 2) permettant de coder les signaux d'épaisseur 1 fabriqués dans la solution originelle à 13 états. Une solution avec un temps de synchronisation uniforme $(n - 3)$ avec 9 états est présentée en annexe.

– Une deuxième (ascendante et analytique), étudie certains comportements assez surprenants observés dans quelques automates n'ayant qu'un nombre très réduit d'états. La recherche de solutions en temps quelconque (non minimal) nous a conduit à découvrir que certains automates possédant une fonction de transition linéaire passent, lors de leur calcul, par toutes les configurations possibles (calculs exponentiellement longs). Ensuite nous verrons comment nous pouvons ramener l'étude de ces automates à celles d'automates cylindriques bien plus faciles à manipuler car sans effets de bord. Le fait que certains calculs simulent le comportement d'automates treillis en pavant le diagramme espace-temps par des losanges nous permet d'expliquer l'évolution de certaines lignes.

Le texte de cette thèse comporte trois chapitres :

1. Le premier introduit toutes les définitions pertinentes et fait le tour du problème de la synchronisation d'une ligne de fusiliers et de ses variations.

2. Le chapitre deux présente l'approche descendante et constructive mentionnée plus haut, il est constitué d'un article en langue anglaise accepté pour publication dans TCS [38].

3. Le dernier chapitre développe l'approche ascendante et analytique.

Chapitre 1

La synchronisation d'une ligne de fusiliers

C'est en 1957 que J. Myhill inventa le problème dit du *Firing Squad*[1] (*Ligne de Fusiliers*). Mais la première version écrite du problème est dûe à Moore [27] :

> "Considérons une ligne finie (mais arbitrairement longue) d'automates finis tous identiques à l'exception des deux situés à chacun des deux bouts. Le fonctionnement du système est synchrone, et l'état de chacune des cellules au temps $t+1$ ne dépend uniquement que de l'état de ses deux voisines et du sien au temps t. Au départ, toutes les machines sont dans un état de repos (dit état *latent*), sauf l'une des deux situées aux bords. Le problème est alors de spécifier l'ensemble (fini) des états que peuvent prendre les automates ainsi que la fonction de transition d'état, et ce de telle façon que tous se mettent en même temps (par rapport à l'horloge globale *rythmant* le système) dans un état particulier jamais apparu avant (dit état de *feu*)."

La *définition* suivante n'est qu'une façon imagée de poser le problème, mais a permis de donner son nom au problème :

> Comment synchroniser une ligne de fusiliers de façon à ce qu'ils se mettent à tirer ensemble, alors que l'ordre donné par un général depuis l'un des deux bords de l'escadron met un certain temps à se propager ?

Il fut tout d'abord résolu par J. Mc Carthy et M. Minsky [25], cela leur permettait de *démarrer* de façon synchrone la copie d'une machine qui venait de s'auto-reproduire. Ils utilisèrent la méthode dite du *divide and conquer*.

1. FSSP : Firing Squad Synchronization Problem.

C'est E. Goto (non publié) qui trouva la première solution en temps minimal [2]. L'automate comportait plusieurs milliers d'états.

Puis A. Waksman [36] et parallèlement R. Balzer [3] trouvèrent chacun une solution en temps minimal avec assez peu d'états : Waksman avec 16 états et Balzer avec 8 états.

Vingt ans plus tard J. Mazoyer [20] découvrit une solution en temps minimal avec 6 états seulement (cette solution reste l'optimum connu à ce jour). Sa construction lui permit d'exhiber toute une famille de solutions synchronisantes fonctionnant en temps minimal.

La modélisation naturelle de ce problème est de le ramener à la recherche d'un automate cellulaire correctement spécifié. Maintenant que de nombreuses solutions sont connues la complexité de ces automates peut être étudiée (complexité exprimée en nombre d'états et/ou en temps de synchronisation).

1.1 Définitions

Nous présentons, ici, un certain nombre de définitions permettant de spécifier formellement des notions couramment utilisées par les cellularistes. Ces définitions ne concernent qu'une classe réduite d'automates cellulaires car nous ne nous intéressons (dans ce travail) qu'à la synchronisation des réseaux linéaires finis avec voisinage de rayon 1.

Définition 1.1 *On appelle Automate Cellulaire Linéaire Fini (ACLiF) toute paire $\mathcal{A} = < \mathcal{Q}, \delta >$, où :*

- *$\mathcal{Q}$ est un ensemble fini d'états, dans lequel on distingue trois d'entre eux :*

  q_l : *dit état latent, quiescent ou de repos.*

  q_i : *dit état initial, départ ou* général *(car il donne l'ordre).*

  q_f : *dit état de feu ou de synchronisation.*

  Remarque : *On utilisera par la suite un état appellé état de bord : $q_\$ \notin \mathcal{Q}$ (c'est le seul état du système que l'on considèrera comme n'étant pas interne).*

- *δ est appellée* fonction de transition locale *et vérifie :*

$$\delta \; : \; \mathcal{Q} \cup \{q_\$\} \times \mathcal{Q} \times \mathcal{Q} \cup \{q_\$\} \longrightarrow \mathcal{Q}$$

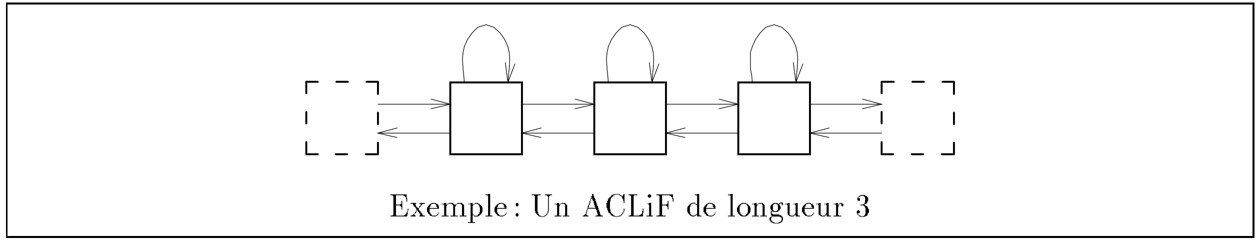
- *On impose à l'état q_l de vérifier la condition :*

$$\delta(q_\$, q_l, q_l) = \delta(q_l, q_l, q_l) = \delta(q_l, q_l, q_\$) = q_l$$

  ceci afin de ne pas créer ce que l'on nomme parfois génération spontanée.

2. On dit qu'une solution est en temps minimal si une ligne de longueur n est synchronisée en $2n-2$ transitions. Intuitivement il s'agit bien du minimum possible car c'est le temps nécessaire à une information partie de l'un des deux bords pour atteindre l'autre et revenir.

Définition 1.2 *Une ligne de n ACLiFs, notée $\mathcal{A}_n$, est constituée par juxtaposition de n machines $\mathcal{A}$ numérotées de 1 à n. Une telle ligne est souvent appellée ACLiF de longueur n.*



Exemple : Un ACLiF de longueur 3

Définition 1.3 *La configuration d'un ACLiF de longueur n est une fonction :*

$$\mathcal{C}^n \ : \ [1,n] \rightarrow \mathcal{Q}$$

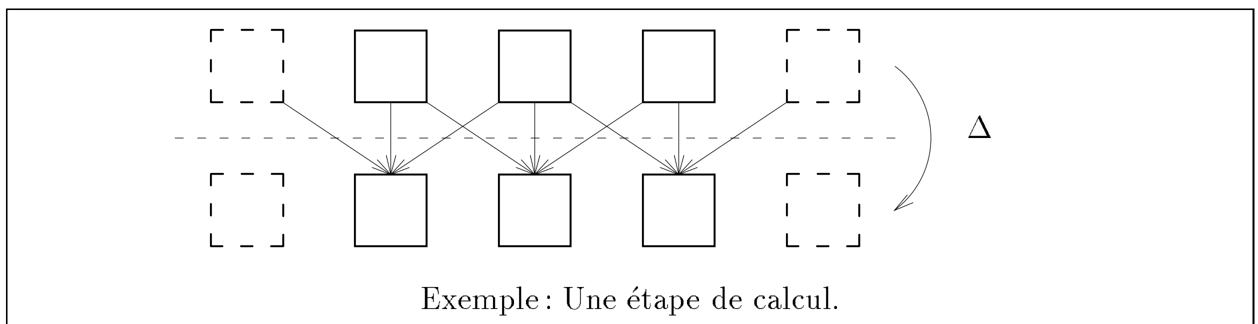
Définition 1.4 *Sur $\mathcal{Q}^\star = \cup_{i=1}^{\infty} \mathcal{Q}^i$ (ensemble de toutes les configurations finies), la fonction de transition locale δ induit une* fonction de transition globale *:*

$$\Delta \ : \ \mathcal{Q}^\star \longrightarrow \mathcal{Q}^\star$$

Définition 1.5 *La fonction Δ prend une configuration de longueur n ($\mathcal{C}^n$) et lui associe une configuration de longueur n ($\mathcal{C}'^n$) compatible avec la fonction de transition locale, δ, de la façon suivante (on note $\mathcal{C}^n(i)$ l'état de la i-ème cellule de la configuration $\mathcal{C}^n$) :*

$$\begin{array}{rl} & \mathcal{C}'^n(1) = \delta\left(q_\$, \mathcal{C}^n(1), \mathcal{C}^n(2)\right) \\ \forall\, i \in]1,n[\quad , & \mathcal{C}'^n(i) = \delta\left(\mathcal{C}^n(i-1), \mathcal{C}^n(i), \mathcal{C}^n(i+1)\right) \\ & \mathcal{C}'^n(n) = \delta\left(\mathcal{C}^n(n-1), \mathcal{C}^n(n), q_\$\right) \end{array}$$

Le passage *de $\mathcal{C}^n$ à $\Delta(\mathcal{C}^n) = \mathcal{C}'^n$ est appellé* étape *de calcul et s'effectue en 1 unité de temps (résolution minimale de l'horloge globale).*



Exemple : Une étape de calcul.

Définition 1.6 *L'*évolution *d'un ACLiF $\mathcal{A}_n$ est une suite d'étapes de calcul : $\mathcal{C}^n_{t_0}, \ldots, \mathcal{C}^n_t, \ldots$ où $\mathcal{C}^n_{t_0}$ est la* configuration initiale *qui n'est pas issue d'un calcul mais fabriquée par intervention extérieure, et où $\mathcal{C}^n_{i+1}$ est la configuration calculée par application synchrone de la fonction de transition locale δ sur chaque nœud de l'ACLiF $\mathcal{A}_n$ dans la configuration $\mathcal{C}^n_i$.*

Définition 1.7 *On appelle* valeur du site*, noté* $\begin{bmatrix} t \\ i \end{bmatrix}$*, l'état de la cellule de rang i au temps t, autrement dit c'est l'état de la cellule de rang i de la t-ième configuration de l'évolution d'un ACLiF.*

Ainsi le calcul de $\mathcal{C}_{t+1}^{n}$ s'exprime par les équations suivantes :

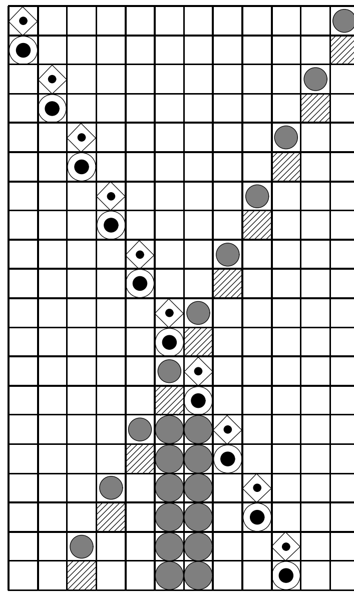
$$\begin{array}{rl} & \begin{bmatrix} t+1 \\ 1 \end{bmatrix} = \delta\left(q_{\$}, \begin{bmatrix} t \\ 1 \end{bmatrix}, \begin{bmatrix} t \\ 2 \end{bmatrix}\right) \\ \forall\, i \in]1,n[\quad , & \begin{bmatrix} t+1 \\ i \end{bmatrix} = \delta\left(\begin{bmatrix} t \\ i-1 \end{bmatrix}, \begin{bmatrix} t \\ i \end{bmatrix}, \begin{bmatrix} t \\ i+1 \end{bmatrix}\right) \\ & \begin{bmatrix} t+1 \\ n \end{bmatrix} = \delta\left(\begin{bmatrix} t \\ n-1 \end{bmatrix}, \begin{bmatrix} t \\ n \end{bmatrix}, q_{\$}\right) \end{array}$$

ou bien encore par :

$$\mathcal{C}_{t+1}^{n} = \Delta(\mathcal{C}_{t}^{n}) \Longleftrightarrow \mathcal{C}_{t+1}^{n} = \Delta^{t+1}(\mathcal{C}_{t_0}^{n})$$

Définition 1.8 *On appelle* diagramme espace-temps *l'ensemble formé des sites parcourus par le calcul d'un ACLiF de longueur n avec les valeurs de ceux-ci :*

$$D_{et}(\mathcal{A}_n) = \{< i, t, \begin{bmatrix} i \\ t \end{bmatrix} > \ / \ i \in \ [1,n]\, ,\ t \geq 1\}$$



Exemple : Un diagramme espace-temps d'ACLiF

Définition 1.9 *On appelle* configuration initiale du FSSP[3] *de $\mathcal{A}_n$, la configuration $\mathcal{C}_{t_0}^{n}$:*

$$\forall\, j \in]1,n]\, ,\ \mathcal{C}_{t_0}^{n}(j) = q_l \ \ et\ \mathcal{C}_{t_0}^{n}(1) = q_i$$

3. Comme nous ne nous intéressons qu'au FSSP, nous dirons de façon plus concise *configuration initiale.*

Définition 1.10 *La* configuration de synchronisation *est une configuration particulière dans laquelle toutes les cellules sont dans l'état de feu :*

$$\mathcal{S}^n \,:\, \forall\, i \in [1,n]\,,\, \mathcal{S}^n(i) = q_f$$

Définition 1.11 *On dit qu'un calcul menant à la configuration de synchronisation est* valide *si le diagramme espace-temps vérifie la condition :*

$$\forall\, t \in [1,s[\,,\, \forall\, i \in [1,n]\,,\, \left[\begin{smallmatrix} t \\ i \end{smallmatrix}\right] \neq q_f$$

autrement dit si avant d'arriver à la synchronisation aucune cellule n'est passée dans l'état de feu.

Le problème de la synchronisation d'une ligne de fusiliers peut alors s'exprimer de la façon suivante :

Existe-t'il un ensemble $\mathcal{Q}$ et une fonction δ telle que pour tout n le calcul de $\mathcal{A}_n$ partant de la configuration initiale $\mathcal{C}^n_{t_0}$ est un calcul valide menant à la configuration de synchronisation $\mathcal{S}^n$?

Définition 1.12 *Si une telle fonction δ existe alors on peut définir la* fonction de temps de synchronisation *qui pour un automate $\mathcal{A}$ est :*

$$\mathcal{F}_s(n) = min\{t \text{ tel que } \Delta^t(\mathcal{C}^n_{t_0}) = \mathcal{S}^n\}$$

et donne pour une longueur de ligne donnée le nombre d'étapes de calcul nécessaires pour arriver à la synchronisation.

1.2 Généralisations du FSSP

Au delà du nombre de solutions disponibles pour le problème tel qu'il est défini par Moore, il est intéressant de noter qu'un nombre important de généralisations a été apportées au FSSP.

1.2.1 dans les lignes d'automates

[25, 22]

Minsky se posa la question de savoir si l'on pouvait synchroniser avec un canal de communication ne permettant de transmettre qu'une information limitée (en tout cas plus petite que la donnée de l'état).

Un très beau résultat de Mazoyer montre comment il est possible de synchroniser en temps minimal en ne transmettant qu'un seul bit d'information à chaque intervalle de temps. L'automate réalisant cette fonction possède 58 états internes.

[26]

Moore et Langdon ont résolu le problème dans le cas d'un ordre donné non plus de l'un des deux bords mais depuis n'importe quel site de la ligne. Leur automate synchronise en temps minimal et possède 17 états.

Le temps de synchronisation d'une telle solution est $2n-2-k$ unités de temps, avec $k = min(p-1, n-p)$ si p est le numéro de la cellule qui lance le processus.

[35]

Varshavsky *et al.* présentent un automate capable de synchroniser une ligne depuis n'importe quelle cellule.

Ils donnent une solution permettant de synchroniser une ligne si chaque machine a un temps de préparation au feu qui lui est propre (chaque soldat met un certain temps à monter son fusil sur l'épaule et à appuyer sur la gâchette).

Une dernière généralisation est présentée : la synchronisation d'une ligne d'automates avec des lignes de communication ayant un délai de latence[4].

[9, 8]

Herman *et al.* se sont intéressés aux lignes d'automates croissantes. Ils voulurent construire, par analogie avec les organismes vivants, une structure capable de croître pendant qu'un processus de synchronisation évoluait. Une technique permettant de réaliser une telle fonction est exposée.

[23]

Mazoyer s'intéresse à la façon de synchroniser deux cellules reliées par un canal de communication dont le temps de latence est constant mais non connu. Une solution en temps $2\Delta(\lceil \log_2 \Delta \rceil + 1) + \Delta$ est obtenue avec $\Delta = 2\tau + 1$ si τ est le temps de communication entre les deux cellules.

Il prouve l'impossibilité de synchroniser deux cellules si le temps de latence du canal de communication n'est pas le même pour chaque sens de transmission.

Il n'est pas non plus possible de réaliser la synchronisation si le temps de transition d'état de chaque cellule n'est pas identique.

4. On appelle temps de latence d'un canal le temps nécessaire à une information émise à l'une des extrémités pour parvenir à l'autre.

[24]

Mazoyer indique comment synchroniser une ligne finie d'automates dont les temps de latence des différents canaux de communication entre les machines ne sont pas égaux. La synchronisation est possible en temps $\Delta(2\Delta + 5)$ où Δ est le délai-diamétral[5].

1.2.2 dans des réseaux de topologie régulière

[32]

Shinahr propose de synchroniser des structures régulières (carré, rectangle, cube) en temps minimal (en utilisant le résultat de Moore et Langdon [26]) :

carré : $2n - 2$ unités de temps pour un carré de $n \times n$ cellules.

rectangle : $n + m + max(n, m) - 3$ unités de temps pour un rectangle de $n \times m$ cellules.

cube : $3n - 3$ unités de temps pour un cube de $n \times n \times n$ cellules.

[6]

Grasselli propose une technique permettant de résoudre le FSSP dans une structure planaire connexe et de forme quelconque.

[29, 30]

Romani a étudié les solutions de synchronisation dans des graphes en forme de *fleurs*. La symétrie intrinsèque de ses graphes lui permet d'accélérer le temps de synchronisation.

[17]

Kobayashi étudie le FSSP dans tout sous-ensemble fini et connexe de cellules du plan. Le général étant situé arbitrairement dans la structure, des solutions en temps minimal sont construites pour certaines familles de graphes.

---

5. Il s'agit du temps nécessaire à un signal pour parcourir le chemin qui mène aux deux cellules les plus éloignées du réseau : sur une ligne c'est le temps qu'il faut à un signal émis par la première cellule pour atteindre la dernière.

[33]

Szwerinski étend la synchronisation aux hyper-parallélépipèdes[6] et construit une solution en temps minimal avec un général situé n'importe où.

1.2.3 dans des graphes quelconques

[31]

Outre la résolution de divers problèmes dans les graphes d'automates, Rosenstiehl *et al.* étudient le Firing Squad dans des graphes d'automates de degré borné.

[18, 19]

Kobayashi obtient des solutions pour des classes très particulières de machines : les *poly-automates* (il s'agit de réseaux d'automates possédant des *fan-in* et *fan-out* variables).

[12, 13]

Jiang expose une solution permettant de synchroniser n'importe quel réseau d'automates, c'est-a-dire un réseau de topologie arbitraire avec des temps de communication qui ne sont pas égaux. Soit τ_{max} la plus longue de ces durées. Le temps de synchronisation est alors borné par $O((\Delta+\tau_{max})^3)$ où Δ est le temps maximum de communication entre deux cellules quelconques.

Une solution est présentée dans le cas où il y a existence d'un nombre fixé de généraux. Il démontre qu'il n'y a pas de solution générale au problème si un nombre arbitraire de généraux est permis.

1.3 Les différentes méthodes de synchronisation

1.3.1 La méthode de Minsky

Plus connu sous le nom de *divide and conquer*, le principe utilisé par Minsky et Mac Carthy (voir figure 1.1) est de *découper* récursivement la ligne initiale en deux parties de longueurs égales. Ceci jusqu'à l'obtention d'un certain nombre de lignes de longueur 1 que l'on sait synchroniser trivialement.

Le procédé conduit à une solution de synchronisation en temps non minimal, puisque la suite des temps auxquels les coupures ont lieu converge vers $3n$. Un premier signal est construit,

6. On appelle hyper-parallélépipède tout parallélépipède défini dans un espace de dimension quelconque (par analogie avec les hyper-cubes).

$\mathcal{S}_1$. Il se propage depuis la cellule 1 jusqu'à la cellule n, ceci en n unités de temps. Puis "rebondit" sur la cellule n pour retourner vers la cellule $n/2$ (qu'il atteint en $n/2$ unités de temps) où il rencontre le signal $\mathcal{S}_3$ venant directement de la cellule 1 (parti en même temps que $\mathcal{S}_1$). On a donc créé deux sous-lignes de longueur $n/2$ dans lesquelles il suffit de relancer le processus de façon symétrique. Lorsque les sous-lignes créées sont suffisamment petites on peut alors synchroniser.

Les coupures s'effectuent aux temps :

$$t_1 = \frac{3n}{2}, t_2 = \frac{9n}{4}, \ldots, t_i = \frac{3n(2^i - 1)}{2^i}, \ldots$$

Au temps t_i, i coupures se produisent simultanément sur les sites :

$$\forall j \in [1, 2^{i-1}], \left[\frac{t_i}{\frac{2j-1}{2^i}}\right]$$

Bien entendu, les calculs précédents ne sont valables que lorsque n est une puissance de deux. Dans le cas général, il convient d'introduire des $\lfloor\rfloor$ et des $\lceil\rceil$ de ces valeurs de sorte à rester en coordonnées entières (voir 2.8).

1.3.2 La méthode de Waksman-Balzer

La synchronisation obtenue par cette méthode est optimale du point de vue du temps : $2n-2$ étapes de calcul.

Le principe utilisé est identique au précédent puisqu'il s'agit de couper la ligne initiale en deux puis en quatre, huit... Dans celle-ci on n'attend pas d'avoir complètement terminé la première coupure pour lancer le calcul de la suivante. En effet il suffit de générer un autre signal de pente $1/3$ au moment du rebond du premier signal de pente 1 sur le bord (voir figure 1.2). Ceci afin d'obtenir une image miroir homothétique de la solution dès l'instant n. Mais procédant ainsi, il faut alors créer une autre coupure "image" afin de garder la symétrie dans le processus de synchronisation. Pour cela il faut au départ, lancer un signal de pente $1/7$. En réitérant le raisonnement ainsi, il apparait qu'une infinité potentielle de signaux doit être créée (or l'automate ne doît posséder qu'un nombre fini d'états). Le signal $\mathcal{S}_i$ devra avoir pour vitesse $\frac{1}{2^i-1}$. La principale difficulté de cette solution est donc de générer une famille infinie de signaux à l'aide d'un nombre fini d'états.

Les coupures s'effectuent aux temps :

$$t_1 = \frac{3n}{2}, t_2 = \frac{7n}{4}, \ldots, t_i = \frac{(2^{i+1} - 1)n}{2^i}, \ldots$$

Au temps t_i, i coupures se produisent simultanément sur les sites :

$$\forall i \in [1, 2^{i-1}], \left[\frac{t_i}{\frac{2j-1}{2^i}}\right]$$

1.3.3 La méthode de Mazoyer

La synchronisation obtenue par cette méthode est une synchronisation en temps minimal (voir figure 1.3).

La différence principale par rapport à la solution de Waksman ou Balzer est qu'il n'y a dans celle-ci que des homothéties, pas d'image miroir de la solution. En fait il s'agit de couper aux 2/3, puis aux 4/9, 8/27... A chacune de ces coupures, le processus est *relancé* dans le même sens (il n'y a pas de notion de solution *image* dans ce procédé (voir figure 1.3).

Dans cette solution les temps auxquels les coupures se produisent ne sont pas simples à calculer et l'on peut remarquer que toutes les coupures ne se produisent pas de façon symétrique par rapport à un axe médian, comme dans les solutions de Waksman-Balzer ou Minsky.

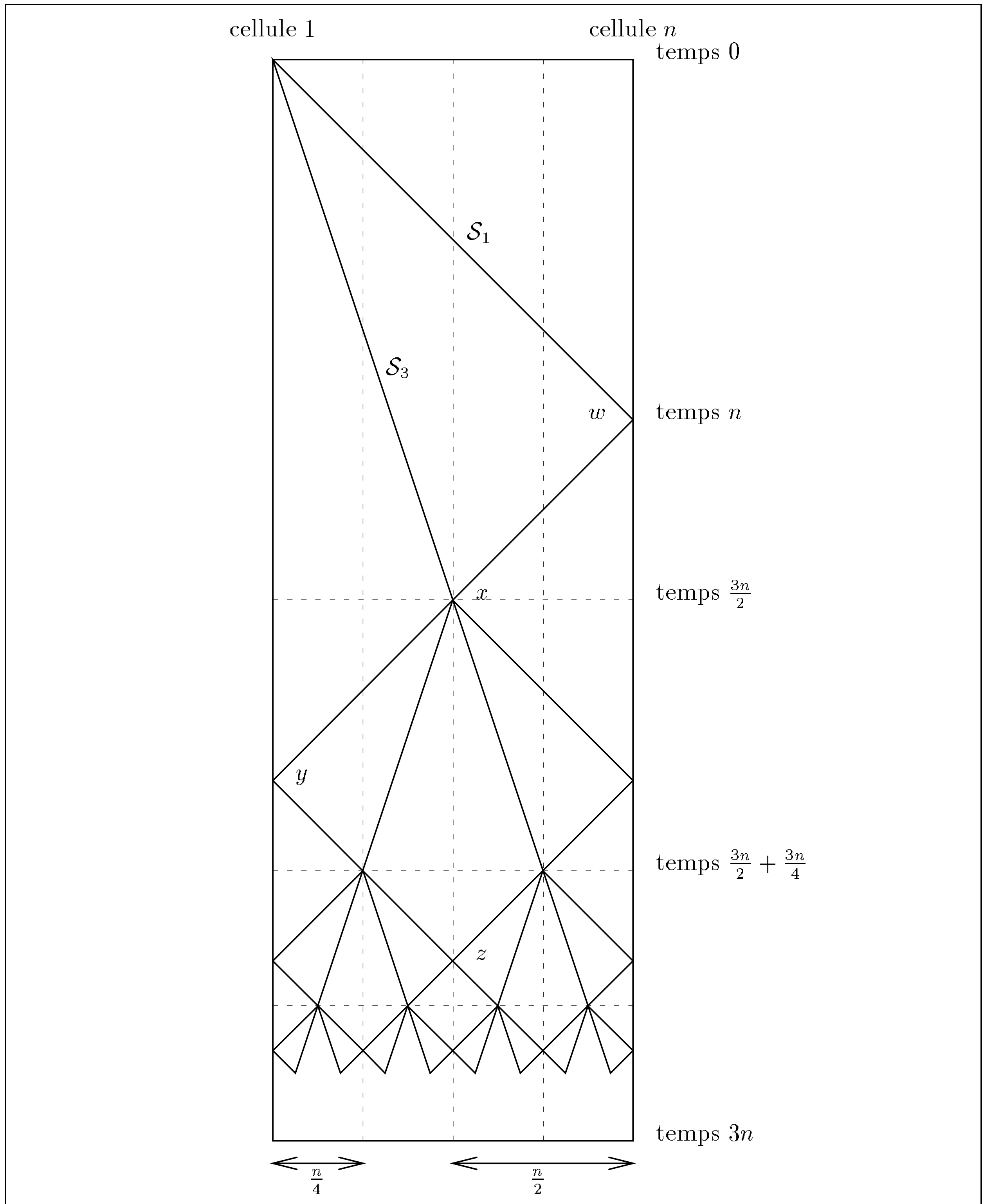


FIG. 1.1 - *Solution de type Minsky*

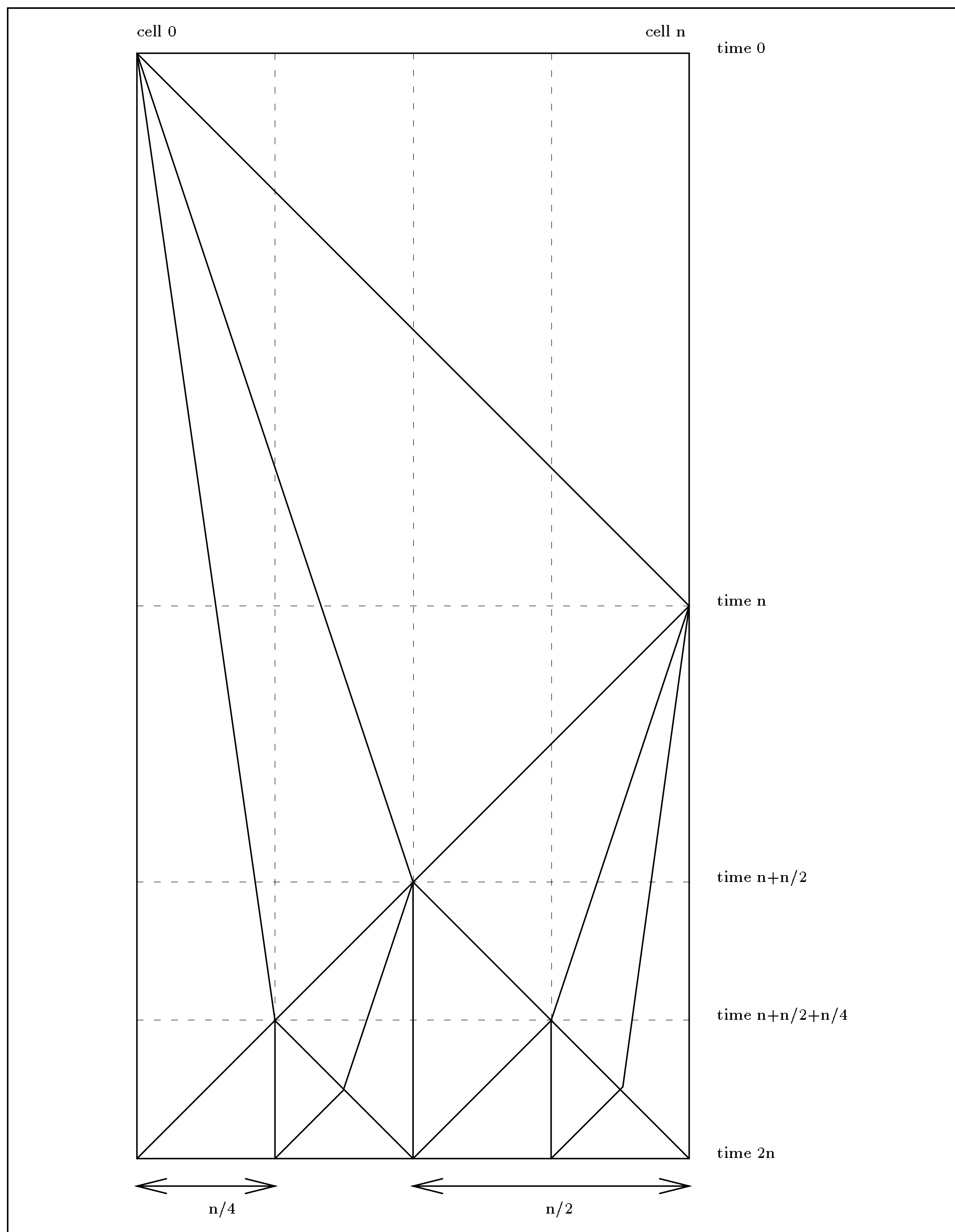


Fig. 1.2 - *Solution de type Waksman-Balzer*

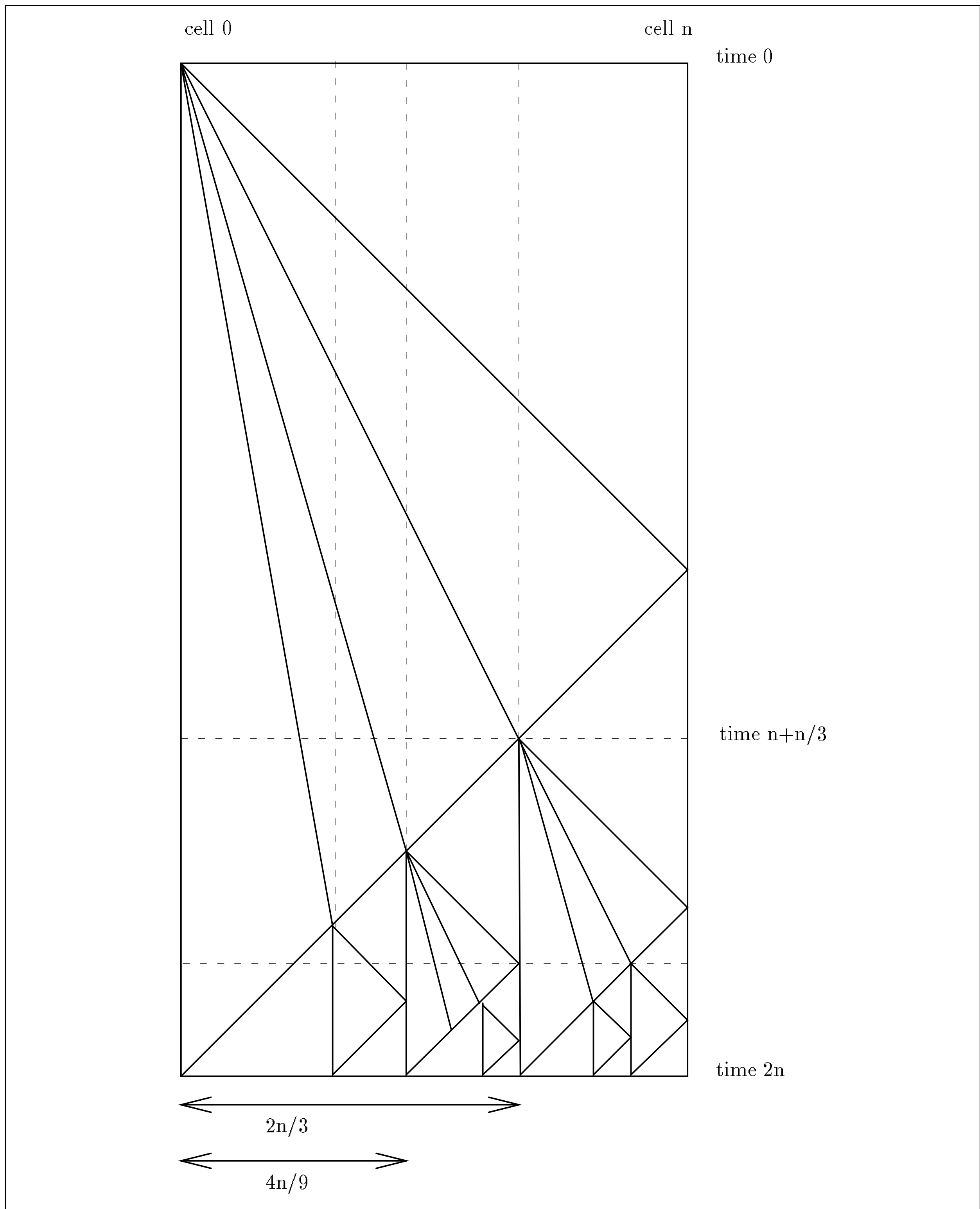


FIG. 1.3 - *Solution de type Mazoyer*

Chapitre 2

Une approche descendante

We show that seven states are enough to implement MINSKY-like solutions to the Firing Squad Synchronization Problem, in a linear network of n finite state cellular automata with null transmission delay.

2.1 Introduction

Originating with J. MYHILL in 1957, the FSSP[1] can be expressed as follows :

> Given a firing squad and knowing that any order given by a general located at one of the two ends needs n units of time to reach the n-th soldier, how can we make all soldiers fire at the very same time ?

We consider that each one of the soldiers is a finite state cellular automaton with linear Von NEUMAN neighbourghood (a soldier is sometimes called node, machine or cell). All cells operate synchronously at discrete steps, and communicate with their nearest neighbours, either sending or receiving signals. The problem comes down to :

- Does there exist automata such that a synchronization occurs whatever the length of the line may be ?
- How complex are such automata ?

In 1964, E.F. MOORE [26, page 213] spread out the FSSP through a publication. M. MINSKY and J. Mac CARTHY [25, page 28] solved the FSSP in 1965 with the *Divide and Conquer* method. Their method works in time $3n$, where n is the length of the line. We will see later that it's possible to do it simply with 13 or 14 states. A year later, E. GOTO (unpublished) found a minimal time[2] solution with a few thousands states. A. WAKSMAN and

[1] FSSP : means Firing Squad Synchronization Problem.

[2] ie : $2n - 2$ units of time for n cells.

R. BALZER in 1967 [3, page 22-42], [36, page 66-78] got an 8-states minimal time solution. Later on, R. BALZER using a computer, checked that synchronization is impossible with 4 states only. About twenty years later, in 1986, J. MAZOYER [20] found a 6-states minimal time solution to the FSSP. Is it an optimal solution ? It is an open problem whether there exists a 5-states minimal time solution or not.

MINSKY-like solutions and WAKSMAN-BALZER-MAZOYER ones are totally different. MINSKY solutions use isolated signals while W-B-M ones fill the plane with an infinity of signals. In a certain sense, we could say that MINSKY-like solutions are much simpler than W-B-M ones: they are *threadlike solutions* while W-B-M are *planar* ones.

A simple implementation of MINSKY's solution in time $3n + \theta_n \log n + C$, where $0 \leq \theta_n < 1$, uses 13 states (see figure 2.1). We show that seven states are sufficient to get MINSKY-like solutions in time $3n \pm 2\theta_n \log n + C$, where $0 \leq \theta_n < 1$. Patrick C. Fischer (see [5]) described a solution with 15 states which synchronizes in time $3n - 4$. J. Mazoyer built one with 14 states synchronizing in time $3n - 1$, and it is possible to transform his solution to an automaton which synchronize in time $3n - 3$ with only 9 states (see annex A).

Our solutions refute MAZOYER's suggestion implying that solutions with few states are necessarily minimal time ones. This is not the case and MINSKY-like solutions can be realized with an automaton with seven states.

2.2 Definitions

2.2.1 Practical definitions

In practice if an LCA $\mathcal{A} = (\mathcal{Q}, \delta)$ is a solution to the FSSP then the whole of δ is not useful: the set of triples (q_1, q_2, q_3) appearing in some of the configurations $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_s$ is a proper subset of the domain of δ.

Thus, to describe a particular solution to the FSSP we shall explicit a restriction of δ. Such a restriction can be given in the form of a set of transition rules $(q_1, q_2, q_3) \longrightarrow q_4$.

2.3 MINSKY's solution

As we can see in figure 1.1, it's a dichotomic process. We split the line into two equal parts and let them iterate the process recursively. When it becomes impossible to cut, then synchronization occurs. Two signals are needed to cut a line, both starting at cell 1 and propagating rightwards:

- A signal of *speed* 1 ($y = x$) called $\mathcal{S}_1$.
- A signal of *speed* $\frac{1}{3}$ ($y = 3x$) called $\mathcal{S}_3$.

Let n be the length of the line. When $\mathcal{S}_1$ meets cell n, it bounces on and propagates leftwards until it crosses $\mathcal{S}_3$. $\mathcal{S}_1$ crosses $\mathcal{S}_3$ at cell $n/2$ and time $n + \frac{n}{2}$:

- n units of time for $\mathcal{S}_1$ to go through the entire line, plus $n/2$ to go back from cell n to cell $n/2$ at the middle of the line
- $\frac{3n}{2}$ units of time for $\mathcal{S}_3$ to go directly at speed $\frac{1}{3}$ to cell $n/2$.

Let:

- $\mathcal{T}(k)$ be the time needed to cut a line of length k, $\mathcal{T}(k) = \frac{3k}{2}$.
- $\mathcal{TS}(k)$ be the time needed to synchronize a line of length k.

In a continous universe we would have:

$$\mathcal{TS}(n) = \sum_{j=0}^{\infty} \mathcal{T}(\frac{n}{2^j}) = \sum_{j=0}^{\infty} \frac{3n}{2^{j+1}} = 3n$$

But LCAs are in a discrete world, and the result is not exactly $3n$ in that case. Moreover, how to cut a line in two parts of equal length ?

- If the line has odd length, do we need to mark the central cell and get two lines of length $\frac{n-1}{2}$, or get two lines of length $\frac{n+1}{2}$ and use a common cell ?
- If the line has even length, do we need to mark the two central nodes and get two lines of length $\frac{n-2}{2}$, or construct two lines of length $\frac{n}{2}$?

These choices are really important in the strategy, but are not so influent in the synchronization time, as we shall see.

2.4 A 13-states solution

2.4.1 How many states ?

Let us enumerate what we need:

- 1 quiescent state q_q, also denoted by a blank.
- 1 firing state q_f, also denoted by `F`.
- 1 state `R` for $\mathcal{S}_1$ going from left to right.
- 1 state `L` for $\mathcal{S}_1$ going from right to left.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
I														0
X	R													1
X		R												2
	A		R											3
	B			R										4
	C				R									5
		A				R								6
		B					R							7
		C						R						8
			A						R					9
			B							R				10
			C								R			11
				A								R		12
				B									R	13
				C									L	14
					A							L		15
					B						L			16
					C					L				17
						A			L					18
						B		L						19
						C	L							20
						I	I							21
					L	I	I	R						22
				L		I	I		R					23
			L		a	I	I	A		R				24
		L			b			B			R			25
	L				c			C				R		26
L				a					A				R	27
R				b					B				L	28
	R			c					C			L		29
		R	a							A	L			30
			I							I				31
		L	I	R					L	I	R			32
	L		I		R			L		I		R		33
L		a	I	A		R	L		a	I	A		R	34
R		b		B		L	R		b		B		L	35
	R	c		C	L			R	c		C	L		36
	I	I		I	I			I	I		I	I		37
L	I	I	X	I	I	R	L	I	I	X	I	I	R	38
R	I	I	Y	I	I	L	R	I	I	Y	I	I	L	39
F	F	F	F	F	F	F	F	F	F	F	F	F	F	40

Figure 2.1: Execution of an LCA with 13 states and 14 cells

- 3 states `A`, `B` and `C` for $\mathcal{S}_3$ going from left to right.
- 3 states `a`, `b` and `c` for $\mathcal{S}_3$ going from right to left.
- 3 auxiliaries states `I`, `X` and `Y` (needed for start and collisions).

The count is 13.

2.4.2 How to propagate $\mathcal{S}_1$?

Table 2.1 shows the signal $\mathcal{S}_1$ (see lines 6-8 columns 7-9 in figure 2.1) and the associated transition rules for it's propagation. Table 2.2 shows a bounce of signal $\mathcal{S}_1$. An example of

What we want					Associated rules	Symmetrical signal :
...	R			...	(R, ,)→(R)	(L, ,)→( )
...		R		...	(,R,)→( )	(,L,)→( )
...			R	...	(, ,R)→( )	(, ,L)→(L)

Table 2.1: Propagation of signal $\mathcal{S}_1$

this (left bounce at point `y`, right bounce at point `w` in figure 1.1) can be see at lines 12-15 columns 13-14 and lines 26-29 columns 1-2 in figure 2.1.

A bounce looks like				Associated rules	Symetrical signal :
...	R		$	(R, ,$)→(R)	($, ,L)→(L)
...		R	$	(,R,$)→(L)	($,L,)→(R)
...		L	$	(,L,$)→( )	($,R,)→( )
...	L		$	(L, ,$)→( )	($, ,R)→( )

Table 2.2: A bounce of signal $\mathcal{S}_1$

2.4.3 How to propagate $\mathcal{S}_3$?

By analogy with the signal $\mathcal{S}_1$, we only need to wait three units of time (this needs three states) in one cell before going to the left or right. This is done by the rules in table 2.3 and illustrated in figure 2.1 between lines 6-14 and columns 3-5 ($\mathcal{S}_3$ propagating rigthwards).

```
(A, , )→( )  (B, , )→( )  (C, , )→(A)
( ,A, )→(B)  ( ,B, )→(C)  ( ,C, )→( )
( , ,A)→( )  ( , ,B)→( )  ( , ,C)→( )
```

and symetrically :

```
(a, , )→( )  (b, , )→( )  (c, , )→( )
( ,a, )→(b)  ( ,b, )→(c)  ( ,c, )→( )
( , ,a)→( )  ( , ,b)→( )  ( , ,c)→(a)
```

Table 2.3: Propagation of signal $\mathcal{S}_3$

2.4.4 Bounces

Specials bounces are to be considerated, when signals $\mathcal{S}_1$ meet the *virtual* ends of line. After the line has been cut in two sublines, signals $\mathcal{S}_1$ bounce one over the other (see figure 1.1 near point `z`) so that each of them play the role of the end of line for the other. According to the parity of n, two cases occur :

- Lines of length $n = 2p$. See lines 33-36 columns 6-9 in figure 2.1.

  The crossing of $\mathcal{S}_1$ and $\mathcal{S}_3$ takes place at cells p and $p+1$ and each signal bounces one over the other without any difficulty. Each signal plays the role of end of line for the other one.

- Lines of length $n = 2p+1$. See lines 37-39 columns 10-12 in figure 2.1.

  Two signals $\mathcal{S}_1$ come on cell $p+1$, so they can't bounce one on the other. So we need to code the *pseudo-bounce* on *virtuals* ends of lines stated at cells p and $p+2$. We use two of the three auxiliary states to do it.

2.4.5 Cuts

A 'cut' refers to a point like point `x` in figure 1.1.

Signals travel through sites like this :

$\mathcal{S}_1$: $<1,0> \rightsquigarrow \ldots \rightsquigarrow <n,n-1> \rightsquigarrow <n,n> \rightsquigarrow \ldots \rightsquigarrow <n-i,n+i>$.

$\mathcal{S}_3$: $<1,0> \rightsquigarrow \ldots \rightsquigarrow <i,3i-3> \rightsquigarrow <i,3i-2> \rightsquigarrow <i,3i-1> \rightsquigarrow <i+1,3i>$ starting and ending in the same state.

Two cases are to be considered :

- $n = 2p$: The cut must occur between cells p and $p+1$, to form two lines. The first

consists of cells numbered from 1 to p, and the second from $p+1$ to n. If we suppose that the process comes from the left, then $\mathcal{S}_1$ arrives at cell $p+1$ when $\mathcal{S}_3$ arrives at p.

- Signal $\mathcal{S}_1$ arrives on cell $p+1$ (ie: $n-(p-1)$) at time $n+p-1=3p-1$.
- Signal $\mathcal{S}_3$ arrives (straight down from cell 1) at time $3(p-1)=3p-3$ in state 1.

So signal $\mathcal{S}_3$ can wait 2 units of time until $\mathcal{S}_1$ takes place in machine $p+1$. Table 2.4 shows the configuration at time $3p-1$ (refers to lines 20-21 columns 6-9 and lines 36-37 columns 1-4 in figure 2.1).

time		p	$p+1$				p	$p+1$	
$3p-1$	...	C	L	...	symetrically	...	R	c	...
$3p$	...	I	I	...		...	I	I	...

Table 2.4: An "even" cut

Thus, we have:

$$\mathcal{T}(2p) = 3p \tag{2.1}$$

- $n=2p+1$: We have to cut the line at cell $p+1$ to form two new lines, one consisting of cells 1 to $p+1$ and the other of cells $p+1$ to n.
  - Signal $\mathcal{S}_1$ arrives at $p+1$ (ie: $n-p$) at time $n+p=3p+1$.
  - Signal $\mathcal{S}_3$ arrives at $p+1$ (straight down from cell 1) at time $3(p+1-1)=3p$.

This time, it is not necessary to wait $\mathcal{S}_1$ one unit of time because we have (at time $3p+1$) the configuration describe in table 2.5. However, no new state is necessary

time		p	p+1	p+2	
$3p$	...		A	L	...
$3p+1$	...		BL		...
$3p+2$	...		I		...

Table 2.5: An "odd" cut

since at time $3p$ we can evolve as illustrated in table 2.6 (refers to lines 30-31 columns 10-13 (symetrically columns 2-5) in figure 2.1).

Then at time $3p$ we can detect the collision. Thus, we have:

$$\mathcal{T}(2p+1) = 3p+1 \tag{2.2}$$

time		p	p+1	p+2	
$3p$	...		A	L	...
$3p+1$	...		I		...

Table 2.6: An "optimized odd" cut

2.4.6 Ignition of signals

Now we need to write rules for the generation of new signals. This can be made without any difficulty, with the third auxiliary state (see columns 6-9 lines 21-24 in figure 2.1).

2.4.7 Synchronization

When lines of length 2 occur, we can simply detect this fact and then start a special process to synchronize all cells. This process makes a special context for all cells. We need 3 units of time to do that, see lines 38-40 in figure 2.1.

2.4.8 The synchronization time

From equations (2.1) and (2.2) we can deduce that:

$$\mathcal{T}(n) = \lfloor \frac{3n}{2} \rfloor \tag{2.3}$$

Let $\mathcal{L}(n)$ be the common length of the two *sub-lines* created after a cut:

$$n = 2p \Longrightarrow \mathcal{L}(n) = \frac{n}{2} \tag{2.4}$$

$$n = 2p+1 \Longrightarrow \mathcal{L}(n) = \frac{n+1}{2} \tag{2.5}$$

From (2.4) and (2.5) we can deduce that:

$$\mathcal{L}(n) = \lceil \frac{n}{2} \rceil \tag{2.6}$$

We know that $\mathcal{TS}(2) = 4$ (See section 2.4.7). Then (2.3) and (2.6) imply:

$$\mathcal{TS}(n) = \lfloor \frac{3n}{2} \rfloor + \mathcal{TS}(\lceil \frac{n}{2} \rceil) \ , \ \ \mathcal{TS}(2) = 4 \tag{2.7}$$

We show in section 2.8 that this leads to:

$$\mathcal{TS}(n) = 3n + \eta(n) + \epsilon(n) + \mathcal{TS}(2) - 7 \tag{2.8}$$

Where:

- $\eta(n)$ be the number of 1's in the $\lfloor \log \frac{n}{2} \rfloor$ last digits of the binary representation of the number $2^t - n$ for any $t > \log 3n$.

- $\epsilon(n) = \begin{cases} 1 \text{ if } n \in \bigcup_k]2^k - 2^{k-1}, 2^k] \\ 0 \text{ if } n \in \bigcup_k]2^k, 2^k + 2^{k-1}] \end{cases}$

Note:

$$\epsilon(n) = \begin{cases} 1 \text{ if } n - 2^{\lfloor \log n \rfloor} \leq 2^{\lceil \log n \rceil} - n \text{ and } \log n \notin N \\ 0 \text{ if } n - 2^{\lfloor \log n \rfloor} > 2^{\lceil \log n \rceil} - n \text{ or } \log n \in N \end{cases}$$

2.4.9 Thickness

We can see that signals have a thickness of one cell. This solution is in a subclass called *threadlike solution of thickness* 1.

2.5 A 7-states solution

2.5.1 Can we eliminate the auxiliary states ?

We can effectively suppress two of them, by resorting to some coding with the other states. The third one is needed in particular contexts such as bounces on virtual ends of line, and start. At this step, it seems impossible to replace it. This leads to a solution with 11 states.

2.5.2 $\mathcal{S}_3$ with fewer states ?

An idea is to make signals thicker (the signal is now represented in two adjacent cells). Thus, we replace a big set of states, each one used in a single cell context, with a smaller set containing states used to code different contexts in more than one cell. The table 2.7 shows how to replace the 6 states `A, B, C, a, b, c` by only 4 states `A, B, C, d`. So this leads to a 9-states solution.

...	d	A		...		...		A	d	...
...	d	B		...	symetrically	...		B	d	...
...	d	C		...		...		C	d	...
...		d	A	...		...	A	d		...

Table 2.7: $\mathcal{S}_3$ with 4 states

2.5.3 $\mathcal{S}_1$ with fewer states ?

The former method is really profitable if we see that, the new state named "`d`" can be used to code signal $\mathcal{S}_1$ (See table 2.8 and lines 5-7 columns 5-8 in figures 2.2 and 2.3). We eliminate old states "`L`" and "`R`" and add state "`Z`". So we have an 8-states solution.

1	2	3	4	5	6	7	8	9	10	11	
G											0
G	Z										1
C	d	Z									2
C	A	d	Z								3
A	A		d	Z							4
d	C			d	Z						5
	d	A			d	Z					6
	A	A				d	Z				7
	d	C					d	Z			8
		d	A					d	Z		9
		A	A						d	Z	10
		d	C							Z	11
			d	A					Z	d	12
			A	A				Z	d		13
			d	C			Z	d			14
				d	A	Z	d				15
				A	C	d					16
					Z						17
				Z	d	Z					18
			Z	d	C	d	Z				19
		Z	d	A	d	A	d	Z			20
	Z	d		d	G	d		d	Z		21
Z	d			C	d	C			d	Z	22
Z			A	d		d	A			Z	23
d	Z		d	d		A	A		Z	d	24
	d	Z	C	d		d	C	Z	d		25
		Z	Z				Z	Z			26
	Z	d	d	Z		Z	d	d	Z		27
Z	d	C	C	d	Z	d	C	C	d	Z	28
Z	A	d	d	A	G	A	d	d	A	Z	29
A	C	G	G	C	G	C	G	G	C	A	30
C	C	C	C	C	C	C	C	C	C	C	31
F	F	F	F	F	F	F	F	F	F	F	32

Figure 2.2: Execution of an LCA with 7 states and 11 cells

1	2	3	4	5	6	7	8	9	10	11	12	
G												0
G	Z											1
C	d	Z										2
C	A	d	Z									3
A	A		d	Z								4
d	C			d	Z							5
	d	A			d	Z						6
	A	A				d	Z					7
	d	C					d	Z				8
		d	A					d	Z			9
		A	A						d	Z		10
		d	C							d	Z	11
			d	A							Z	12
			A	A						Z	d	13
			d	C					Z	d		14
				d	A			Z	d			15
				A	A		Z	d				16
				d	C	Z	d					17
					Z	Z						18
				Z	d	d	Z					19
			Z	d	C	C	d	Z				20
		Z	d	A	d	d	A	d	Z			21
	Z	d		d	G	G	d		d	Z		22
Z	d			C	d	d	C			d	Z	23
Z			A	d			d	A			Z	24
d	Z		d	d			A	A		Z	d	25
	d	Z	C	d			d	C	Z	d		26
		Z	Z					Z	Z			27
	Z	d	d	Z			Z	d	d	Z		28
Z	d	C	C	d	Z	Z	d	C	C	d	Z	29
Z	A	d	d	A	G	G	A	d	d	A	Z	30
A	C	G	G	C	G	G	C	G	G	C	A	31
C	C	C	C	C	C	C	C	C	C	C	C	32
F	F	F	F	F	F	F	F	F	F	F	F	33

Figure 2.3: Execution of an LCA with 7 states and 12 cells

...	d	Z			...
...		d	Z		...
...			d	Z	...

symetrically

...			Z	d	...
...		Z	d		...
...	Z	d			...

Table 2.8: The new $\mathcal{S}_1$

2.5.4 A better way to construct $\mathcal{S}_3$

In table 2.9 we present a more subtle way to build the signal of *speed* $\frac{1}{3}$ (See lines 6-9 columns 2-4 in figures 2.2 and 2.3). This is our last optimization, and now, we have a 7-states solution.

...	d	A		...
...	A	A		...
...	d	C		...
...		d	A	...

symetrically

...		A	d	...
...		d	d	...
...		C	d	...
...	A	d		...

Table 2.9: $\mathcal{S}_3$ with 3 states

The transition tables are given in table 2.10.

2.5.5 The synchronization time

Equation (2.7) has this solution:

$$\mathcal{TS}(n) = 3n + 2\eta'(n) + \epsilon(n) + \mathcal{TS}(2) - 5 \tag{2.9}$$

Where:

- $\eta'(n)$ be the number of 1's in the $\lfloor \log \frac{n}{3} \rfloor$ last digits of the binary representation of the number $2^t - n$ for any $t > \log 3n$.
- $\epsilon(n) = \begin{cases} 1 \text{ if } n \in \bigcup_k \,]2^k - 2^{k-1}, 2^k] \\ 0 \text{ if } n \in \bigcup_k \,]2^k, 2^k + 2^{k-1}] \end{cases}$

Note:

$$\epsilon(n) = \begin{cases} 1 \text{ if } n - 2^{\lfloor \log n \rfloor} \leq 2^{\lceil \log n \rceil} - n \text{ and } \log n \notin N \\ 0 \text{ if } n - 2^{\lfloor \log n \rfloor} > 2^{\lceil \log n \rceil} - n \text{ or } \log n \in N \end{cases}$$

2.5.6 Note

The time functions (2.8) and (2.9) are not exactly equal. The factor 2 before η' in equation (2.9) is due to the fact that, each time we split a line with an odd length, we wait two units of time.

		G	Z	A	C	d	$
		Z	Z		A		
G	Z	.	.	.	.	Z	Z
Z	Z	.	Z	.	.	Z	Z
A		.	Z	.	.		.
C	A	.	.	.	.	.	.
d		Z	Z		.		
$		.	Z		.		.

G		G	Z	A	C	d	$
	G	G	.	.	.	.	.
G	G	.		G	C	d	.
Z	.			.	.	.	.
A	.	G	.	G	.	.	.
C	.	C	.	.	C	.	.
d	.	d	.	.	.	d	.
$	G	.	C	.	.	.	F

Z		G	Z	A	C	d	$
	d	d	d	.	.	d	d
G	d	.	.	A	Z	.	C
Z	d	.	.	.	.	G	.
A	.	d	.	.	.	d	A
C	.	Z	.	.	.	Z	C
d	d	.	G	A	Z	G	Z
$	d	.	.	A	C	Z	.

A		G	Z	A	C	d	$
	.	.	.	d		d	.
G	.	.	.	.	.	C	.
Z	.	.	.	.	.	C	.
A	C	.	.	.	.	.	.
C	.	.	C	.	.	A	C
d	A	C	C	.	.	d	.
$	.	.	.	d	C	.	.

C		G	Z	A	C	d	$
	.	.	.	.	.	d	.
G	.	C	.	C	C	.	.
Z	.	.	.	.	C	Z	.
A	.	C	.	C	.	Z	.
C	.	C	C	.	F	d	F
d	d	.	Z	.	d	d	.
$	.	.	.	A	F	C	.

d		G	Z	A	C	d	$
	.	C		A		C	.
G	C	.	.	.	.	.	.
Z		.	C		A	C	
A	d	.		G	.	G	.
C		.	A	.			.
d	d	.	C	G		.	.
$	.	.		.		.	.

Table 2.10: Transition table for a solution with 7 states
Note: The character '.' represents an unused transition in the table.

2.5.7 Thickness

This solution is in another subclass called *threadlike solutions of thickness 2*.

2.6 Another 7-states solution

In section 2.3, we have noticed that two options are possible for a split. If we try to implement (naïvely) a solution we need 14 states. The 14th one is a special state named 'general', denoted `G` in figure 2.4, used to mark the *virtual* end of line after a cut. This solution seems to be very interesting because we only have to wait for all cells to become 'general'[3] before ordering cells to synchronize. All cells in state 'general' means that we have split the entire line in pieces of length 1. This is illustrated with figure 2.4.

Optimizations used here are those of section 2.5, except that the third auxiliary state can be suppressed too, because we can use the 'general' to replace it. This solution is in the subclass of *threadlike solutions of thickness* 2. The transition tables are given in table 2.11.

[3] this produces a *pre*-synchronization.

1	2	3	4	5	6	7	8	9	10	11	12	13	
Z													0
d	Z												1
C	d	Z											2
G	A	d	Z										3
G	A		d	Z									4
G	C			d	Z								5
G	d	A			d	Z							6
G	A	A				d	Z						7
G	d	C					d	Z					8
G		d	A					d	Z				9
G		A	A						d	Z			10
G		d	C							d	Z		11
G			d	A							d	Z	12
G			A	A								Z	13
G			d	C							Z	d	14
G				d	A					Z	d		15
G				A	A				Z	d			16
G				d	C			Z	d				17
G					d	A	Z	d					18
G					A	Z	Z						19
G					Z	d	d	Z					20
G				Z	d	C	C	d	Z				21
G			Z	d	A	G	G	A	d	Z			22
G		Z	d		A	G	G	A		d	Z		23
G	Z	d			C	G	G	C			d	Z	24
G	Z			A	d	G	G	d	A			Z	25
G	d	Z		d	d	G	G	A	A		Z	d	26
G		d	Z	C	d	G	G	d	C	Z	d		27
G			Z	A		G	G		A	Z			28
G		Z	d	Z		G	G		Z	d	Z		29
G	Z	d	C	d	Z	G	G	Z	d	C	d	Z	30
G	Z	A	G	A	Z	G	G	Z	A	G	A	Z	31
G	Z	Z	G	Z	Z	G	G	Z	Z	G	Z	Z	32
G	G	G	G	G	G	G	G	G	G	G	G	G	33
F	F	F	F	F	F	F	F	F	F	F	F	F	34

Figure 2.4: Execution of an LCA with 7 states and 13 cells

		G	Z	A	C	d	$
			Z		A		
G		.	Z		.		.
Z	Z	Z	.	Z	.	Z	Z
A			Z	.	.		.
C	A	.	.	.	.	.	.
d			.		.	.	
$	.	.	.	.	.	.	.

G		G	Z	A	C	d	$
	G	G	.	.	.	.	.
G	G	F	G	G	G	G	F
Z	.	G	G	.	.	d	.
A	.	G	.	G	.	.	.
C	.	G	.	.	G	.	.
d	.	G	.	G	.	G	.
$	G	F	G	G	G	G	.

Z		G	Z	A	C	d	$
	d	d	.	d	.	d	d
G	d	.	G	Z	G	Z	.
Z	d	G	.	.	.	.	G
A	d	Z	d	.	.	Z	Z
C	.	G	.	.	.	Z	G
d	d	Z	.	Z	Z	.	Z
$	d	.	.	.	.	.	F

A		G	Z	A	C	d	$
	.	C	Z	d	.	d	.
G	C	.	Z	d	.	A	.
Z	Z	Z	.	.	.	G	.
A	C	.	.	.	.	.	.
C	.	.	.	.	.	.	.
d	A	A	Z	.	.	.	.
$	.	.	.	.	.	.	.

C		G	Z	A	C	d	$
	.	d	.	.	.	d	.
G	d	.		.	.	.	.
Z	.		G	.	G	A	.
A	.	.	.	.	.	.	.
C	.	.	G	.	.	G	.
d	d	.	A	.	G	G	.
$	.	.	G	.	.	G	.

d		G	Z	A	C	d	$
	.	.		A		C	.
G	Z	Z		A		.	.
Z			C		A	C	
A	d	d		.	.	.	.
C			A	.	.	.	.
d	d	d	C	.	.	.	.
$	.	.	C	.	.	.	.

Table 2.11: Transition table for another solution with 7 states
Note : The character '.' represents an unused transition in the table.

2.6.1 The synchronization time

Let $\mathcal{L}(n)$ be the common length of the two *sub-lines* created after a split :

$$n = 2p \Longrightarrow \mathcal{L}(n) = \frac{n}{2} \tag{2.10}$$

$$n = 2p + 1 \Longrightarrow \mathcal{L}(n) = \frac{n-1}{2} \tag{2.11}$$

$$(2.10) \wedge (2.11) \Longrightarrow \mathcal{L}(n) = \lfloor \frac{n}{2} \rfloor \tag{2.12}$$

So equations (2.3) and (2.12) give the equation :

$$\mathcal{TS}(n) = \lfloor \frac{3n}{2} \rfloor + \mathcal{TS}(\lfloor \frac{n}{2} \rfloor), \mathcal{TS}(1) = 2 \tag{2.13}$$

We have this solution :

$$\mathcal{TS}(n) = 3n - 2\eta''(n) + \epsilon'(n) + \mathcal{TS}(1) - 5 \tag{2.14}$$

Where :

- $\eta''(n)$ be the number of 1's in the $\lfloor \log \frac{n}{3} \rfloor$ last digits of the binary representation of the number n.

- $\epsilon'(n) = \begin{cases} 0 \text{ if } n \in \bigcup_k [2^k - 2^{k-1}, 2^k[\\ 1 \text{ if } n \in \bigcup_k [2^k, 2^k + 2^{k-1}[\end{cases}$

Note:

$$\epsilon(n) = \begin{cases} 1 \text{ if } n - 2^{\lfloor \log n \rfloor} < 2^{\lceil \log n \rceil} - n \text{ or } \log n \in N \\ 0 \text{ if } n - 2^{\lfloor \log n \rfloor} \geq 2^{\lceil \log n \rceil} - n \text{ and } \log n \notin N \end{cases}$$

This solution has a time function really different from that given by equations (2.8) and (2.9), however all are in $3n + O(\log n)$.

2.7 Verifying Balzer's conditions

In is article [3] enumerate some conditions that (he think) the automaton must verify to synchronize:

1. The solution must be an image solution: There exists a function $\mathcal{I}$ such that:

   $\mathcal{I}(\mathcal{Q}) = \mathcal{Q}$ and $\forall\, x \in \mathcal{Q}\,,\, \mathcal{I}(\mathcal{I}(x)) = x$

   and $\forall\, x, y, z \in \mathcal{Q}^3$, if $(x, y, z) \rightarrow (t)$ then $(\mathcal{I}(z), \mathcal{I}(y), \mathcal{I}(x)) \rightarrow (\mathcal{I}(t))$
2. There exists a resident state R: $\forall\, x, y \in \mathcal{Q}^2 - \{(R, R)\}$, $(x, R, y) \rightarrow (R)$
3. There exists a pre-synchronization state S: $\forall\, x, y \in \{\$, S\}^2$, $(x, S, y) \rightarrow (\mathtt{F})$
4. There must be a dominant state D: $\forall\, x \in \mathcal{Q} - \{D\}$, $(D, x, D) \rightarrow (D)$

We can see that if a state `X` is resident then it's the only one that can be possibly dominant.

The first seven state solution is not an image solution and it's easy to verify in figures 2.1, 2.2 and 2.3 that our solution do not have a resident state. But the state `C` is a pre-synchronization state and dominant (see table 2.10).

The second seven state solution is not an image solution but state `G` is a pre-synchronization state and a resident state, and there is no dominant state.

2.8 Some mathematical results

Let :

- N be the set of integers.
- n be any arbitrary integer.
- $Bin(n)$ be the binary representation of n.
- $Bin_{last(d)}(n)$ be the last d digits of the binary representation of n, this equals to $Bin(n\ mod\ 2^d)$.
- w be any word in Σ^*, where $\Sigma = \{0, 1\}$.
- $\sharp_0(w)$ be the number of 0's in w.
- $\sharp_1(w)$ be the number of 1's in w.
- $\sharp_{ne0}(wx)$ be $\left\{ \begin{array}{l} \sharp_0(w) \text{ if } x = 1 \\ \sharp_{ne0}(w) \text{ if } x = 0 \end{array} \right.$ ($ne0$ means non-endings 0).
- $\lambda(xw)$ be w if $x = 0$, xw if $x = 1$ (λ holds significants digits).
- $\rho(wx)$ be $\rho(w).1$ if $x = 0$, $w.0$ if $x = 1$ (ρ transforms n into $n - 1$).

Lemma 1 *The sequence* $(\lfloor \frac{n}{2^i} \rfloor)_{i=0,\ldots,\lfloor \log n \rfloor}$ *(which is* $n, \lfloor \frac{n}{2} \rfloor, \ldots, \lfloor \frac{n}{2^{\lfloor \log n \rfloor}} \rfloor = 1$*) contains :*

- $1 + \lfloor \log \frac{n}{k} \rfloor$ *terms* $\geq k$.
- $\sharp_0(Bin_{last(1+\lfloor \log \frac{n}{k} \rfloor)}(n))$ *even terms* $\geq k$.
- $\sharp_1(Bin_{last(1+\lfloor \log \frac{n}{k} \rfloor)}(n))$ *odd terms* $\geq k$.

Proof

Observe that $\lfloor \frac{n}{2^i} \rfloor \geq k$ iff $\frac{n}{2^i} \geq k$ iff $i \leq \lfloor \log \frac{n}{k} \rfloor$. Thus there is $1 + \lfloor \log \frac{n}{k} \rfloor$ terms $\geq k$. Since :

- The parity of $\lfloor \frac{n}{2^i} \rfloor$ is that of its last digit.
- $Bin(\lfloor \frac{n}{2^i} \rfloor)$ is obtained by deleting the ith last digit in $Bin(n)$.
- The family $(last\ digit\ of\ \lfloor \frac{n}{2^i} \rfloor)_{i=0,\ldots,\lfloor \log \frac{n}{k} \rfloor}$ is exactly the family of the $1 + \lfloor \log \frac{n}{k} \rfloor$ last digits of n, ie the family of digits in $Bin_{last(1+\lfloor \log \frac{n}{k} \rfloor)}(n)$.

We see that there are :

- $\sharp_0(Bin_{last(1+\lfloor \log \frac{n}{k} \rfloor)}(n))$ even terms $\geq k$

- $\sharp_1(Bin_{last(1+\lfloor \log \frac{n}{k} \rfloor)}(n))$ odd terms $\geq k$.

Lemma 2 *The sequence* $(\lceil \frac{n}{2^i} \rceil)_{i=0,\ldots,\lceil \log n \rceil}$ *(which is* $n, \lceil \frac{n}{2} \rceil, \ldots, \lceil \frac{n}{2^{\lceil \log n \rceil}} \rceil = 1$*) contains :*

- $1 + \lceil \log n \rceil$ *terms.*
- $\sharp_{ne0}(Bin(n)) + min(2, \sharp_1(Bin(n)))$ *odd terms.*
- $1 + \lceil \log n \rceil - \sharp_{ne0}(Bin(n)) + min(2, \sharp_1(Bin(n)))$ *even terms.*

Proof

It suffices to count the odd terms.

Let $\phi(n) = \sharp_{ne0}(Bin(n)) + min(2, \sharp_1(Bin(n)))$.

Let $\Phi(n) = Card\{j \in [0, \lceil \log n \rceil] / \lceil \frac{n}{2^j} \rceil \text{ } is \text{ } odd\}$.

We show that $\Phi(n) = \phi(n)$.

- if $n = 1$ it's trivial.
- if it's true for n we show that it is true for $2n$.
  $Bin(2n) = Bin(n).0$ imply that $Bin(2n)$ have as many *non-endings* 0's and as many 1's as $Bin(n)$.
  Thus $\phi(2n) = \phi(n)$.
  Since $\lceil \frac{2n}{2} \rceil = n$ and $2n$ are even we have :

  $$\{\lceil \frac{2n}{2^j} \rceil, j \in [0, \lceil \log 2n \rceil]\} = \{\lceil \frac{n}{2^j} \rceil, j \in [0, \lceil \log n \rceil]\} \cup \{2n\}$$

  then

  $$\Phi(2n) = \Phi(n)$$

- if it's true for $n \geq 2$ we show that it is true for $2n - 1$.
  $Bin(2n-1) = \lambda(\rho(Bin(n).0))$:
  - $n = 2^p (p \geq 1)$:
    $Bin(n) = 1_p 0_{p-1} \ldots 0_1 0_0 \Rightarrow Bin(2n-1) = 1_p 1_{p-1} \ldots 1_1 1_0$.
    So we have :
    * $min(2, \sharp_1(Bin(2n-1))) = 1 + min(2, \sharp_1(Bin(n)))$
    * $\sharp_{ne0}(Bin(2n-1)) = \sharp_{ne0}(Bin(n))$

    Thus $\phi(2n-1) = \phi(n) + 1$.
  - $n \neq 2^p$:
    If $Bin(n) = 1_p x \ldots x 1_{j+1} 0_j \ldots 0_1 0_0$
    then $Bin(2n-1) = 1_{p+1} x \ldots x 0_{j+2} 1_{j+1} 1_j \ldots 1_1 1_0$.
    So, we have :
    * $min(2, \sharp_1(Bin(2n-1))) = min(2, \sharp_1(Bin(n)))$

$* \ \sharp_{ne0}(Bin(2n-1)) = 1 + \sharp_{ne0}(Bin(n))$

Thus $\phi(2n-1) = \phi(n) + 1$.

Since $\lceil \frac{2n-1}{2} \rceil = n$ and $2n-1$ is odd we have:

$$\{\lceil \frac{2n-1}{2^j} \rceil, j \in [0, \lceil \log(2n-1) \rceil]\} =$$

$$\{\lceil \frac{n}{2^j} \rceil, j \in [0, \lceil \log n \rceil]\} \cup \{2n-1\}$$

then

$$\Phi(2n-1) = 1 + \Phi(n)$$

Fact 1 *$\forall n \in N, \forall k \in [2, n], Card\{j \in [0, \lceil \log n \rceil] / \lceil \frac{n}{2^j} \rceil \geq k$ and odd $\} = \sharp_1(Bin_{last(\lceil \log \frac{n}{k-1} \rceil)}(2^t - n))$, for any $t > \log 3n$.*

Proof

For the following, let t being any integer $> \log 3n$.

- if $x \in N$, then $x - \lceil \frac{n}{2^i} \rceil = \lfloor x - \frac{n}{2^i} \rfloor$ (because $\forall z, \lfloor -z \rfloor = -\lceil z \rceil$).

  $$\begin{aligned} \lfloor \tfrac{2^t-n}{2^i} \rfloor \text{ is odd} &\iff \lfloor \tfrac{2^t-n}{2^i} \rfloor + 2^t - \tfrac{2^t}{2^i} \text{ is odd} \\ &\iff \lfloor 2^t - \tfrac{n}{2^i} \rfloor \text{ is odd} \\ &\iff 2^t - \lceil \tfrac{n}{2^i} \rceil \text{ is odd} \\ &\iff \lceil \tfrac{n}{2^i} \rceil \text{ is odd.} \end{aligned}$$

  Now we can use this result to transform a $\lceil \ \rceil$-sequence into an equivalent $\lfloor \ \rfloor$-sequence.

- $$\begin{aligned} \lceil \tfrac{n}{2^i} \rceil \geq k &\iff i \leq \lceil \log \tfrac{n}{k-1} \rceil - 1 \\ &\iff \lfloor \tfrac{2^t-n}{2^i} \rfloor \geq \lfloor \tfrac{2^t-n}{2^{\lceil \log \frac{n}{k-1} \rceil - 1}} \rfloor. \end{aligned}$$

  So the number of odd terms $\geq k$ in the sequence $(\lceil \frac{n}{2^i} \rceil)_{i=0,\ldots,\lceil \log n \rceil}$ equals the number of odd terms $\geq \lfloor \frac{2^t-n}{2^{\lceil \log \frac{n}{k-1} \rceil - 1}} \rfloor$ in the equivalent $\lfloor \ \rfloor$-sequence $(\lfloor \frac{2^t-n}{2^i} \rfloor)_{i=0,\ldots,\lfloor \log(2^t-n) \rfloor}$.

  That number equals to (see Lemma 1):

  $$\sharp_1(Bin_{last(1+\lfloor \log \frac{P}{\lfloor \frac{P}{2^Q} \rfloor} \rfloor)}(2^t - n))$$

  with $P = 2^t - n$ and $Q = \lceil \log \frac{n}{k-1} \rceil - 1$.

- We have $\lfloor \log \frac{P}{\lfloor \frac{P}{2^Q} \rfloor} \rfloor = \lfloor \log P - \log \lfloor \frac{P}{2^Q} \rfloor \rfloor$ and

  $$\begin{array}{rcccl} \log(\frac{P}{2^Q} - 1) & < & \log \lfloor \frac{P}{2^Q} \rfloor & \leq & \log \frac{P}{2^Q} \\ \log(\frac{P}{2^Q}) - Q & < & \log \lfloor \frac{P}{2^Q} \rfloor & \leq & \log P - Q \end{array}$$

then

$$\begin{array}{lllll} Q & \leq & \log P - \log \lfloor \frac{P}{2^Q} \rfloor & < & \log P - \log(P - 2^Q) + Q \\ Q & \leq & \log P - \log \lfloor \frac{P}{2^Q} \rfloor & < & \log(\frac{1}{1-\frac{2^Q}{P}}) + Q \end{array}$$

To have $\lfloor \log P - \log \lfloor \frac{P}{2^Q} \rfloor \rfloor = Q$ we need $\log \frac{1}{1-\frac{2^Q}{P}} \leq 1$ then it's sufficient to have:

$$\begin{aligned} \frac{1}{1-\frac{2^Q}{P}} \leq 2 &\iff 2^Q \leq \frac{P}{2} \\ &\iff 2^{Q+1} \leq P \\ &\iff 2^{\lceil \log \frac{n}{k-1} \rceil} \leq 2^t - n \\ &\iff 2^t \geq n + 2^{\lceil \log \frac{n}{k-1} \rceil} \\ &\Longrightarrow 2^t \geq n + 2^{\log \frac{n}{k-1} + \varepsilon} \\ &\Longrightarrow 2^t \geq n + 2^{\varepsilon} \frac{n}{k-1} \end{aligned}$$

So we can see that the condition $t > \log 3n$ suffices to verify the preceding equation (where $0 \leq \varepsilon < 1$).

Thus if $t > \log 3n$ then $1 + \lfloor \log \frac{P}{\lfloor \frac{P}{2^Q} \rfloor} \rfloor = 1 + \lceil \log \frac{n}{k-1} \rceil - 1 = \lceil \log \frac{n}{k-1} \rceil$.

Fact 2 *$\forall n \in N, \forall k \in [2, n], Card\{j \in [0, \lceil \log n \rceil] / \lceil \frac{n}{2^j} \rceil \geq k$ and even $\} = \sharp_0(Bin_{last(\lceil \log \frac{n}{k-1} \rceil)}(x - n))$, for any $t > \log 3n$.*

Proof

Lemma 1 and fact 1 prove this.

Lemma 3 *In the sequence $(\lceil \frac{n}{2^i} \rceil)_{i=0,\ldots,\lceil \log n \rceil}$ there is $1 + \lceil \log \frac{n}{k-1} \rceil$ terms $\geq k$, odd ones are $\sharp_1(Bin_{last(\lceil \log \frac{n}{k-1} \rceil)}(2^t - n))$, even ones are $\sharp_0(Bin_{last(\lceil \log \frac{n}{k-1} \rceil)}(2^t - n))$, for any $t > \log 3n$.*

Proof

See fact 1 and fact 2.

Chapitre 3

Une approche ascendante

Historique

Ce travail a comme point de départ le résultat de Balzer [3, 2] concernant l'impossibilité de construire une solution au FSSP avec un automate à 4 états internes seulement.

Ayant obtenu une solution minimale (en temps) avec 8 états il chercha à prouver que sa solution était un optimum en nombre d'états. Cette preuve pouvait prendre selon lui deux formes :

- la première pourrait être de type *analytique*, c'est à dire qu'il s'agirait de prouver que certaines fonctions de base doivent être réalisées afin qu'un automate soit solution au problème. L'ennui est qu'aujourd'hui nous ne voyons toujours pas comment construire une telle preuve.
- la deuxième approche serait une preuve par *exhaustion*. Autrement dit montrer qu'aucune des machines avec moins de 8 états ne synchronise.

Ainsi Balzer construisit un programme qui, étant donné un nombre d'états, permet de vérifier s'il existe ou non une machine synchronisant en temps minimal. Son logiciel lui permit d'affirmer qu'avec 4 états il était impossible de résoudre le FSSP. Malheureusement le nombre de machines comportant 5 états l'empêcha d'obtenir un résultat. En effet il y a $n^{(n-1)n^2-(n-1)-3}$ ensembles de transitions possibles avec n états :

$n-1$ tables : Nous n'avons pas besoin de regarder les transitions de la forme $(x,\mathtt{F},y)\rightarrow(z)$, puisqu'elles sont interdites.

n^2 transitions par table : Nous devons enlever toutes les transitions de la forme $(\mathtt{F},x,y)\rightarrow(z)$ et $(x,y,\mathtt{F})\rightarrow(z)$ (elles sont aussi interdites et il y en a $(n-1)(2n-1)$) mais nous devons tenir compte du fait que la ligne est finie et qu'il existe un état de bord. Il faut alors rajouter toutes les transitions du type $(\$,x,y)\rightarrow(z)$ et $(x,y,\$)\rightarrow(z)$, il y en a aussi $(n-1)(2n-1)$.

$n-1$: Nous ne devons pas compter les transitions qui concernent la ligne de longueur 1, puisque sa synchronisation est triviale. Il faut donc enlever $n-1$ transitions (toutes celles du type `($,`x`,$)`$\rightarrow$`(`z`)`).

3 : Il ne faut pas compter non plus les transitions *latentes*, puisqu'elles sont fixées : `($,L,L)`$\rightarrow$`(L)`, `(L,L,L)`$\rightarrow$`(L)` et `(L,L,$)`$\rightarrow$`(L)`.

n	Nombre d'automates
3	1594323
4	$1,9 \times 10^{25}$
5	$1,0 \times 10^{65}$
6	$6,9 \times 10^{133}$
7	$7,1 \times 10^{240}$
8	$3,5 \times 10^{395}$

TAB. 3.1 - *Nombre d'automates avec n états*

3.1 Un nouveau programme de *Balzer*

Nous avons réécrit[1] un programme équivalent à celui de Balzer (voir annexe E) permettant de vérifier qu'il n'existe pas d'automate synchronisant en temps minimal avec un nombre d'états fixé à l'avance.

Le calcul avec 4 états nécessite environ 153 secondes de temps `CPU` sur une machine `HP 750` de type `RISC` pour balayer 23.925.498 automates[2]. L'éxécution du programme permet de remarquer qu'il a suffi de ne regarder que les lignes d'automates dont la longueur est comprise entre 2 et 9. Autrement dit il est possible de construire des automates à 4 états synchronisant toutes les lignes de longueur comprise entre 2 et 8 (il en existe 27 différents) mais aucun d'entre eux n'est capable de synchroniser en même temps la ligne de longueur 9. Le tableau 3.1 montre que l'explosion combinatoire de l'espace de recherche nous empêche d'attendre le résultat pour les automates avec 5 états.

Le tableau 3.2 résume la situation, et le problème d'existence d'une solution de synchronisation en temps minimal à 5 états internes est un problème ouvert depuis [20].

1. Le but premier était de voir si il était possible aujourd'hui d'obtenir un résultat pour les automates à 5 états.

2. Il s'agit d'un comptage dans lequel on ne tient pas compte des instructions inutiles par lesquelles l'automate ne passera jamais lors de ses évolutions.

Toutes les lignes	Temps minimal
3 états	**NON** ([2], Yunès)
4 états	**NON** ([2], Yunès)
5 états	ouvert
6 états	**OUI** ([20])
7 états	**OUI** ([20])
8 états	**OUI** ([3], [20])

TAB. 3.2 - *L'état du problème*

3.2 Un programme de *Balzer* modifié

Nous nous sommes alors intéressés à l'existence de solutions au FSSP avec peu d'états mais en temps non-minimal. C'est ce que nous avons appelé solutions en temps quelconque.

Définition 3.1 *On appelle solution en* temps quelconque *tout automate cellulaire résolvant le problème de la ligne de fusiliers pour lequel la fonction du temps de synchronisation d'une ligne de longueur n est : $n-1 \leq \mathcal{T}(n) \leq e^n - 2$ (si $e = |\mathcal{Q}|$)*

Remarque: On peit voir qu'il n'y a qu'un nombre fini de n pour lesquels de temps de synchronisation est $n-1 \leq \mathcal{T}(n) < 2n-2$.

L'algorithme a donc été transformé afin de rechercher de tels automates (voir annexe F). Il est possible d'obtenir un résultat avec les automates à 3 états (il n'en existe pas un capable de synchroniser en temps quelconque) mais l'algorithme ne permet pas d'avoir une réponse pour ceux avec 4 états, car il faut non seulement vérifier une quantité importante d'automates mais pour chacun d'eux le diagramme espace-temps à parcourir est de longueur potentiellement exponentielle. Le tableau 3.3 résume la situation concernant ce type de solutions.

Toutes les lignes	Temps quelconque
3 états	**NON** (Yunès)
4 états	ouvert
5 états	ouvert
6 états	ouvert
7 états	**OUI** (Yunès)
8 états	**OUI** (Yunès)

TAB. 3.3 - *L'état du problème*

Nous avons donc voulu regarder quels étaient les automates qui empêchaient le calcul de terminer rapidement, autrement dit si il était possible d'éliminer rapidement des candidats sachant qu'ils ne peuvent être des solutions mais pour lesquels le programme est amené à faire de longs calculs inutiles.

Notations

La suite du travail utilise diverses notations et définitions que nous donnons ici :

- On note N l'ensemble des entiers naturels ($\mathsf{N}^\star = \mathsf{N} - \{0\}$).

- On note Z l'ensemble des entiers relatifs ($\mathsf{Z}^\star = \mathsf{Z} - \{0\}$).

- On note $\mathcal{P}$ l'ensemble des nombres premiers.

- On considèrera que l'ensemble des états est ordonné totalement. Cela permettra d'identifier pour des raisons de commodité chaque état avec son rang dans l'ensemble ordonné. Exemple : Si $\mathcal{Q} = \{q_0, q_1, \ldots, q_n\}$ est l'ensemble des états, et si l'ordre donné est $q_i < q_j$ pour tout $i < j$ alors on identifiera q_i et i. L'état quiescent sera toujours l'état de rang 0, et l'état de feu celui de rang n.

- $\binom{n}{p}$ le coefficient binômial égal à $\frac{n!}{p!(n-p)!}$. Dans la suite nous conviendrons de considérer que $\binom{n}{p} = 0$ si $n < p$.

- $\varphi(n)$ le nombre d'Euler associé au nombre n. Il s'agit du nombre d'entiers inférieurs à n et qui sont premiers avec lui (1 est premier avec tous les entiers). On sait que $\varphi(p) = p - 1$ si p est premier, et que $\varphi(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) = p_1^{\alpha_1 - 1} \ldots p_k^{\alpha_k - 1}(p_1 - 1) \ldots (p_k - 1)$. D'autre part on a $2^{\varphi(n)} \equiv 1 \bmod (n)$.

- $Ord_2(n) = min\{t \in \mathsf{N}^\star / 2^t \equiv 1 \bmod n\}$. Cette fonction est appellée ordre de n. On sait que $Ord_2(n) | \varphi(n)$.

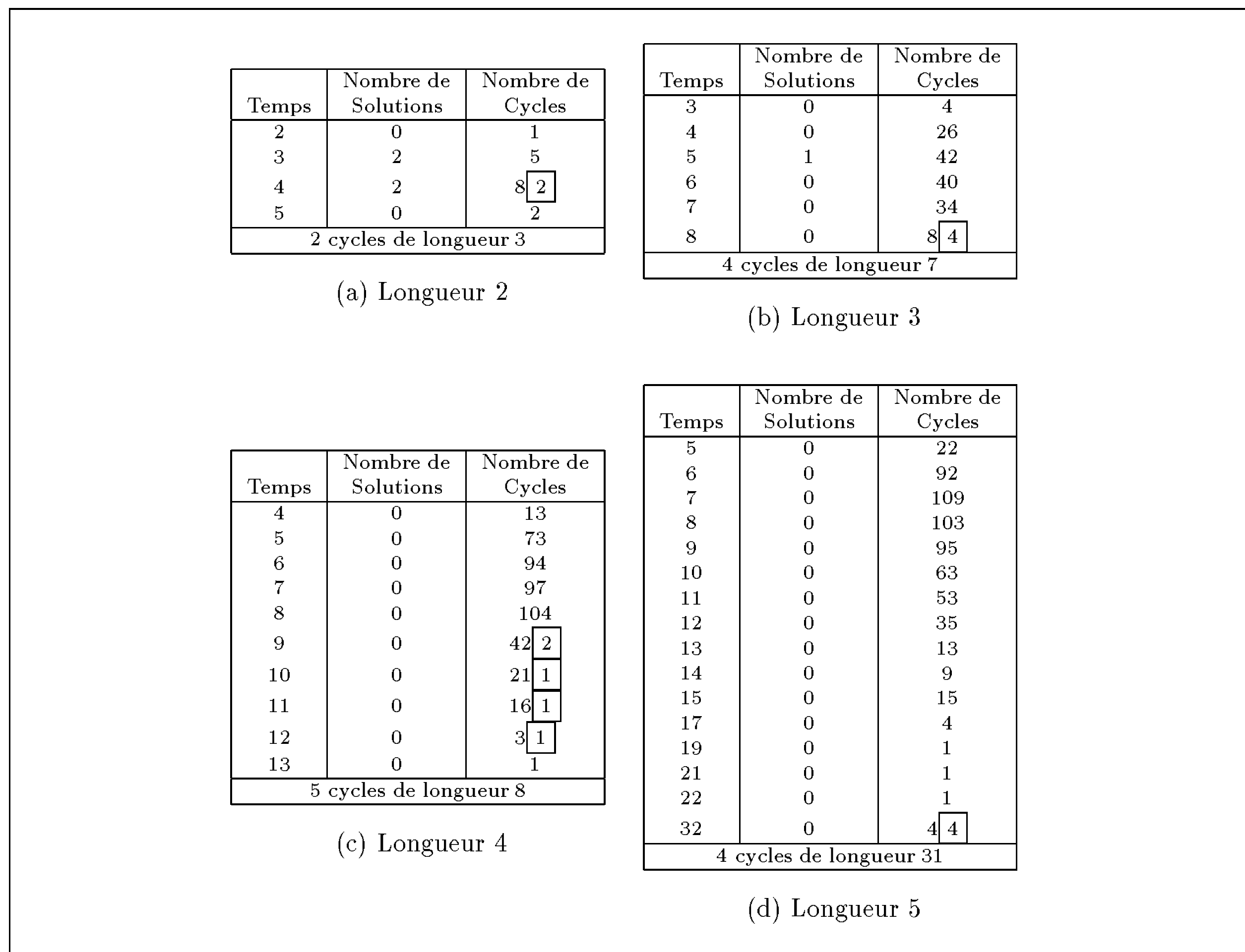
- $Sord_2(n) = min\{t \in \mathsf{N}^\star / 2^t \equiv \pm 1 \bmod n\}$. Cette fonction est appellée sous-ordre de n. On sait que $Sord_2(n) \in \{Ord_2(n), \frac{Ord_2(n)}{2}\}$ (attention même si $Ord_2(n)$ est pair il est possible d'avoir $Sord_2(n) = Ord_2(n)$, par exemple 63, mais évidemment si $Ord_2(n)$ est impair alors $Sord_2(n) = Ord_2(n)$).

- $\oplus$: L'addition modulo 2. Pour des raisons de commodité nous utiliserons parfois la notation usuelle $+$ pour désigner la même opération.

3.3 Observation des résultats du programme de Balzer modifié

Les tableaux de la figure 3.1 représentent :

- d'une part le nombre d'automates (modulo les instructions inutiles) conduisant à la synchronisation (voir la colonne "Nombre de Solutions").
- d'autre part le nombre d'automates pour lesquels n'apparaît pas l'état de "feu" dans leur calcul partant de la configuration initiale (voir la colonne "Nombre de Cycles" dans laquelle le nombre de cycles de longueur maximale détectés est encadré).



Temps	Nombre de Solutions	Nombre de Cycles
2	0	1
3	2	5
4	2	8 [2]
5	0	2
2 cycles de longueur 3		

(a) Longueur 2

Temps	Nombre de Solutions	Nombre de Cycles
3	0	4
4	0	26
5	1	42
6	0	40
7	0	34
8	0	8 [4]
4 cycles de longueur 7		

(b) Longueur 3

Temps	Nombre de Solutions	Nombre de Cycles
4	0	13
5	0	73
6	0	94
7	0	97
8	0	104
9	0	42 [2]
10	0	21 [1]
11	0	16 [1]
12	0	3 [1]
13	0	1
5 cycles de longueur 8		

(c) Longueur 4

Temps	Nombre de Solutions	Nombre de Cycles
5	0	22
6	0	92
7	0	109
8	0	103
9	0	95
10	0	63
11	0	53
12	0	35
13	0	13
14	0	9
15	0	15
17	0	4
19	0	1
21	0	1
22	0	1
32	0	4 [4]
4 cycles de longueur 31		

(d) Longueur 5

FIG. 3.1 - *Longueurs de cycles*

Définition 3.2 *On appelle* cycle de longueur maximale *le plus grand cycle calculable par ACLiF partant de la configuration initiale et pour un ensemble d'états fixé.*

Longueur de ligne	Longueur du plus grand cycle
2	3
3	7
4	8
5	31
6	63
7	63
8	34
9	511
10	511
11	2047

TAB. 3.4 - *Longueurs de cycles*

L'observation attentive du tableau 3.4 et de la figure 3.1 fait ressortir les faits suivants :

1. Pour la plupart des longueurs de ligne, la longueur du plus grand cycle constructible par ACLiF est de la forme $2^x - 1$ (les lignes de longueur 2, 3, 5, 6, 7, 9, 10 et 11 vérifient cette condition).
2. Pour un certain nombre de lignes (longueur n) la longueur du plus grand cycle calculé est exactement $2^n - 1$ (lignes de longueur 2, 3, 5, 6, 9 et 11). Cela signifie que l'on passe au cours de l'évolution par toutes les configurations possibles sauf la configuration nulle[3] car elle est stable (cycle de longueur 1).
3. Pour chaque ligne il n'y a que peu d'automates permettant d'obtenir des cycles de longueur maximale (en général il y en a 4).

Si l'on regarde les différents automates qui permettent de construire les cycles maximaux on remarque rapidement que 4 d'entre eux se retrouvent très fréquemment (voir tableau 3.5). Ces automates possèdent une fonction de transition linéaire, c'est à dire vérifiant :

$$\Delta(\mathcal{C}^n \oplus \mathcal{C}'^n) = \Delta(\mathcal{C}^n) \oplus \Delta(\mathcal{C}'^n)$$

Définition 3.3 *Soient $\mathcal{C}^n$ et $\mathcal{C}'^n$ deux configurations d'un ACLiF $\mathcal{A}$ dont l'ensemble des états comporte n éléments. On définit l'addition de deux configurations par l'opération :*

$$\mathcal{C}^n \oplus \mathcal{C}'^n = \mathcal{C}''^n$$

où $\mathcal{C}''^n$ est la configuration vérifiant :

$$\forall i \in [1, n], \mathcal{C}''^n(i) = \mathcal{C}^n(i) \oplus \mathcal{C}^n(i)$$

3. La configuration nulle est la configuration dans laquelle toutes les machines sont dans l'état latent (état *zéro*).

t+1	Xor1	Xor2	Xor3	Xor4
gmd	g⊕d	g⊕m⊕d	g⊕m⊕d	g⊕d
$md	d	d	m⊕d	m⊕d
gm$	g⊕m	g⊕m	g	g

TAB. 3.5 - *Les 4 automates à long cycle*

Remarque: On note:

g: l'état du voisin de gauche au temps t.

m: l'état de la cellule considérée au temps t.

d: l'état du voisin de droite au temps t.

3.4 La loi `Xor1`

Le tableau 3.6 et la figure 3.2 nous indiquent la longueur du cycle calculé étant donné la longueur de ligne de l'ACLiF calculant la loi `Xor1`. On peut remarquer plusieurs choses:

- Pour presque toutes les lignes, le cycle calculé est de la forme $2^p - 1$ (la première exception est la ligne de longueur 18, toutefois 87381 est un diviseur de $2^{18} - 1$).
- Pour un certain nombre de longueurs de ligne l le cycle fabriqué est de longueur $2^l - 1$:
  - que $2l + 1$ soit un nombre premier semble être une condition nécessaire pour que le cycle soit de longueur $2^l - 1$.
  - si l est un nombre premier ainsi que $2l + 1$ alors le cycle est de longueur $2^l - 1$ (voir les nombres encadrés).
- Pour les lignes de longueur $l = 2^p$ (resp. $l = 2^p - 1$) le cycle calculé est de longueur $2l = 2^{p+1}$ (resp. $2(l + 1) = 2^{p+1}$).
- Si l est un nombre premier on peut vérifier que la longueur du cycle est un diviseur de $2^l - 1$ (voir les nombres avec une astérisque).

  N.B. $255 = 15 * 17$, $32767 = 31 * 1057$, $262143 = 87381 * 3$, $1048575 = 1023 * 1025$, $2097151 = 127 * 16513$.
- Si $2l + 1$ n'est pas un nombre premier alors le cycle n'est pas de longueur maximale ($2^l - 1$).

Nous allons donc essayer de prouver ces différentes remarques.

l	$2l+1$	t	2^l
2	5*	4	4
3	7*	8	8
4	9	8	16
5	11*	32	32
6	13*	64	64
7	15	16	128
8	17*	16	256
9	19*	512	512
10	21	64	1024
11	23*	2048	2048
12	25	1024	4096
13	27	512	8192
14	29*	16384	16384
15	31*	32	32768
16	33	32	65536
17	35	4096	131072
18	37*	**87382**	262144
19	39	4096	524288
20	41*	1024	1048576
21	43*	128	2097152
22	45	4096	4194304
23	47*	8388608	8388608
24	49	2097152	16777216
25	51	256	33554432
26	53*	67108864	67108864
27	55	1048576	134217728
28	57	512	268435456
29	59*	536870912	536870912
30	61*	1073741824	1073741824
31	63	64	2147483648

TAB. 3.6 - *Longueur des cycles calculés par* `Xor1`

3.4.1 Equivalence avec un automate torique

Les automates cellulaires toriques (ACT[4]) dont les fonctions de transitions sont linéaires ont été étudiés par Wolfram et Jen (voir [37], [10] et [11]).

Construction d'un tore

Le problème considéré concernait les automates cellulaires linéaires de longueur finie. Nous avons donc trouvé un moyen de transformer le problème dans les ACLiFs en un problème dans les automates cellulaire toriques. En effet cela nous évite d'avoir à manipuler les *effets de bord* induit par les deux cellules du bout qui n'ont qu'un seul voisin.

Voici (exprimées formellement) les dépendances liant les sites du diagramme espace-temps

4. Car les cellules ne sont plus disposées en ligne mais sur un tore.

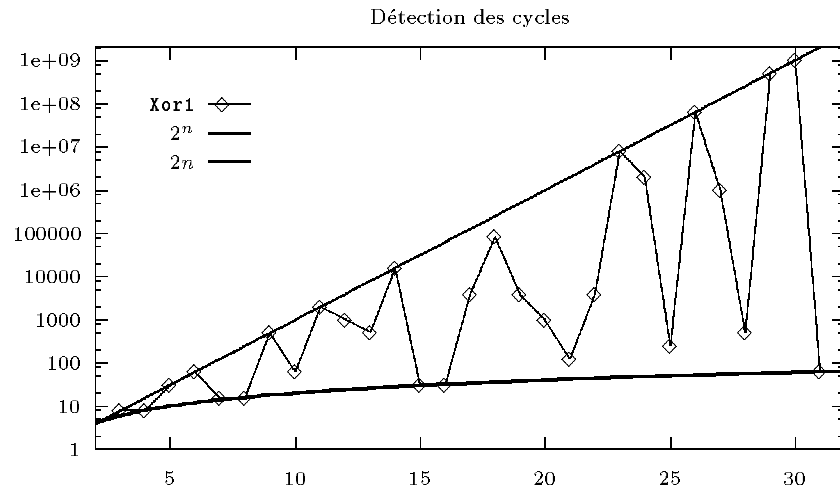


FIG. 3.2 - *Longueur des cycles calculés par* Xor1

d'un ACLiF :

$$\begin{cases} \begin{bmatrix} t+1 \\ 1 \end{bmatrix} = \begin{bmatrix} t \\ 2 \end{bmatrix} \\ \forall i \in]1, n[, \begin{bmatrix} t+1 \\ i \end{bmatrix} = \begin{bmatrix} t \\ i-1 \end{bmatrix} + \begin{bmatrix} t \\ i+1 \end{bmatrix} \\ \begin{bmatrix} t+1 \\ n \end{bmatrix} = \begin{bmatrix} t \\ n-1 \end{bmatrix} + \begin{bmatrix} t \\ n \end{bmatrix} \end{cases} \tag{3.1}$$

Pour faire l'étude comportementale de l'automate il faudrait supprimer les "effets de bords" ($\begin{bmatrix} t \\ 1 \end{bmatrix}$ et $\begin{bmatrix} t \\ n \end{bmatrix}$) afin "d'uniformiser" le calcul.
On a :

$$\begin{bmatrix} t+1 \\ 1 \end{bmatrix} = \begin{bmatrix} t \\ 2 \end{bmatrix} \Longleftrightarrow \begin{bmatrix} t+1 \\ 1 \end{bmatrix} = 0 + \begin{bmatrix} t \\ 2 \end{bmatrix}$$

Introduisons donc une cellule de numéro 0 telle que :

$$\forall t \in \mathsf{N}^\star, \begin{bmatrix} t \\ 0 \end{bmatrix} = 0 \tag{3.2}$$

Ainsi $\begin{bmatrix} t+1 \\ 1 \end{bmatrix} = \begin{bmatrix} t \\ 0 \end{bmatrix} + \begin{bmatrix} t \\ 2 \end{bmatrix}$. Or pour vérifier la condition de l'équation (3.2) cette cellule devrait avoir une voisine de numéro -1 dont l'état devrait être le même que l'état de la cellule de numéro 1 (*ie* $\begin{bmatrix} t \\ -1 \end{bmatrix} = \begin{bmatrix} t \\ 1 \end{bmatrix}$). Ainsi on aura :

$$\begin{bmatrix} t+1 \\ 0 \end{bmatrix} = \begin{bmatrix} t \\ -1 \end{bmatrix} + \begin{bmatrix} t \\ 1 \end{bmatrix} = 0$$

En réitérant ce procédé on peut ainsi créer n nouvelles cellules (de numéro compris entre $-n$ et 0) qui auront comme dépendances dans le diagramme espace-temps :

$$\begin{cases} \begin{bmatrix} t+1 \\ -n \end{bmatrix} = \begin{bmatrix} t \\ -n \end{bmatrix} + \begin{bmatrix} t \\ -n+1 \end{bmatrix} \\ \forall i \in]-n, n[, \begin{bmatrix} t+1 \\ i \end{bmatrix} = \begin{bmatrix} t \\ i-1 \end{bmatrix} + \begin{bmatrix} t \\ i+1 \end{bmatrix} \\ \begin{bmatrix} t+1 \\ n \end{bmatrix} = \begin{bmatrix} t \\ n-1 \end{bmatrix} + \begin{bmatrix} t \\ n \end{bmatrix} \end{cases} \tag{3.3}$$

On a $\begin{bmatrix} t \\ -n \end{bmatrix} = \begin{bmatrix} t \\ n \end{bmatrix}$ (ceci par construction) donc on peut réécrire l'équation (3.3) comme suit :

$$\begin{cases} \begin{bmatrix} t+1 \\ -n \end{bmatrix} = \begin{bmatrix} t \\ n \end{bmatrix} + \begin{bmatrix} t \\ -n+1 \end{bmatrix} \\ \forall i \in]-n, n[, \begin{bmatrix} t+1 \\ i \end{bmatrix} = \begin{bmatrix} t \\ i-1 \end{bmatrix} + \begin{bmatrix} t \\ i+1 \end{bmatrix} \\ \begin{bmatrix} t+1 \\ n \end{bmatrix} = \begin{bmatrix} t \\ n-1 \end{bmatrix} + \begin{bmatrix} t \\ -n \end{bmatrix} \end{cases} \tag{3.4}$$

En considérant les cellules $-n$ et n comme voisines, on obtient un automate cellulaire torique

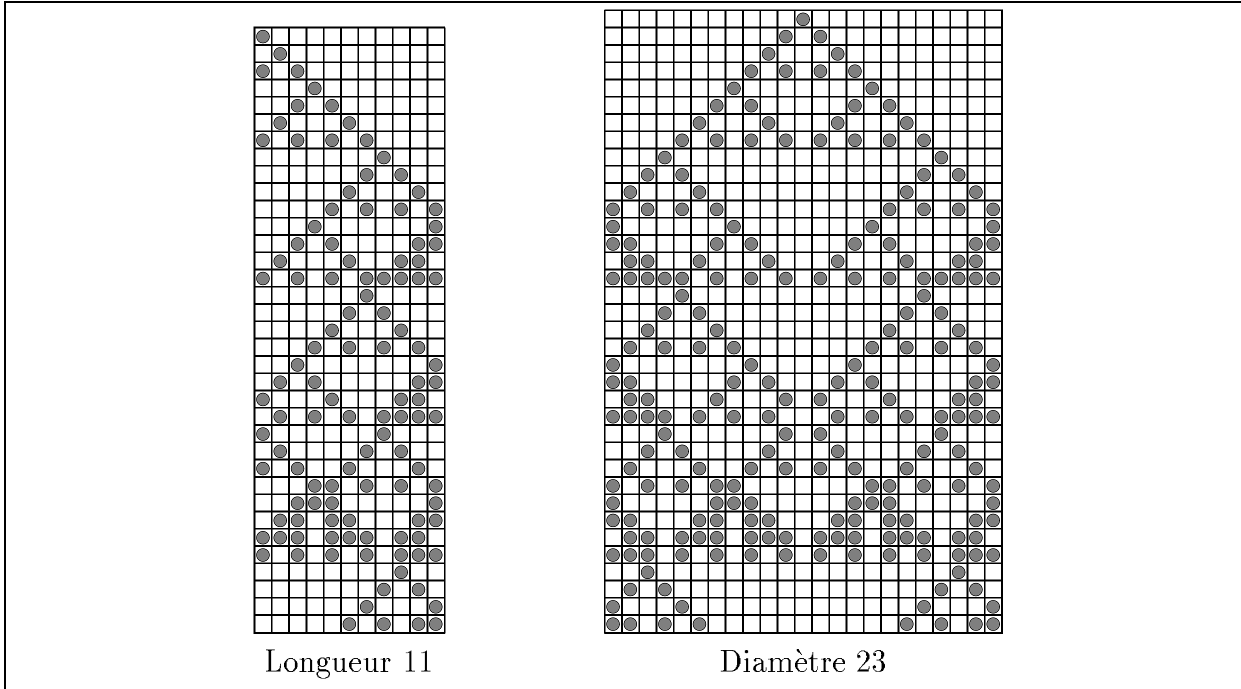


FIG. 3.3 - *Equivalence entre un ACLiF et un ACT*

de circonférence $2n+1$ dont la loi d'évolution (3.4) n'a plus de particularités aux bords. La trace des cellules 1 à n de l'évolution de cet automate torique (voir l'exemple de la figure 3.3) n'est autre que celle de l'automate défini par l'équation (3.1).

Initialisation du tore

Tel qu'il est décrit l'automate de l'équation (3.4) devrait avoir comme configuration de départ la configuration suivante (reflet miroir pour les cellules $-n$ à -1 par rapport aux cellules 1 à n) :

$$\begin{cases} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = 1 \\ \forall i \in [-n, -1[, \begin{bmatrix} 1 \\ i \end{bmatrix} = 0 \\ \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0 \\ \forall i \in [1, n[, \begin{bmatrix} 1 \\ i \end{bmatrix} = 0 \\ \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 1 \end{cases} \tag{3.5}$$

Or cette configuration n'est pas un jardin d'Eden et possède un antécédent au temps initial 0 :

$$\begin{cases} \left[{0 \atop 0}\right] = 1 \\ \forall i \in [-n, -1] \cup [1, n], \left[{0 \atop i}\right] = 0 \end{cases} \tag{3.6}$$

Extension infinitaire

On peut encore considérer que l'automate cellulaire torique à étudier (voir section précédente) est en fait un automate bi-infini à deux états et voisinage de diamètre 1. La configuration de départ est périodique de période $2n+1$ (c'est une extension infinitaire de (3.6)) :

$$\forall x \in \mathsf{Z},\ \left[{0 \atop x}\right] = \begin{cases} 1 \text{ si } \exists k \in \mathsf{Z},\ x = k\,(2n+1) \\ 0 \text{ sinon} \end{cases} \tag{3.7}$$

avec pour fonction de transition (extension infinitaire de (3.4)) :

$$\forall t \in \mathsf{N}^\star, \forall i \in \mathsf{Z},\ \left[{t \atop i}\right] = \left[{t-1 \atop i-1}\right] + \left[{t-1 \atop i+1}\right]$$

Définitions récurrentes

Tout d'abord quelques généralités peuvent être observées dans le comportement global de l'automate. Premièrement on peut voir que l'évolution de l'automate est symétrique par rapport à l'axe engendré par l'ensemble des sites médians (sites $\left[{t \atop 0}\right]$) du diagramme espace-temps et qu'il s'agit du développement d'un triangle de Pascal modulo 2 sur un tore. C'est la proposition suivante :

Proposition 3.4 *Deux faits :*

1. $\forall t \in \mathsf{N}, \forall i \in \mathsf{Z},\ \left[{t \atop i}\right] = \sum_{p=0}^{t} \binom{t}{p} \cdot \left[{0 \atop i-t+2p}\right] \pmod 2$
2. $\forall t \in \mathsf{N}, \forall i \in \mathsf{Z},\ \left[{t \atop -i}\right] = \left[{t \atop i}\right]$

Preuve:

1. $t = 0$: La proposition est vraie par définition.

   $t \neq 0$: Supposons que c'est vrai pour t et montrons que cela l'est pour $t+1$:

$$\begin{aligned} \left[{t+1 \atop i}\right] &= \left[{t \atop i-1}\right] + \left[{t \atop i+1}\right] \text{ par définition.} \\ \left[{t+1 \atop i}\right] &= \sum_{p=0}^{t} \binom{t}{p} \cdot \left[{0 \atop i-1-t+2p}\right] + \sum_{p=0}^{t} \binom{t}{p} \cdot \left[{0 \atop i+1-t+2p}\right] \text{ par hypothèse.} \\ \left[{t+1 \atop i}\right] &= \sum_{p=0}^{t} \binom{t}{p} \cdot \left[{0 \atop i-1-t+2p}\right] + \sum_{p=1}^{t+1} \binom{t}{p-1} \cdot \left[{0 \atop i+1-t+2p-2}\right] \end{aligned}$$

$$\begin{bmatrix} t+1 \\ i \end{bmatrix} = \binom{t}{0}.\begin{bmatrix} 0 \\ i-1-t \end{bmatrix} + \sum_{p=1}^{t}(\binom{t}{p-1} + \binom{t}{p}).\begin{bmatrix} 0 \\ i-1-t+2p \end{bmatrix} + \binom{t}{t}.\begin{bmatrix} 0 \\ i+t+1 \end{bmatrix}$$

$$\begin{bmatrix} t+1 \\ i \end{bmatrix} = \sum_{p=0}^{t+1}\binom{t+1}{p}.\begin{bmatrix} 0 \\ i-(t+1)+2p \end{bmatrix}$$

2. Il s'agit ici de démontrer la symétrie dans l'espace-temps du calcul de l'automate :

$$\begin{bmatrix} t \\ -i \end{bmatrix} = \sum_{p=0}^{t}\binom{t}{p}\begin{bmatrix} 0 \\ -i-t+2p \end{bmatrix} \text{ par définition.}$$

$$\begin{bmatrix} t \\ -i \end{bmatrix} = \sum_{q=0}^{t}\binom{t}{t-q}\begin{bmatrix} 0 \\ -i-t+2t-2q \end{bmatrix}$$

$$\begin{bmatrix} t \\ -i \end{bmatrix} = \sum_{q=0}^{t}\binom{t}{q}\begin{bmatrix} 0 \\ -i+t-2q \end{bmatrix}$$

$$\begin{bmatrix} t \\ -i \end{bmatrix} = \sum_{q=0}^{t}\binom{t}{q}\begin{bmatrix} 0 \\ i-t+2q \end{bmatrix} \text{ car } \begin{bmatrix} 0 \\ i \end{bmatrix} = \begin{bmatrix} 0 \\ -i \end{bmatrix} \text{ (voir équation (3.7))}$$

$$\begin{bmatrix} t \\ -i \end{bmatrix} = \begin{bmatrix} t \\ i \end{bmatrix}$$

□

D'autre part nous rappelons ici le Théorème de Lucas (1878) (voir [14]) :

Théorème 3.5 *Soit p premier, soient x et y deux entiers tels que $0 \leq y \leq x$. Si $\ldots x_2 x_1 x_0$ et $\ldots y_2 y_1 y_0$ sont les écritures de x et y en base p alors :*

$$\binom{x}{y} \equiv \ldots \binom{x_2}{y_2}\binom{x_1}{y_1}\binom{x_0}{y_0} \pmod p$$

La proposition suivante souligne une particularité du triangle de Pascal :

Proposition 3.6 $\forall t \in \mathsf{N}, \forall i \in \mathsf{Z}, \begin{bmatrix} 2^t \\ i \end{bmatrix} = \begin{bmatrix} 0 \\ i-2^t \end{bmatrix} + \begin{bmatrix} 0 \\ i+2^t \end{bmatrix}$

Preuve: D'après la Proposition 3.4, on a $\begin{bmatrix} 2^t \\ i \end{bmatrix} = \sum_{p=0}^{2^t}\binom{2^t}{p}.\begin{bmatrix} 0 \\ i-2^t+2p \end{bmatrix}$. En comparant les décompositions binaires de 2^t et i, et avec le Théorème de Lucas 3.5 on peut remarquer que :

$$\forall i \in]0, 2^t[\,, \binom{2^t}{i} = 0 \pmod 2 \wedge \binom{2^t}{0} = \binom{2^t}{2^t} = 1 \pmod 2$$

□

La proposition suivante exprime le fait que des configurations remarquables (ne contenant qu'une seule cellule active[5]) apparaissent en des temps tout aussi remarquables (des temps de la forme $t = 2^p$), voir la figure 3.4.

5. Non quiescente.

Temps	Configuration
1	●□□□□□□□□
2	□●□□□□□□□
4	□□□●□□□□□
8	□□□□□□□●□
16	□□●□□□□□□
32	□□□□□●□□□
64	□□□□□□●□□
128	□□□□●□□□□
256	□□□□□□□□●
512	●□□□□□□□□

(a) Longueur 9

Temps	Configuration
1	●□□□□□□□□□□□□□
2	□●□□□□□□□□□□□□
4	□□□●□□□□□□□□□□
8	□□□□□□□●□□□□□□
16	□□□□□□□□□□□□●□
32	□□●□□□□□□□□□□□
64	□□□□□●□□□□□□□□
128	□□□□□□□□□□□●□□
256	□□□□●□□□□□□□□□
512	□□□□□□□□□●□□□□
1024	□□□□□□□□●□□□□□
2048	□□□□□□□□□□●□□□
4096	□□□□□□●□□□□□□□
8192	□□□□□□□□□□□□□●
16384	●□□□□□□□□□□□□□

(b) Longueur 14

FIG. 3.4 - *Exemples de configurations atomiques*

Proposition 3.7 *Deux faits :*

1. $\forall n \in \mathsf{N}^\star$, $\forall t \in \mathsf{N}$, $\exists! i \in [1,n]$, $\left[{2^t \atop i}\right] = 1$ *: une telle configuration sera appellée configuration atomique.*
2. *Soit* $\lambda_n(t) = |[(2^t + n) \bmod (2n+1)] - n|$ *alors* $1 \leq \lambda_n(t) \leq n$ *et* $\left[{2^t \atop \lambda_n(t)}\right] = 1$

Preuve:

1. ● $\left[{2^t \atop i}\right] = \left[{0 \atop i-2^t}\right] + \left[{0 \atop i+2^t}\right]$ d'après la Proposition 3.6. Donc $\left[{2^t \atop i}\right] = 1$ ssi ou bien $\left[{0 \atop i-2^t}\right] = 1$ ou bien $\left[{0 \atop i+2^t}\right] = 1$ (de façon exclusive). Donc il n'est pas possible que les deux conditions soient réunies car $(i+2^t) - (i-2^t) = 2^{t+1} \not\equiv 0 \bmod (2n+1)$. Donc :

$$\left[{2^t \atop i}\right] = 1 \ ssi \begin{cases} \left[{0 \atop i-2^t}\right] = 1 \ ssi \ i - 2^t = k.(2n+1) \ ssi \ i = k.(2n+1) + 2^t \\ ou \\ \left[{0 \atop i+2^t}\right] = 1 \ ssi \ i + 2^t = k.(2n+1) \ ssi \ i = k.(2n+1) - 2^t \end{cases}$$

   On peut alors définir deux ensembles : $\mathcal{A} = 2^t + (2n+1)\mathsf{Z}$ et $\mathcal{B} = -2^t + (2n+1)\mathsf{Z}$. Ainsi $\left[{2^t \atop i}\right] = 1$ ssi $i \in \mathcal{A} \cup \mathcal{B}$.

   ● Dans l'intervalle $[-n, n]$ qui est de taille $2n+1$, il y a exactement un élément de $\mathcal{A}$ et un élément de $\mathcal{B}$ car $\mathcal{A}$ et $\mathcal{B}$ sont deux ensembles “périodiques” et de période $2n+1$.

- $\mathcal{A} \cap \mathcal{B} = \phi$ car $k.(2n+1) + 2^t = k'.(2n+1) - 2^t \iff (2n+1).(k'-k) = 2^{t+1}$. Mais $\forall\, n \in \mathsf{N}^\star$, $\forall\, t \in \mathsf{N}$, $(2n+1) \nmid 2^{t+1}$. Donc les deux points de l'intervalle $[-n, n]$ sont distincts.
- L'intervalle $[-n, n]$ est symétrique et la répartition de $\mathcal{A} \cup \mathcal{B}$ aussi puisque :

$$\forall\, k \in \mathsf{Z},\; k.(2n+1) + 2^t \in \mathcal{A} \iff -k.(2n+1) - 2^t \in \mathcal{B}$$

Ainsi les deux points de $\mathcal{A} \cup \mathcal{B}$ dans $[-n, n]$ sont symétriques par rapport au point milieu de l'intervalle. Il y en a un dans $[-n, -1]$ et un dans $[1, n]$.

2. • L'ensemble $\mathcal{A}$ s'exprime aussi $\mathcal{A} = [(2^t + n) + (2n+1)\mathsf{Z}] - n$, donc :

$$\begin{aligned} & [(2^t+n) \bmod (2n+1)] - n \in \mathcal{A} \\ \Longrightarrow\; & [(2^t+n) \bmod (2n+1)] - n \in [0, 2n] - n \\ \Longrightarrow\; & [(2^t+n) \bmod (2n+1)] - n \in [-n, n] \end{aligned}$$

Pour l'ensemble $\mathcal{B}$ s'exprimant $\mathcal{B} = n - [(2^t + n) + (2n+1)\mathsf{Z}]$ on a :

$$\begin{aligned} & n - [(2^t+n) \bmod (2n+1)] \in \mathcal{B} \\ \Longrightarrow\; & n - [(2^t+n) \bmod (2n+1)] \in n - [0, 2n] \\ \Longrightarrow\; & n - [(2^t+n) \bmod (2n+1)] \in [-n, n] \end{aligned}$$

Or on peut remarquer que :

$$|n - [(2^t+n) \bmod (2n+1)]| = |[(2^t+n) \bmod (2n+1)] - n| \in [1, n]$$

On notera désormais $\lambda_n(t) = |[(2^t+n) \bmod (2n+1)] - n| \in [1, n]$.

□

Nous avons donc démontré que pour tous les temps qui sont une puissance de deux (2^t) l'automate cellulaire est dans une configuration atomique pour laquelle le seul 1 présent est en position $\lambda_n(t)$. On peut déjà remarquer qu'il n'y a qu'un nombre fini de positions possibles pour les $\lambda_n(t)$ (il y en a exactement n). Mais cette suite des $\lambda_n(t)$ est une suite infinie, il existe donc $s, t \in \mathsf{N}^2$ / $0 \le s < t$ *et* $\lambda_n(s) = \lambda_n(t)$. Autrement dit il existe donc un cycle dont la longueur est un diviseur de $2^t - 2^s$.

Théorème 3.8 *Si a et m sont premiers entre eux alors* $a^{\varphi(m)} \equiv 1 \pmod{m}$.
Voir [7] page 63.

On obtient donc ici (d'une autre manière) le résultat de Wolfram *et al.* [37] affirmant que les cycles contiennent tous la configuration initiale :

Proposition 3.9 $\forall n \in \mathsf{N}^\star$, $\exists t \in \mathsf{N}$, $\lambda_n(t) = \lambda_n(0)$ *et la longueur de la période est un diviseur de* $2^{Sord_2(2n+1)} - 1$.

Preuve: $\lambda_n(0) = 1$.

Autrement dit pour un n donné existe-t'il un $t \neq 0$ tel que : $\lambda_n(t) = 1$?

$$\lambda_n(t) = 1 \iff \begin{cases} 2^t \bmod (2n+1) = 1 \\ 2^t \bmod (2n+1) = -1 \end{cases} \iff \begin{cases} 2^t \equiv 1 \bmod (2n+1) \\ 2^t \equiv -1 \bmod (2n+1) \end{cases}$$

Or nous pouvons appliquer le théorème 3.8 puisque 2 et $2n+1$ sont premiers entre eux. Ainsi il existe toujours un t qui vérifie l'équation de la proposition. Ce t est $\varphi(2n+1)$.

Note : Le théorème de Fermat-Euler ne nous indique que l'existence d'un tel t, pas le plus petit vérifiant l'équation (le plus petit étant $Sord_2(2n+1)$ par définition). □

Corollaire 3.10 *Pour tout n, l'évolution de l'automate* Xor1 *sur une ligne de n cellules est cyclique et sans phase transitoire.*

Nous venons de voir que non seulement il y a toujours un cycle mais en plus ce cycle contient la configuration de départ de l'automate fini (*ie* celle contenant un 1 en position $\lambda_n(0) = 1$). D'autre part nous savons que $\varphi(n)$ est du même ordre de grandeur que n ainsi les cycles sont de l'ordre de 2^n.

Une interprétation géométrique

Il est possible de visualiser géométriquement les valeurs successives de $\lambda_n(t)$. En effet si l'on définit la fonction :

$$f(x) = \begin{cases} 2n+1-2x & \text{si } 2x > n \\ 2x & \text{si } 2x \leq n \end{cases}$$

A partir de laquelle on définit la suite $\lambda_n(t) = f(f^{t-1}(1))$. Alors graphiquement on obtient les figures 3.5a et 3.5b. Les ordonnées successives des points d'intersection avec la droite $y = t$ donnent la suite des $\lambda_n(t)$, et l'on comprend "visuellement" que pour certaines valeurs de n on n'obtient pas tout les entiers compris entre 1 et n.

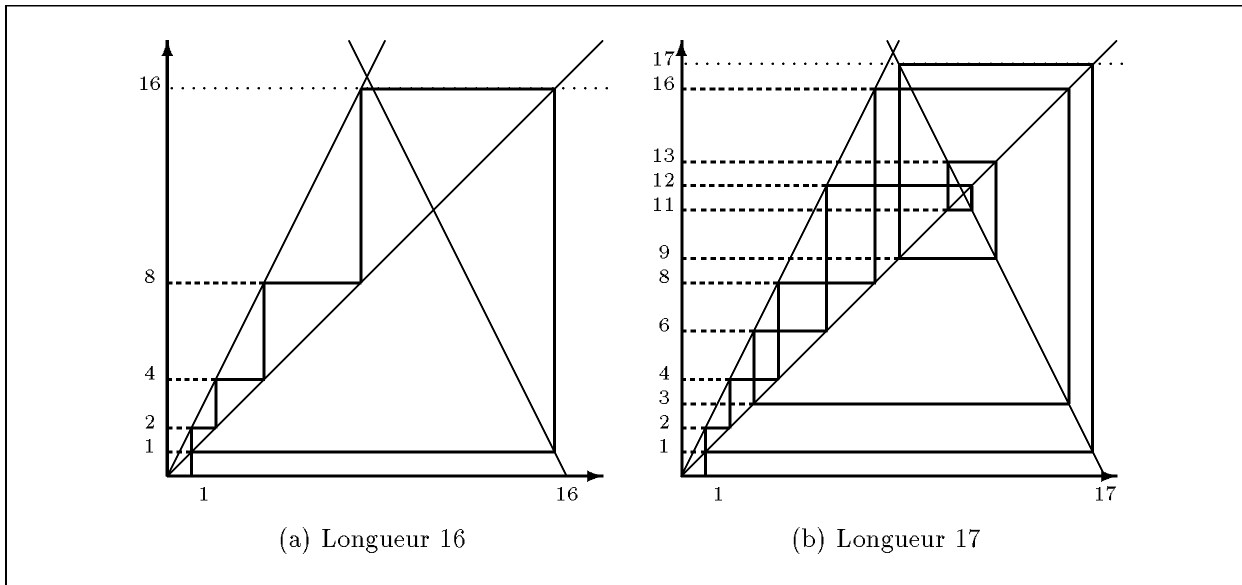
Un lien avec la théorie des nombres

Nous aimerions montrer qu'il existe une infinité de n pour lesquels la ligne de n ACLiFs calcule un cycle de longueur maximale $2^n - 1$. Une condition nécessaire (non suffisante) est que :

$$min\{t \in \mathsf{N}^\star / \lambda_n(t) = \lambda_n(0)\} = n \tag{3.8}$$

Conjecture 3.11 $Card\{n \in \mathsf{N} /\ n \in \mathcal{P}\ \wedge\ 2n+1 \in \mathcal{P}\ \wedge\ 2^n - 1 \in \mathcal{P}\} = \infty$

Proposition 3.12 $\forall n \in \mathcal{P},\ 2n+1 \in \mathcal{P} \Longrightarrow Sord_2(2n+1) = n$



(a) Longueur 16 (b) Longueur 17

FIG. 3.5 - *Itération géométrique*

Preuve: On sait que $\forall n \in \mathbb{N},\ Ord_2(n)|\varphi(n)$ et que $\forall n \in \mathbb{N},\ 2^{\varphi(n)} \equiv 1 \bmod n$. D'autre part $\forall n \in \mathcal{P},\ \varphi(n) = n - 1$.

Si $n \in \mathcal{P}$ et si $2n+1 \in \mathcal{P}$ on a $Ord_2(2n+1)|\varphi(2n+1) = 2n$. Or $2n$ ne possède que deux diviseurs : 2 et n, donc

- Ou bien $Ord_2(2n+1) = 1$ et on a :

$$2^1 \equiv 1 \bmod (2n+1) \Longrightarrow n = 0$$

- Ou bien $Ord_2(2n+1) = 2$ et on a :

$$2^2 \equiv 1 \bmod (2n+1) \Longrightarrow n = 1$$

- Ou bien $Ord_2(2n+1) = n$ car :

$$\begin{aligned} & 2^{\varphi(2n+1)} \equiv 1 \bmod (2n+1) \\ \Longleftrightarrow\ & 2^{2n} \equiv 1 \bmod (2n+1) \\ \Longleftrightarrow\ & 2^{n} \equiv \pm 1 \bmod (2n+1) \\ \Longleftrightarrow\ & Sord_2(2n+1) = n \end{aligned}$$

□

Si l'on avait une preuve de la conjecture 3.11 nous aurions prouvé qu'il existe une infinité de lignes calculant un cycle de longueur maximal. Car on sait que la longueur de la période est un diviseur de $2^{Sord_2(2n+1)}-1$ donc si $2^{Sord_2(2n+1)}-1$ est un nombre premier et si $Sord_2(2n+1) = n$ alors la période serait de longueur maximale (*ie* $2^n - 1$).

D'autre part il est possible de démontrer que si $2n+1$ (ligne de longueur n) n'est pas un nombre premier alors le cycle ne peut être de longueur maximale (*ie* 2^n-1).

Proposition 3.13 $\forall n \in \mathsf{N},\ 2n+1 \notin \mathcal{P} \Longrightarrow Sord_2(2n+1) < n$

Preuve:

- Si $2n+1 = p^\alpha$ (avec $\alpha > 1$) alors $\varphi(p^\alpha) = p^\alpha - p^{\alpha-1}$ et

$$\begin{aligned} 2^{p^\alpha - p^{\alpha-1}} - 1 &\equiv 0 \bmod (p^\alpha) \\ \left(2^{\frac{p^\alpha - p^{\alpha-1}}{2}} - 1\right)\left(2^{\frac{p^\alpha - p^{\alpha-1}}{2}} + 1\right) &\equiv 0 \bmod (p^\alpha) \\ 2^{\frac{p^\alpha - p^{\alpha-1}}{2}} &\equiv \pm 1 \bmod (p^\alpha) \end{aligned}$$

Donc $Sord_2(2n+1) | \dfrac{p^\alpha - p^{\alpha-1}}{2}$ et d'autre part on a :

$$\begin{aligned} \frac{p^\alpha - p^{\alpha-1}}{2} &< n = \frac{p^\alpha - 1}{2} \\ p^\alpha - p^{\alpha-1} &< p^\alpha - 1 \\ p^{\alpha-1} &> 1 \text{ car } (\alpha > 1) \end{aligned}$$

alors $Sord_2(2n+1) < n$.

- $2n+1 = \prod_{i=1}^{k} p_i^{\alpha_i}$ (avec $k > 1$ et $\alpha_i > 1$) alors $\varphi(\prod_{i=1}^{k} p_i^{\alpha_i}) = \prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i - 1})$ et

$$\begin{aligned} 2^{\left[\prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i-1})\right]} - 1 &\equiv 0 \bmod (\prod_{i=1}^{k} p_i^{\alpha_i}) \\ \left(2^{\frac{\prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i-1})}{2}} - 1\right)\left(2^{\frac{\prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i-1})}{2}} + 1\right) &\equiv 0 \bmod (\prod_{i=1}^{k} p_i^{\alpha_i}) \\ 2^{\frac{\prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i-1})}{2}} &\equiv \pm 1 \bmod (\prod_{i=1}^{k} p_i^{\alpha_i}) \end{aligned}$$

Donc $Sord_2(2n+1) | \frac{\prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i - 1})}{2}$ et d'autre part on a :

$$\frac{\prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i - 1})}{2} < n = \frac{\prod_{i=1}^{k} p_i^{\alpha_i} - 1}{2}$$

mais on sait que $p_i^{\alpha_i} - p_i^{\alpha_i - 1} \leq p_i^{\alpha_i} - 1$ puisque $\alpha_i \geq 1$. Donc

$$\prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i - 1}) \leq \prod_{i=1}^{k}(p_i^{\alpha_i} - 1) < (p_1^{\alpha_1} - 1)\prod_{i=2}^{k} p_i^{\alpha_i} = \prod_{i=1}^{k} p_i^{\alpha_i} - \prod_{i=2}^{k} p_i^{\alpha_i} < \prod_{i=1}^{k} p_i^{\alpha_i} - 1$$

alors $Sord_2(2n+1) < n$. □

Puisque la longueur du cycle pour une ligne de longueur n est un diviseur de $2^{Sord_2(2n+1)} - 1$ et que $Sord_2(2n+1) < n$ si $2n+1 \notin \mathcal{P}$ alors le cycle ne peut être de longueur maximale.

3.4.2 Equivalence avec des automates treillis

Nous allons voir qu'une autre méthode est applicable pour comprendre le comportement des ACLiFs avec peu d'états. L'entrelacement de deux automates treillis permet de comprendre ce qui se passe lors de l'évolution de lignes particulières (longueurs 2^p et $2^p - 1$).

Triangle de Pascal modulo 2

La figure 3.6a représente le calcul d'un automate cellulaire "treillis" calculant le triangle de Pascal modulo 2 dont la configuration de départ est :

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1;\ \forall i \in \mathsf{Z}^\star,\ \begin{bmatrix} 0 \\ 2i \end{bmatrix} = 0$$

et la fonction de transition :

$$\forall t \in \mathsf{N}^\star,\ \forall i \in \mathsf{Z},\ \begin{bmatrix} t \\ i \end{bmatrix} = \begin{bmatrix} t-1 \\ i-1 \end{bmatrix} \oplus \begin{bmatrix} t-1 \\ i+1 \end{bmatrix} \tag{3.9}$$

On peut remarquer que :

$$\begin{bmatrix} t \\ i \end{bmatrix} = \binom{t}{\frac{i+t}{2}} \pmod 2 \tag{3.10}$$

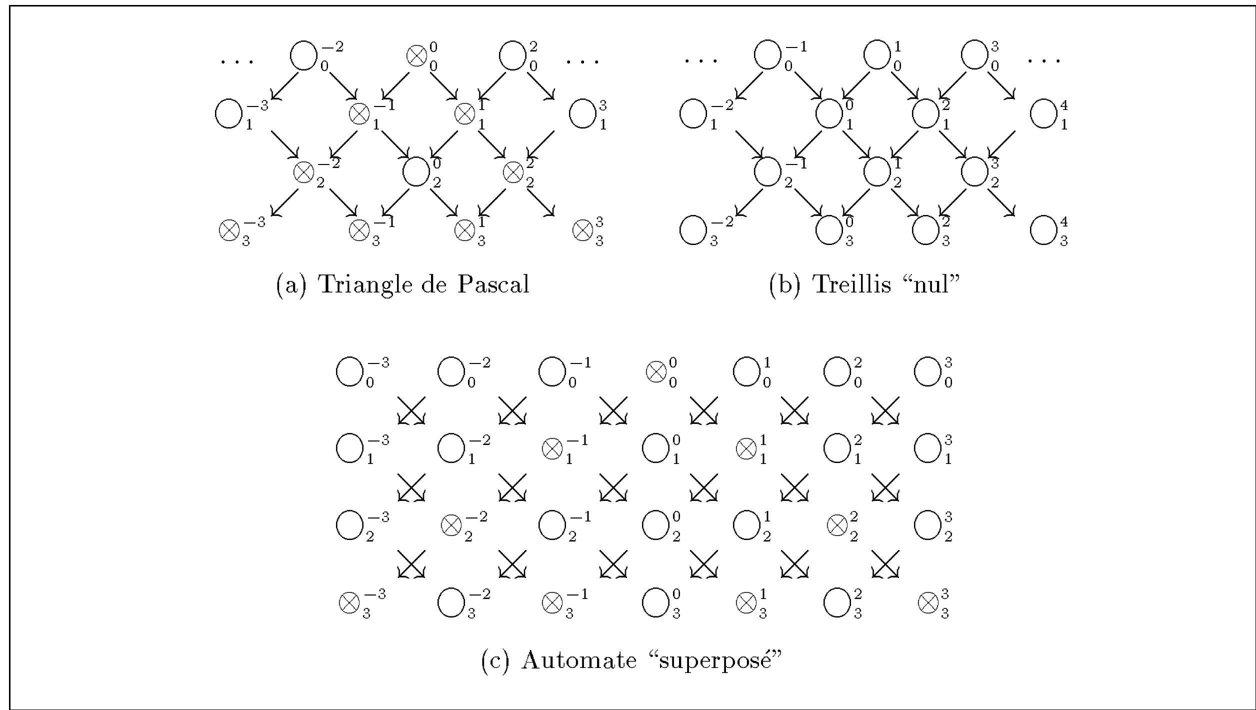
Treillis nul

L'automate cellulaire de la figure 3.6b utilise la même fonction de transition que l'automate précédent (voir équation (3.9)) avec comme configuration de départ la configuration "nulle" :

$$\forall i \in \mathsf{Z},\ \begin{bmatrix} 0 \\ 2i+1 \end{bmatrix} = 0$$

donc

$$\forall i \in \mathsf{Z},\ \forall t \in \mathsf{N},\ \begin{bmatrix} t \\ i \end{bmatrix} = 0 \tag{3.11}$$



(a) Triangle de Pascal (b) Treillis "nul"

(c) Automate "superposé"

FIG. 3.6 - *Automates cellulaires*

Superposition

L'automate de la figure 3.6c est obtenu par entrelacement et superposition des automates représentés sur les figures 3.6a et 3.6b. Ainsi on obtient un automate cellulaire linéaire bi-infini dont la configuration de départ est :

$$\begin{bmatrix}0\\0\end{bmatrix} = 1;\ \forall i \in \mathtt{N},\ \begin{bmatrix}0\\i\end{bmatrix} = 0$$

et la fonction de transition :

$$\mathtt{gmd} \longrightarrow \mathtt{g} \oplus \mathtt{d} \tag{3.12}$$

Une équivalence possible

Dans la suite on notera $\mathcal{A}_s$ l'automate "superposé" et $\mathcal{A}_e$ l'automate calculant la loi `Xor1`.

On aimerait ramener l'étude de $\mathcal{A}_e$ à l'étude du triangle de Pascal modulo 2 puisque les fonctions de transition se ressemblent. On va voir que cela est possible, et ce de façon simple, pour les temps inférieurs à n (si n est la longueur de $\mathcal{A}_e$). Pour cela il faut remarquer que :

1. $\begin{bmatrix}1\\1\end{bmatrix}_s = 1,\ \forall i \in]1,n],\ \begin{bmatrix}1\\i\end{bmatrix}_s = 0$ $\qquad$ $\begin{bmatrix}1\\1\end{bmatrix}_e = 1,\ \forall i \in]1,n],\ \begin{bmatrix}1\\i\end{bmatrix}_e = 0$

   La configuration de départ de $\mathcal{A}_e$ est donc extraite de l'espace-temps de l'automate "superposé" de la façon suivante :

$$\forall i \in [1,n],\ \begin{bmatrix}1\\i\end{bmatrix}_e = \begin{bmatrix}1\\i\end{bmatrix}_s$$

2. En observant le tableau 3.5 et la fonction de transition donnée par l'équation 3.12, on a :
$$\forall i \in]1,n[\,,\ \forall t \in]1,n]\,,\ \begin{bmatrix} t \\ i \end{bmatrix}_e = \begin{bmatrix} t \\ i \end{bmatrix}_s$$

3. **Proposition 3.14** $\forall p \in \mathsf{N},\ \binom{2p}{p} \equiv 0 \pmod 2$

   Preuve:
$$\binom{0}{0} = 0, \text{ et } \forall p \in \mathsf{N}^\star, \left\{ \begin{array}{lcl} \binom{2p}{p} & = & \binom{2p-1}{p} + \binom{2p-1}{p-1} \\ & = & \binom{2p-1}{p} + \binom{2p-1}{(2p-1)-(p-1)} \\ & = & \binom{2p-1}{p} + \binom{2p-1}{p} \\ & = & 2\binom{2p-1}{p} \end{array} \right.$$
   □

4. $\forall t \in]1,n]\,,\ \begin{bmatrix} t \\ 1 \end{bmatrix}_s = \begin{bmatrix} t-1 \\ 0 \end{bmatrix}_s \oplus \begin{bmatrix} t-1 \\ 2 \end{bmatrix}_s$ or $\forall t \in]1,n]\,,\ \begin{bmatrix} t \\ 0 \end{bmatrix}_s = 0$ puisque :
   - $\begin{bmatrix} 2p \\ 0 \end{bmatrix}_s = 0$ (voir Proposition 3.14)
   - $\begin{bmatrix} 2p+1 \\ 0 \end{bmatrix}_s = 0$ (voir paragraphe 3.4.2)

   Donc
$$\forall t \in]1,n]\,,\ \begin{bmatrix} t \\ 1 \end{bmatrix}_e = \begin{bmatrix} t \\ 1 \end{bmatrix}_s$$

5. $\forall t \in]1,n]\,,\ \begin{bmatrix} t \\ n \end{bmatrix}_s = \begin{bmatrix} t-1 \\ n-1 \end{bmatrix}_s \oplus \begin{bmatrix} t-1 \\ n+1 \end{bmatrix}_s$ mais $\forall t \in]1,n]\,,\ \begin{bmatrix} t-1 \\ n+1 \end{bmatrix}_s = \begin{bmatrix} t-1 \\ n \end{bmatrix}_s = 0$

   Donc
$$\forall t \in]1,n]\,,\ \begin{bmatrix} t \\ n \end{bmatrix}_s = \begin{bmatrix} t-1 \\ n-1 \end{bmatrix}_s \oplus \begin{bmatrix} t-1 \\ n \end{bmatrix}_s$$
   Alors
$$\forall t \in]1,n]\,,\ \begin{bmatrix} t \\ n \end{bmatrix}_e = \begin{bmatrix} t \\ n \end{bmatrix}_s$$

L'automate $\mathcal{A}_e$ se comporte donc dans les premiers temps comme un automate calculant un triangle de Pascal modulo 2.

Longueurs de lignes particulières

On peut remarquer que des configurations très simples sont obtenues pour des lignes de longueurs particulières (voir figure 3.7) :

- $\boxed{n = 2^p}$

  On a $\binom{2^p}{0} = \binom{2^p}{2^p} = 1$ et $\forall i \in]1,n[\,,\ \binom{2^p}{i} = 0 \pmod 2$ (lemme 1, p.7 dans [28])

  Donc
$$\begin{bmatrix} n \\ n \end{bmatrix} = 1 \text{ et } \forall i \in [1,n[\,,\ \begin{bmatrix} n \\ i \end{bmatrix} = 0 \qquad (3.13)$$
  Voir les équations (3.10) et (3.11).

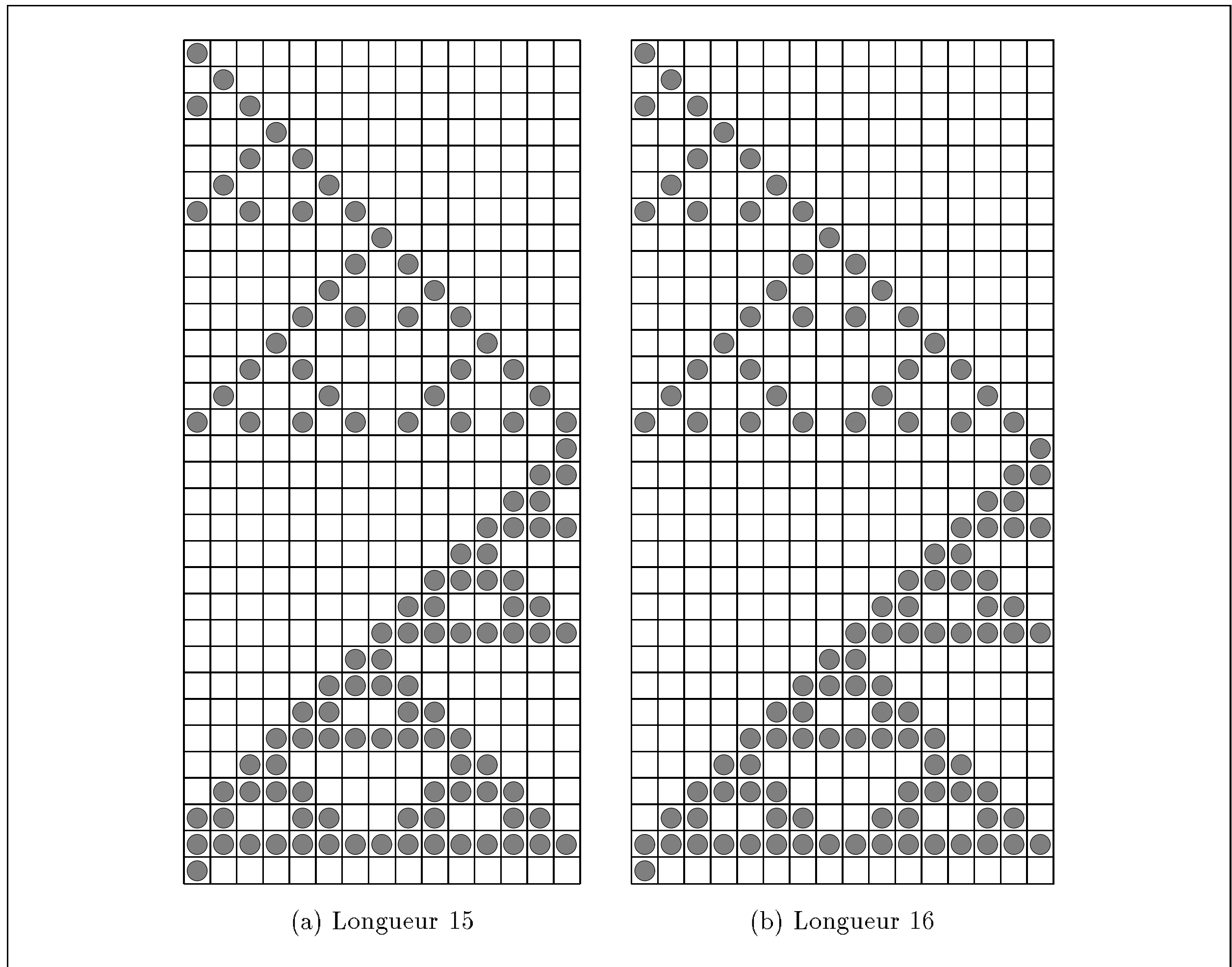


FIG. 3.7 - *Exemples*

– $\boxed{n = 2^p - 1}$

En effet dans ce cas on a $\forall i \in [1, n]$, $\binom{2^p-1}{i} = 1 \pmod 2$

Donc

$$\forall i \in [1, n[, \begin{cases} \text{si } i = 2k, & \begin{bmatrix} n \\ i \end{bmatrix} = 0 \\ \text{si } i = 2k+1, & \begin{bmatrix} n \\ i \end{bmatrix} = 1 \end{cases}$$

ainsi au temps $n + 1 = 2^p$ on se trouve dans la configuration suivante :

$$\begin{cases} \begin{bmatrix} n+1 \\ n \end{bmatrix} = \begin{bmatrix} n-1 \\ n \end{bmatrix} + \begin{bmatrix} n \\ n \end{bmatrix} = 1 + 0 = 1 \\ \forall i \in [1, n[, \begin{bmatrix} n+1 \\ i \end{bmatrix} = \begin{bmatrix} n \\ i-1 \end{bmatrix} + \begin{bmatrix} n \\ i+1 \end{bmatrix} = 2\begin{bmatrix} n \\ i-1 \end{bmatrix} = 0 \end{cases} \tag{3.14}$$

3.4.3 Pavage de l'espace-temps

Cette section nous montre comment il est possible de voir l'évolution de certains ACLiFs comme une simulation d'automate treillis. Autrement dit, le diagramme espace-temps de

l'ACLiF est pavé par des morceaux représentant des cellules d'un treillis calculant un triangle de Pascal modulo 2. Ce qui est intéressant c'est qu'il est possible de prendre des pavés de taille arbitraire mais tous auto-similaires.

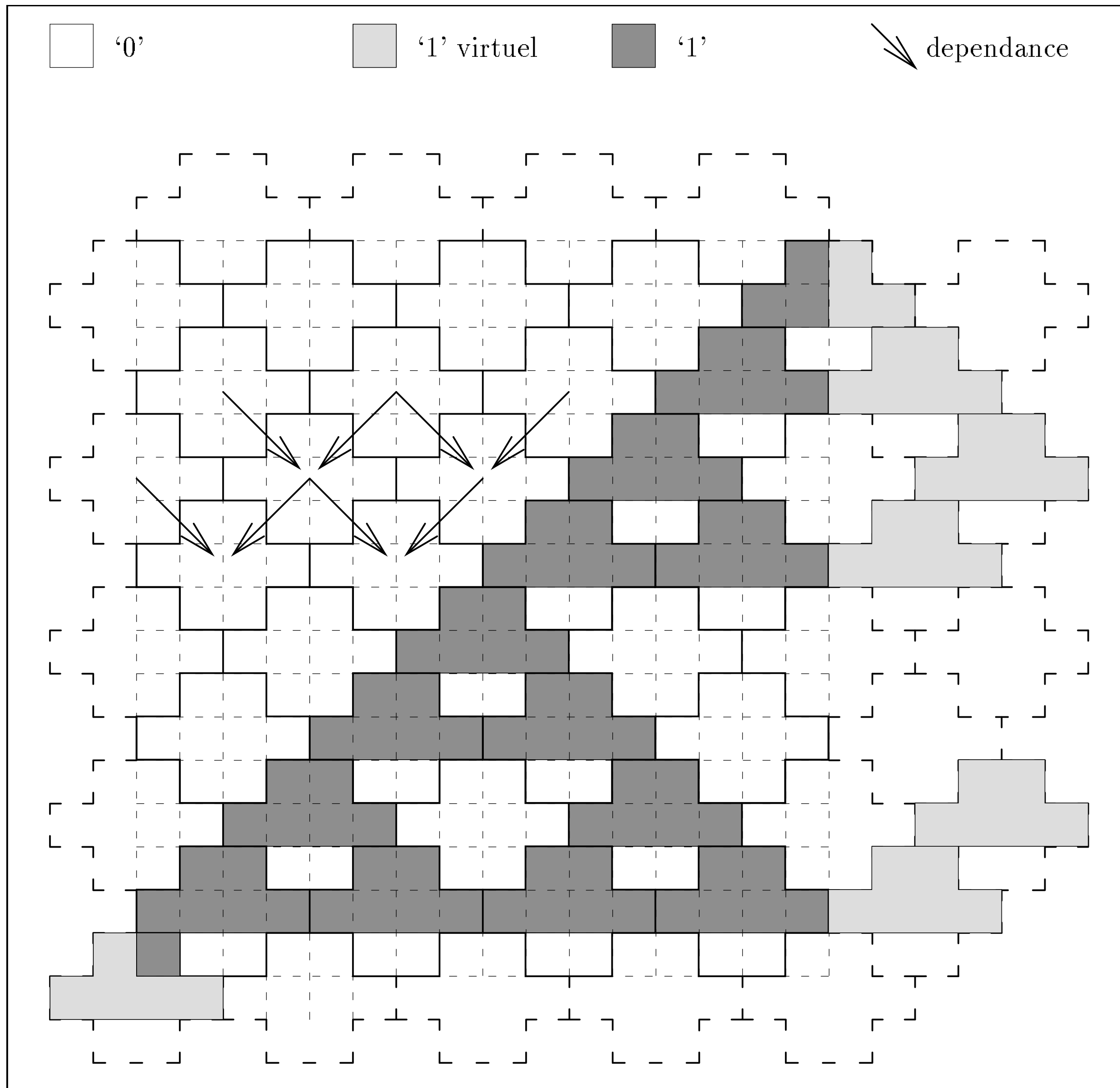


FIG. 3.8 - *Construction d'un treillis*

Définitions

Nous définissons deux objets que nous utiliserons par la suite :

- $\overrightarrow{0} = \left(\begin{array}{cccc} & 0 & 0 & \\ 0 & 0 & 0 & 0 \\ & 0 & 0 & \end{array}\right).$
- $\overrightarrow{1} = \left(\begin{array}{cccc} & 1 & 1 & \\ 1 & 1 & 1 & 1 \\ & 0 & 0 & \end{array}\right).$

Un treillis de pavés

Une interprétation possible du diagramme espace-temps de l'automate cellulaire calculant la loi `Xor1` est illustrée par la figure 3.8. Il s'agit en fait d'un pavage régulier du plan par un unique pavé de la forme

$$\Diamond_i^t = \left(\begin{array}{cccc} & \begin{bmatrix}2t\\2i\end{bmatrix} & \begin{bmatrix}2t\\2i+1\end{bmatrix} & \\ \begin{bmatrix}2t+1\\2i-1\end{bmatrix} & \begin{bmatrix}2t+1\\2i\end{bmatrix} & \begin{bmatrix}2t+1\\2i+1\end{bmatrix} & \begin{bmatrix}2t+1\\2i+2\end{bmatrix} \\ & \begin{bmatrix}2t+2\\2i\end{bmatrix} & \begin{bmatrix}2t+2\\2i+1\end{bmatrix} & \end{array}\right)$$

Si nous considérons un automate cellulaire "treillis" comme celui de la figure 3.6a et des équations (3.9) et (3.10), et que nous remplaçons chaque cellule $\bigcirc_i^t$ par un pavé $\Diamond_i^t$, on obtient alors un treillis de $\Diamond$ dont la fonction de transition est :

$$\Diamond_i^{t+1} = \Diamond_{i-1}^t + \Diamond_{i+1}^t \tag{3.15}$$

On peut observer que si la configuration de départ du treillis ne contient que des $\overrightarrow{0}$ et $\overrightarrow{1}$ alors il ne peut apparaître que des $\overrightarrow{0}$ ou des $\overrightarrow{1}$ dans l'évolution, car :

- $\begin{array}{ccc} \overrightarrow{0} & & \overrightarrow{0} \\ \searrow & & \swarrow \\ & \overrightarrow{0} & \end{array}$ puisque $\begin{array}{ccc} \left(\begin{array}{cccc} & 0 & 0 & \\ 0 & 0 & 0 & 0 \\ & 0 & 0 & \end{array}\right) & & \left(\begin{array}{cccc} & 0 & 0 & \\ 0 & 0 & 0 & 0 \\ & 0 & 0 & \end{array}\right) \\ \searrow & & \swarrow \\ & \left(\begin{array}{cccc} & 0 & 0 & \\ 0 & 0 & 0 & 0 \\ & 0 & 0 & \end{array}\right) & \end{array} \equiv \begin{array}{cccccccc} & 0 & 0 & & & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & \\ & & 0 & 0 & 0 & 0 & & \\ & & & 0 & 0 & & & \end{array}$
- $\begin{array}{ccc} \overrightarrow{1} & & \overrightarrow{0} \\ \searrow & & \swarrow \\ & \overrightarrow{1} & \end{array}$ puisque $\begin{array}{ccc} \left(\begin{array}{cccc} & 1 & 1 & \\ 1 & 1 & 1 & 1 \\ & 0 & 0 & \end{array}\right) & & \left(\begin{array}{cccc} & 0 & 0 & \\ 0 & 0 & 0 & 0 \\ & 0 & 0 & \end{array}\right) \\ \searrow & & \swarrow \\ & \left(\begin{array}{cccc} & 1 & 1 & \\ 1 & 1 & 1 & 1 \\ & 0 & 0 & \end{array}\right) & \end{array} \equiv \begin{array}{cccccccc} & 1 & 1 & & & 0 & 0 & \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 1 & 1 & 0 & 0 & \\ & & 1 & 1 & 1 & 1 & & \\ & & & 0 & 0 & & & \end{array}$
- $\begin{array}{ccc} \overrightarrow{0} & & \overrightarrow{1} \\ \searrow & & \swarrow \\ & \overrightarrow{1} & \end{array}$ puisque $\begin{array}{ccc} \left(\begin{array}{cccc} & 0 & 0 & \\ 0 & 0 & 0 & 0 \\ & 0 & 0 & \end{array}\right) & & \left(\begin{array}{cccc} & 1 & 1 & \\ 1 & 1 & 1 & 1 \\ & 0 & 0 & \end{array}\right) \\ \searrow & & \swarrow \\ & \left(\begin{array}{cccc} & 1 & 1 & \\ 1 & 1 & 1 & 1 \\ & 0 & 0 & \end{array}\right) & \end{array} \equiv \begin{array}{cccccccc} & 0 & 0 & & & 1 & 1 & \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ & 0 & 0 & 1 & 1 & 0 & 0 & \\ & & 1 & 1 & 1 & 1 & & \\ & & & 0 & 0 & & & \end{array}$
- $\begin{array}{ccc} \overrightarrow{1} & & \overrightarrow{1} \\ \searrow & & \swarrow \\ & \overrightarrow{0} & \end{array}$ puisque $\begin{array}{ccc} \left(\begin{array}{cccc} & 1 & 1 & \\ 1 & 1 & 1 & 1 \\ & 0 & 0 & \end{array}\right) & & \left(\begin{array}{cccc} & 1 & 1 & \\ 1 & 1 & 1 & 1 \\ & 0 & 0 & \end{array}\right) \\ \searrow & & \swarrow \\ & \left(\begin{array}{cccc} & 0 & 0 & \\ 0 & 0 & 0 & 0 \\ & 0 & 0 & \end{array}\right) & \end{array} \equiv \begin{array}{cccccccc} & 1 & 1 & & & 1 & 1 & \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & 0 & 0 & 0 & 0 & 0 & 0 & \\ & & 0 & 0 & 0 & 0 & & \\ & & & 0 & 0 & & & \end{array}$

Cet automate calcule un triangle de Pascal modulo 2 dans un “treillis” de $\diamond$. Si on “développe” les $\diamond$ on obtient un diagramme espace-temps d'automate cellulaire dont la fonction de transition vérifie bien l'équation (3.12).

Longueurs de lignes particulières

Si l'on initialise un treillis avec la configuration suivante :

$$\diamond_0^0 = \overrightarrow{1},\ \forall i \in \mathsf{N}^\star,\ \diamond_{2i}^0 = \overrightarrow{0} \tag{3.16}$$

Au temps 2^{p-1} on obtient la configuration suivante (voir équation (3.13)) :

$$\diamond_{-2^{p-1}}^{2^{p-1}} = \diamond_{2^{p-1}}^{2^{p-1}} = \overrightarrow{1} \bigwedge \forall i \in]-2^{p-1}, 2^{p-1}[,\ \diamond_i^{2^{p-1}} = \overrightarrow{0} \tag{3.17}$$

Des évolutions remarquables sont obtenues pour des longueurs de ligne très particulières :

- $\boxed{n = 2^p}$

  Si l'on pave le plan défini par une ligne de longueur 2^p exécutant la loi `Xor1` de façon à ce que :

$$\diamond_0^0 = \left(\begin{bmatrix} 2^p+1 \\ 2^p-1 \end{bmatrix} \begin{array}{c} \begin{bmatrix} 2^p \\ 2^p \end{bmatrix} \\ \begin{bmatrix} 2^p+1 \\ 2^p \end{bmatrix} \\ \begin{bmatrix} 2^p+2 \\ 2^p \end{bmatrix} \end{array} \begin{array}{c} \begin{bmatrix} 2^p \\ 2^p+1 \end{bmatrix} \\ \begin{bmatrix} 2^p+1 \\ 2^p+1 \end{bmatrix} \\ \begin{bmatrix} 2^p+2 \\ 2^p+1 \end{bmatrix} \end{array} \begin{bmatrix} 2^p+1 \\ 2^p+2 \end{bmatrix} \right) = \begin{pmatrix} 1\ 1 \\ 1\ 1\ 1\ 1 \\ 0\ 0 \end{pmatrix} = \overrightarrow{1}$$

  et

$$\begin{aligned} \forall i \in]0, 2^{p-2}],\ \diamond_{-2i}^0 &= \left(\begin{bmatrix} 2^p+1 \\ 2^p-4i-1 \end{bmatrix} \begin{array}{c} \begin{bmatrix} 2^p \\ 2^p-4i \end{bmatrix} \\ \begin{bmatrix} 2^p+1 \\ 2^p-4i \end{bmatrix} \\ \begin{bmatrix} 2^p+2 \\ 2^p-4i \end{bmatrix} \end{array} \begin{array}{c} \begin{bmatrix} 2^p \\ 2^p-4i+1 \end{bmatrix} \\ \begin{bmatrix} 2^p+1 \\ 2^p-4i+1 \end{bmatrix} \\ \begin{bmatrix} 2^p+2 \\ 2^p-4i+1 \end{bmatrix} \end{array} \begin{bmatrix} 2^p+1 \\ 2^p-4i+2 \end{bmatrix} \right) \\ &= \begin{pmatrix} 0\ 0 \\ 0\ 0\ 0\ 0 \\ 0\ 0 \end{pmatrix} \\ &= \overrightarrow{0} \end{aligned}$$

  ce qui est équivalent à partir de la configuration $\overbrace{0 \cdots 0}^{2^p-1}1$ au temps 2^p.

  Alors on obtient :

$$\diamond_{-2^{p-1}}^{2^{p-1}} = \left(\begin{bmatrix} 2^{p+1}+1 \\ -1 \end{bmatrix} \begin{array}{c} \begin{bmatrix} 2^{p+1} \\ 0 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}+1 \\ 0 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}+2 \\ 0 \end{bmatrix} \end{array} \begin{array}{c} \begin{bmatrix} 2^{p+1} \\ 1 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}+1 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}+2 \\ 1 \end{bmatrix} \end{array} \begin{bmatrix} 2^{p+1}+1 \\ 2 \end{bmatrix} \right) = \begin{pmatrix} 1\ 1 \\ 1\ 1\ 1\ 1 \\ 0\ 0 \end{pmatrix} = \overrightarrow{1}$$

et

$$\forall\, i \in [0, 2^{p-2}[\,,\ \Diamond_{-2i}^{2^{p-1}} = \left(\begin{bmatrix} 2^{p+1}+1 \\ 2^p-4i-1 \end{bmatrix} \begin{matrix} \begin{bmatrix} 2^{p+1} \\ 2^p-4i \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}+1 \\ 2^p-4i \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}+2 \\ 2^p-4i \end{bmatrix} \end{matrix} \begin{matrix} \begin{bmatrix} 2^{p+1} \\ 2^p-4i+1 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}+1 \\ 2^p-4i+1 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}+2 \\ 2^p-4i+1 \end{bmatrix} \end{matrix} \begin{bmatrix} 2^{p+1}+1 \\ 2^p-4i+2 \end{bmatrix} \right) = \begin{pmatrix} 0\ 0 \\ 0\ 0\ 0\ 0 \\ 0\ 0 \end{pmatrix} = \overrightarrow{0}$$

c'est à dire $1\overbrace{0\cdots0}^{2^p-1}$ au temps 2^{p+1} (développement de l'équation (3.17)).

La configuration obtenue au temps $2n$ est donc :

$$\begin{bmatrix} 2n \\ 1 \end{bmatrix} = 1 \bigwedge \forall\, i \in]1, n]\,,\ \begin{bmatrix} 2n \\ i \end{bmatrix} = 0 \tag{3.18}$$

- $\boxed{n = 2^p - 1}$

  Si l'on pave le plan défini par une ligne de longueur $2^p - 1$ exécutant la loi Xor1 de façon à ce que :

$$\Diamond_0^0 = \left(\begin{bmatrix} 2^p+1 \\ 2^p-2 \end{bmatrix} \begin{matrix} \begin{bmatrix} 2^p \\ 2^p-1 \end{bmatrix} \\ \begin{bmatrix} 2^p+1 \\ 2^p-1 \end{bmatrix} \\ \begin{bmatrix} 2^p+2 \\ 2^p-1 \end{bmatrix} \end{matrix} \begin{matrix} \begin{bmatrix} 2^p \\ 2^p \end{bmatrix} \\ \begin{bmatrix} 2^p+1 \\ 2^p \end{bmatrix} \\ \begin{bmatrix} 2^p+2 \\ 2^p \end{bmatrix} \end{matrix} \begin{bmatrix} 2^p+1 \\ 2^p+1 \end{bmatrix} \right) = \begin{pmatrix} 1\ 1 \\ 1\ 1\ 1\ 1 \\ 0\ 0 \end{pmatrix} = \overrightarrow{1}$$

et

$$\forall\, i \in]0, 2^{p-2}]\,,\ \Diamond_{-2i}^{0} = \left(\begin{bmatrix} 2^p+1 \\ 2^p-4i-2 \end{bmatrix} \begin{matrix} \begin{bmatrix} 2^p \\ 2^p-4i-1 \end{bmatrix} \\ \begin{bmatrix} 2^p+1 \\ 2^p-4i-1 \end{bmatrix} \\ \begin{bmatrix} 2^p+2 \\ 2^p-4i-1 \end{bmatrix} \end{matrix} \begin{matrix} \begin{bmatrix} 2^p \\ 2^p-4i \end{bmatrix} \\ \begin{bmatrix} 2^p+1 \\ 2^p-4i \end{bmatrix} \\ \begin{bmatrix} 2^p+2 \\ 2^p-4i \end{bmatrix} \end{matrix} \begin{bmatrix} 2^p+1 \\ 2^p-4i+1 \end{bmatrix} \right) = \begin{pmatrix} 0\ 0 \\ 0\ 0\ 0\ 0 \\ 0\ 0 \end{pmatrix} = \overrightarrow{0}$$

ce qui est équivalent à partir de la configuration $\overbrace{0\cdots0}^{2^p-2}1$ au temps 2^p.

On obtient alors :

$$\forall\, i \in [0, 2^{p-2}-1]\,,\ \Diamond_{-2i-1}^{2^{p-1}-1} = \left(\begin{bmatrix} 2^{p+1}-1 \\ 2^p-4i-4 \end{bmatrix} \begin{matrix} \begin{bmatrix} 2^{p+1}-2 \\ 2^p-4i-3 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}-1 \\ 2^p-4i-3 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1} \\ 2^p-4i-3 \end{bmatrix} \end{matrix} \begin{matrix} \begin{bmatrix} 2^{p+1}-2 \\ 2^p-4i-2 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1}-1 \\ 2^p-4i-2 \end{bmatrix} \\ \begin{bmatrix} 2^{p+1} \\ 2^p-4i-2 \end{bmatrix} \end{matrix} \begin{bmatrix} 2^{p+1}-1 \\ 2^p-4i-1 \end{bmatrix} \right)$$

$$= \begin{pmatrix} 1\,1 \\ 1\,1\,1\,1 \\ 0\,0 \end{pmatrix}$$
$$= \overrightarrow{1}$$

qui entraîne la configuration $\overbrace{1\cdots1}^{2^p-1}$ au temps $2^{p+1}-1$.

Donc la configuration $1\,\overbrace{0\cdots0}^{2^p-2}$ au temps 2^p :

$$\left[{2(n+1) \atop 1}\right] = 1 \bigwedge \forall i \in]1,n] \, , \, \left[{2(n+1) \atop i}\right] = 0 \qquad (3.19)$$

3.4.4 Conclusion

Nous avons obtenu un certain nombre de résultats concernant la longueur des cycles. En effet, nous venons de voir (équations (3.13), (3.14), (3.18) et (3.19)) que pour les lignes de longueur $n = 2^p$ (resp. $n = 2^p - 1$) le cycle calculé par la fonction de transition `Xor1` est de longueur $2n = 2^{p+1}$ (resp. $2(n+1) = 2^{p+1}$).

D'autre part, il semble que la borne maximale (cycle de longueur 2^n pour une ligne de longueur n) soit atteinte une infinité de fois. Toutefois ce résultat paraît être lié à un problème de théorie des nombres ouvert. Ce qui signifie que toutes les configurations possibles de l'automate sont atteintes (il s'agit alors d'une "incrémentation" parallèle du nombre représenté).

3.5 La loi `Xor4`

Nous allons voir qu'il est possible de ramener l'étude de l'automate calculant la loi `Xor4` à celle de l'étude déjà faite de l'automate exécutant la loi `Xor1`. En effet nous pouvons remarquer que l'automate calculant la loi `Xor4` possède un diagramme espace-temps comparable à celui de la loi `Xor1`. La fonction de transition de l'automate `Xor4` ayant pour effet de bord à gauche (resp. à droite) celui que l'automate `Xor1` possède à droite (resp. à gauche) (voir tableau 3.5).

3.5.1 Partitionnement de l'espace-temps

Examinons l'équation exprimant le sous-ordre de n :

–
$$\begin{aligned}
2^{Sord_2(2n+1)} &\equiv 1 \bmod (2n+1) \\
2^{Sord_2(2n+1)} &\equiv 2n+2 \bmod (2n+1) \\
2^{Sord_2(2n+1)-1} &\equiv n+1 \bmod (2n+1) \\
2^{Sord_2(2n+1)-1} &\equiv -n \bmod (2n+1)
\end{aligned}$$

– $$\begin{array}{rcl} 2^{Sord_2(2n+1)} & \equiv & -1 \bmod (2n+1) \\ 2^{Sord_2(2n+1)} & \equiv & 2n \bmod (2n+1) \\ 2^{Sord_2(2n+1)-1} & \equiv & n \bmod (2n+1) \end{array}$$

Donc on a :

$$|(2^{Sord_2(2n+1)-1} + n) \bmod (2n+1) - n| = n \tag{3.20}$$

Ce qui signifie que l'espace-temps de la loi `Xor1` peut-être coupé en deux parties de longueurs égales.

On définit alors :

- $\Pi_1^n(X)$: la partie de l'espace-temps de l'automate de longueur n exécutant la loi X dont les temps sont inférieurs à $2^{Sord_2(2n+1)-1}$:

$$\Pi_1^n(X) = \{\left[\begin{smallmatrix} t \\ i \end{smallmatrix}\right] \text{ avec } i \in [1,n] \ \wedge \ t \in [1, 2^{Sord_2(2n+1)-1}[\}$$

- $\Pi_2^n(X)$: la partie de l'espace-temps de l'automate de longueur n exécutant la loi X dont les temps sont supérieurs à $2^{Sord_2(2n+1)-1}$:

$$\Pi_2^n(X) = \{\left[\begin{smallmatrix} t \\ i \end{smallmatrix}\right] \text{ avec } i \in [1,n] \ \wedge \ t \in [2^{Sord_2(2n+1)-1}, 2^{Sord_2(2n+1)}[\}$$

Pour l'automate `Xor1`, si l'on part de la configuration $1\overbrace{0\cdots0}^{n-1}$ au temps 1, au temps $2^{Sord_2(2n+1)-1}$ on obtient la configuration $\overbrace{0\cdots0}^{n-1}1$ et donc la configuration $1\overbrace{0\cdots0}^{n-1}$ au temps $2^{Sord_2(2n+1)}$.

3.5.2 Equivalence avec la loi `Xor1`

Définition 3.15 *On appelle* image miroir *de la configuration $\mathcal{C}_t^n$ la configuration $\overline{\mathcal{C}_t^n}$ dans laquelle :*

$$\forall\, i \in [1,n]\,,\ \overline{\mathcal{C}_t^n}(i) = q \ \textit{ssi}\ \mathcal{C}_t^n(n-i) = q$$

Définition 3.16 *On étend naturellement la notion de configuration miroir à celle d'*espace-temps miroir *en inversant toutes les configurations apparaissant dans un diagramme esapce-temps donné.*

L'inversion des effets de bords dans les lois `Xor1` et `Xor4` nous donnent :

$$\begin{array}{rcl} \forall\, t \in [1, 2^{Sord_2(2n+1)-1}[\,,\ \forall\, i \in [1,n] & , & \left[\begin{smallmatrix} t \\ i \end{smallmatrix}\right]_{Xor4} = \left[\begin{smallmatrix} t+2^{Sord_2(2n+1)-1}-1 \\ n-i+1 \end{smallmatrix}\right]_{Xor1} \\ & \Longleftrightarrow & \\ \forall\, t \in [1, 2^{Sord_2(2n+1)-1}[& , & \mathcal{C}_t^n[Xor4] = \overline{\mathcal{C}^n_{t+2^{Sord_2(2n+1)-1}}[Xor1]} \end{array}$$

et

$$\begin{array}{rcl} \forall\, t \in [2^{Sord_2(2n+1)-1}, 2^{Sord_2(2n+1)}[,\ \forall\, i \in [1,n] & , & \begin{bmatrix} t \\ i \end{bmatrix}_{Xor4} = \begin{bmatrix} t-2^{Sord_2(2n+1)-1}+1 \\ n-i+1 \end{bmatrix}_{Xor1} \\ & \Longleftrightarrow & \\ \forall\, t \in [2^{Sord_2(2n+1)-1}, 2^{Sord_2(2n+1)}[& , & \mathcal{C}_t^n[Xor4] = \overline{\mathcal{C}^n_{t-2^{Sord_2(2n+1)+1})}[Xor1]} \end{array}$$

Finalement on obtient :

$$\forall n \in \mathsf{N}^\star, \begin{cases} \Pi_1^n(Xor4) = \overline{\Pi_2^n(Xor1)} \\ \Pi_2^n(Xor4) = \overline{\Pi_1^n(Xor1)} \end{cases}$$

Si l'on part de la configuration $1\overbrace{0\cdots0}^{n-1}$ au temps 1, au temps $2^{Sord_2(2n+1)-1}$ on obtient la configuration $\overbrace{0\cdots0}^{n-1}1$ et donc la configuration $1\overbrace{0\cdots0}^{n-1}$ au temps $2^{Sord_2(2n+1)}$.

3.5.3 Conclusion

Ainsi l'automate exécutant la fonction de transition `Xor4` calcule des cycles de même longueur que l'automate `Xor1`, ces cycles contiennent les mêmes configurations à la fonction miroir et la fonction de partitionnement près.

Annexe A

ACLiF synchronisant en temps $3n-3$ avec 9 états

(-,.,.)↦(.)	(-,.,X)↦(X)	(-,.,Y)↦(.)	(-,.,z)↦(.)	(-,.,d)↦(z)	(.,.,-)↦(.)
(.,.,.)↦(.)	(.,.,X)↦(d)	(.,.,Y)↦(.)	(.,.,z)↦(.)	(.,.,d)↦(d)	(.,.,a)↦(.)
(.,.,b)↦(Y)	(.,.,g)↦(.)	(X,.,-)↦(X)	(X,.,.)↦(d)	(X,.,X)↦(X)	(X,.,g)↦(d)
(Y,.,-)↦(.)	(Y,.,.)↦(.)	(Y,.,Y)↦(.)	(Y,.,d)↦(d)	(Y,.,g)↦(.)	(z,.,-)↦(.)
(z,.,.)↦(.)	(z,.,z)↦(.)	(z,.,a)↦(.)	(z,.,g)↦(.)	(d,.,-)↦(z)	(d,.,.)↦(d)
(d,.,Y)↦(d)	(d,.,d)↦(z)	(d,.,b)↦(X)	(d,.,g)↦(d)	(a,.,.)↦(.)	(a,.,z)↦(.)
(b,.,.)↦(Y)	(b,.,d)↦(X)	(g,.,.)↦(.)	(g,.,X)↦(d)	(g,.,Y)↦(.)	(g,.,z)↦(.)
(g,.,d)↦(d)	(-,X,.)↦(a)	(-,X,X)↦(F)	(.,X,X)↦(X)	(X,X,-)↦(F)	(X,X,.)↦(X)
(X,X,X)↦(F)	(X,X,z)↦(g)	(X,X,d)↦(X)	(X,X,g)↦(X)	(z,X,X)↦(g)	(z,X,z)↦(X)
(z,X,g)↦(X)	(d,X,X)↦(X)	(d,X,d)↦(b)	(a,X,-)↦(F)	(g,X,X)↦(X)	(g,X,z)↦(X)
(.,Y,z)↦(a)	(.,Y,b)↦(X)	(z,Y,.)↦(a)	(z,Y,z)↦(a)	(z,Y,g)↦(a)	(b,Y,.)↦(X)
(b,Y,g)↦(X)	(g,Y,z)↦(a)	(g,Y,b)↦(X)	(-,z,X)↦(X)	(-,z,Y)↦(z)	(-,z,z)↦(z)
(-,z,d)↦(.)	(-,z,a)↦(z)	(-,z,b)↦(.)	(.,z,X)↦(d)	(.,z,Y)↦(z)	(.,z,z)↦(d)
(.,z,d)↦(.)	(.,z,a)↦(z)	(.,z,b)↦(.)	(X,z,-)↦(X)	(X,z,.)↦(d)	(X,z,X)↦(X)
(X,z,z)↦(X)	(X,z,d)↦(Y)	(X,z,g)↦(d)	(Y,z,.)↦(z)	(Y,z,Y)↦(z)	(Y,z,z)↦(d)
(Y,z,d)↦(.)	(Y,z,g)↦(z)	(z,z,-)↦(z)	(z,z,.)↦(d)	(z,z,X)↦(X)	(z,z,Y)↦(d)
(z,z,z)↦(z)	(z,z,d)↦(g)	(z,z,b)↦(X)	(d,z,-)↦(.)	(d,z,.)↦(.)	(d,z,X)↦(Y)
(d,z,Y)↦(.)	(d,z,z)↦(g)	(d,z,d)↦(.)	(d,z,b)↦(Y)	(d,z,g)↦(.)	(a,z,.)↦(z)
(a,z,a)↦(z)	(a,z,g)↦(z)	(b,z,.)↦(.)	(b,z,z)↦(X)	(b,z,d)↦(Y)	(b,z,b)↦(.)
(b,z,g)↦(.)	(g,z,X)↦(d)	(g,z,Y)↦(z)	(g,z,d)↦(.)	(g,z,a)↦(z)	(g,z,b)↦(.)
(.,d,X)↦(z)	(.,d,z)↦(z)	(X,d,.)↦(z)	(X,d,g)↦(z)	(z,d,.)↦(z)	(z,d,a)↦(Y)
(z,d,g)↦(z)	(a,d,.)↦(z)	(a,d,z)↦(Y)	(g,d,X)↦(z)	(g,d,z)↦(z)	(-,a,X)↦(F)
(-,a,d)↦(b)	(.,a,z)↦(b)	(.,a,g)↦(b)	(z,a,.)↦(b)	(z,a,d)↦(b)	(d,a,z)↦(b)
(d,a,g)↦(b)	(g,a,.)↦(b)	(g,a,d)↦(b)	(-,b,z)↦(z)	(.,b,z)↦(z)	(.,b,g)↦(z)
(Y,b,z)↦(X)	(Y,b,g)↦(X)	(z,b,.)↦(z)	(z,b,Y)↦(X)	(z,b,z)↦(z)	(g,b,.)↦(z)
(g,b,Y)↦(X)	(.,g,g)↦(g)	(X,g,g)↦(X)	(Y,g,g)↦(g)	(z,g,g)↦(g)	(d,g,g)↦(z)
(a,g,g)↦(g)	(b,g,g)↦(g)	(g,g,.)↦(g)	(g,g,X)↦(X)	(g,g,Y)↦(g)	(g,g,z)↦(g)
(g,g,d)↦(z)	(g,g,a)↦(g)	(g,g,b)↦(g)			

TAB. A.1 - *Transitions de l'automate synchronisant avec 9 états (165 transitions)*

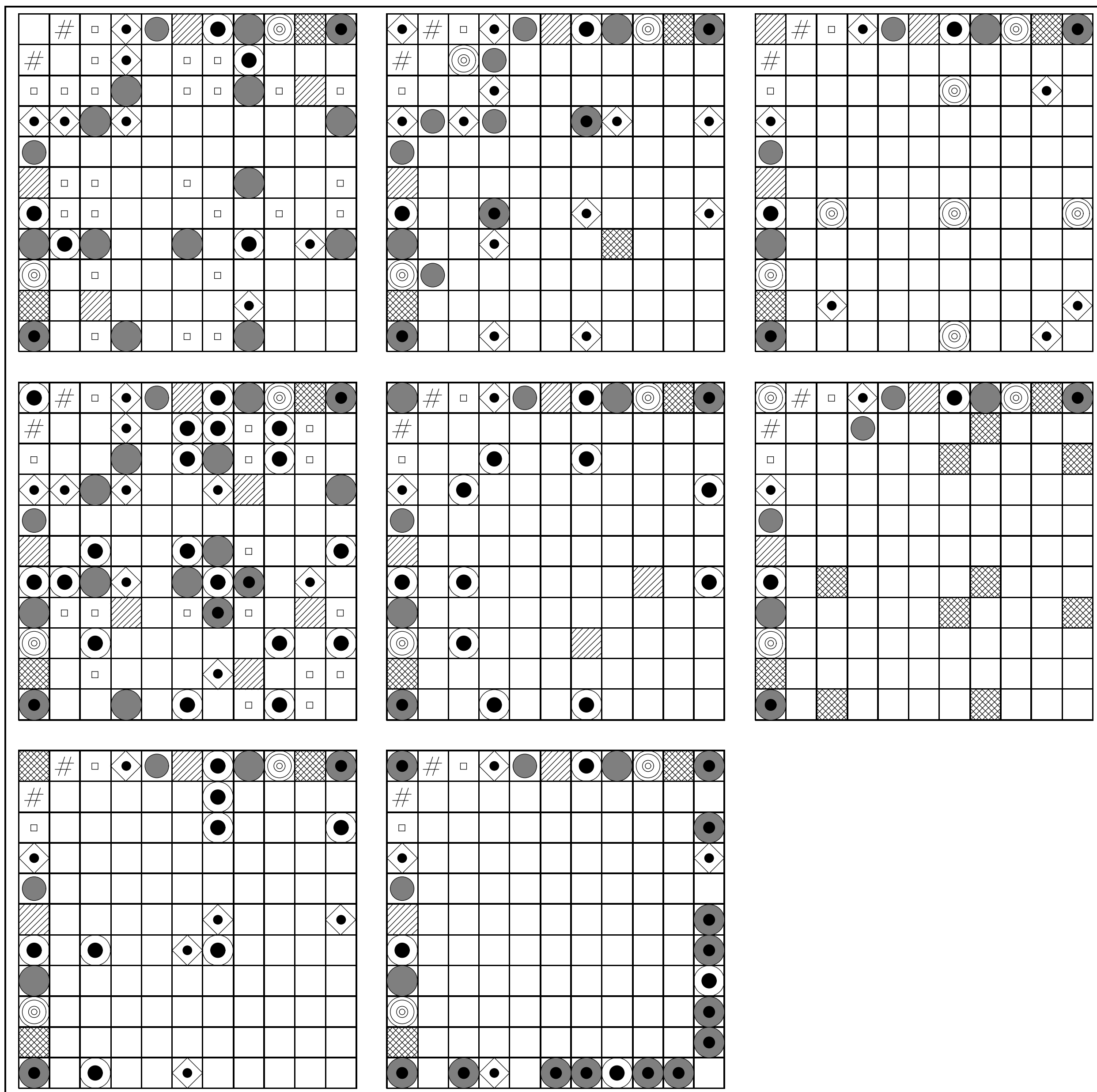


FIG. A.1 - *Automate synchronisant avec 9 états*

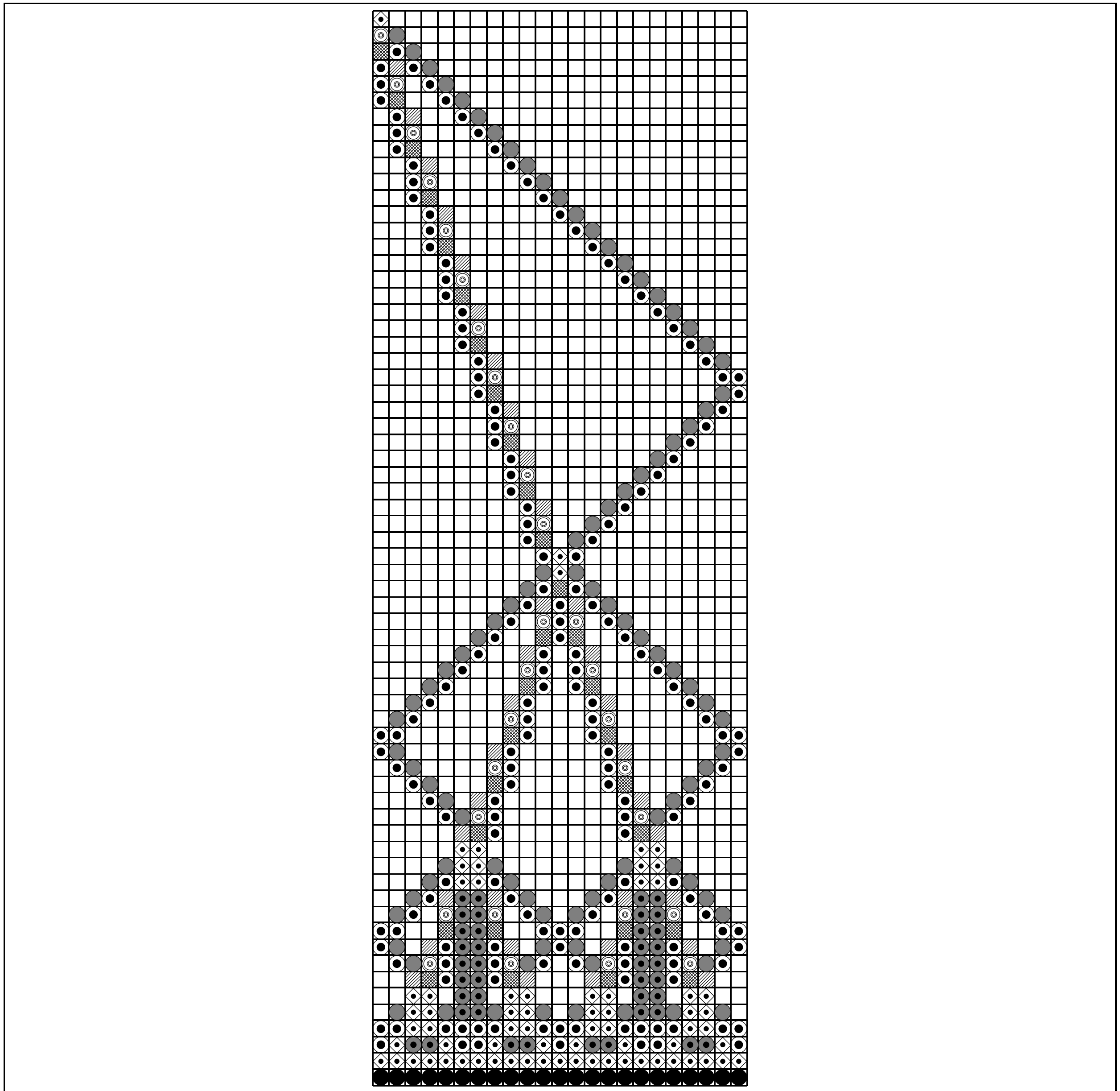


FIG. A.2 - *Synchronisation d'une ligne de longueur 23 avec 9 états*

Annexe B

ACLiF synchronisant en temps $3n + O(\log n)$ avec 13 états

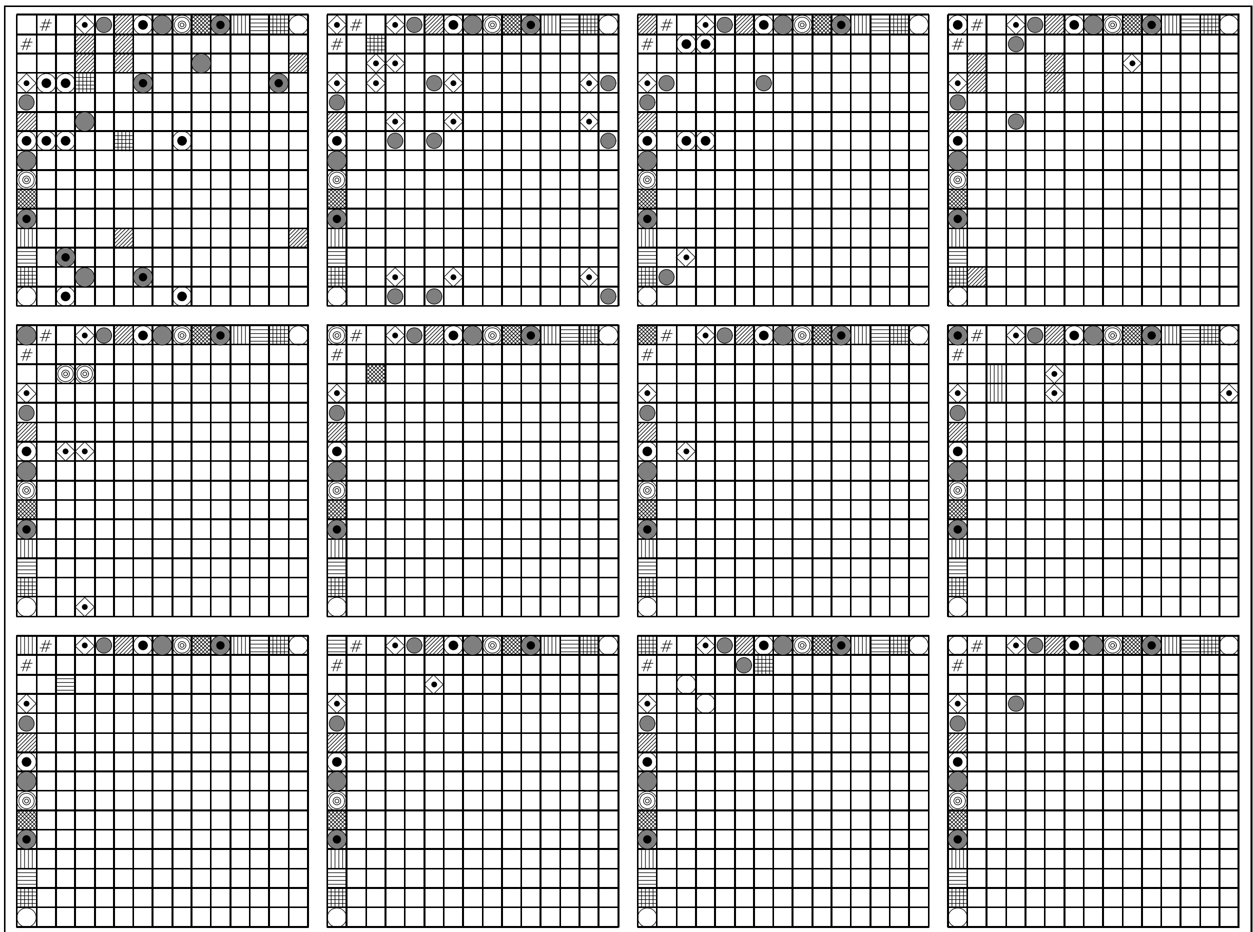


FIG. B.1 - *Automate synchronisant avec 13 états (Yunès)*

(-,.,.)↦(.)	(-,.,X)↦(L)	(-,.,L)↦(L)	(-,.,R)↦(.)	(-,.,A)↦(.)	(-,.,B)↦(.)
(-,.,C)↦(.)	(.,.,-)↦(.)	(.,.,.)↦(.)	(.,.,X)↦(L)	(.,.,L)↦(L)	(.,.,R)↦(.)
(.,.,a)↦(.)	(.,.,b)↦(.)	(.,.,c)↦(a)	(.,.,A)↦(.)	(.,.,B)↦(.)	(.,.,C)↦(.)
(.,.,Y)↦(.)	(.,.,Z)↦(L)	(X,.,-)↦(R)	(X,.,.)↦(R)	(X,.,X)↦(Y)	(X,.,R)↦(A)
(X,.,Y)↦(A)	(L,.,-)↦(.)	(L,.,.)↦(.)	(L,.,X)↦(a)	(L,.,R)↦(.)	(L,.,a)↦(.)
(R,.,-)↦(R)	(R,.,.)↦(R)	(R,.,L)↦(Y)	(R,.,b)↦(R)	(a,.,.)↦(.)	(b,.,.)↦(.)
(b,.,B)↦(.)	(c,.,.)↦(.)	(c,.,C)↦(.)	(A,.,.)↦(.)	(A,.,R)↦(.)	(A,.,Y)↦(.)
(B,.,.)↦(.)	(B,.,L)↦(L)	(B,.,Z)↦(L)	(C,.,.)↦(A)	(Y,.,.)↦(.)	(Y,.,X)↦(a)
(Y,.,R)↦(A)	(Y,.,a)↦(.)	(Z,.,.)↦(R)	(Z,.,b)↦(R)	(-,X,.)↦(Y)	(.,X,.)↦(X)
(.,X,X)↦(X)	(X,X,.)↦(X)	(X,X,L)↦(F)	(X,X,R)↦(X)	(X,X,A)↦(.)	(X,X,Y)↦(X)
(X,X,Z)↦(F)	(L,X,X)↦(X)	(L,X,R)↦(X)	(L,X,Y)↦(X)	(R,X,X)↦(F)	(R,X,L)↦(F)
(R,X,Z)↦(F)	(a,X,X)↦(.)	(a,X,A)↦(.)	(Y,X,X)↦(X)	(Y,X,R)↦(X)	(Y,X,Y)↦(X)
(Z,X,X)↦(F)	(Z,X,L)↦(F)	(Z,X,Z)↦(F)	(-,L,.)↦(R)	(-,L,X)↦(R)	(.,L,-)↦(.)
(.,L,.)↦(.)	(.,L,X)↦(.)	(.,L,R)↦(.)	(X,L,-)↦(F)	(X,L,R)↦(F)	(R,L,.)↦(R)
(R,L,X)↦(R)	(A,L,-)↦(.)	(A,L,.)↦(.)	(A,L,R)↦(.)	(C,L,.)↦(X)	(Y,L,-)↦(F)
(-,R,.)↦(.)	(-,R,X)↦(F)	(-,R,a)↦(.)	(.,R,-)↦(L)	(.,R,.)↦(.)	(.,R,L)↦(L)
(.,R,a)↦(.)	(.,R,c)↦(X)	(X,R,-)↦(L)	(X,R,.)↦(.)	(X,R,L)↦(L)	(L,R,.)↦(.)
(L,R,X)↦(F)	(L,R,a)↦(.)	(Y,R,-)↦(L)	(Y,R,.)↦(.)	(.,a,.)↦(b)	(.,a,X)↦(b)
(R,a,.)↦(X)	(R,a,X)↦(X)	(Z,a,X)↦(X)	(.,b,.)↦(c)	(.,c,.)↦(.)	(R,c,.)↦(X)
(.,A,.)↦(B)	(.,A,L)↦(X)	(X,A,.)↦(B)	(X,A,L)↦(X)	(X,A,Z)↦(X)	(.,B,.)↦(C)
(.,C,.)↦(.)	(.,C,L)↦(X)	(-,Y,.)↦(.)	(-,Y,L)↦(F)	(-,Y,R)↦(Y)	(.,Y,.)↦(Z)
(X,Y,X)↦(Z)	(.,Z,.)↦(.)	(X,Z,X)↦(F)	(A,Z,a)↦(.)		

TAB. B.1 - *Solution de Minsky (130 transitions)*

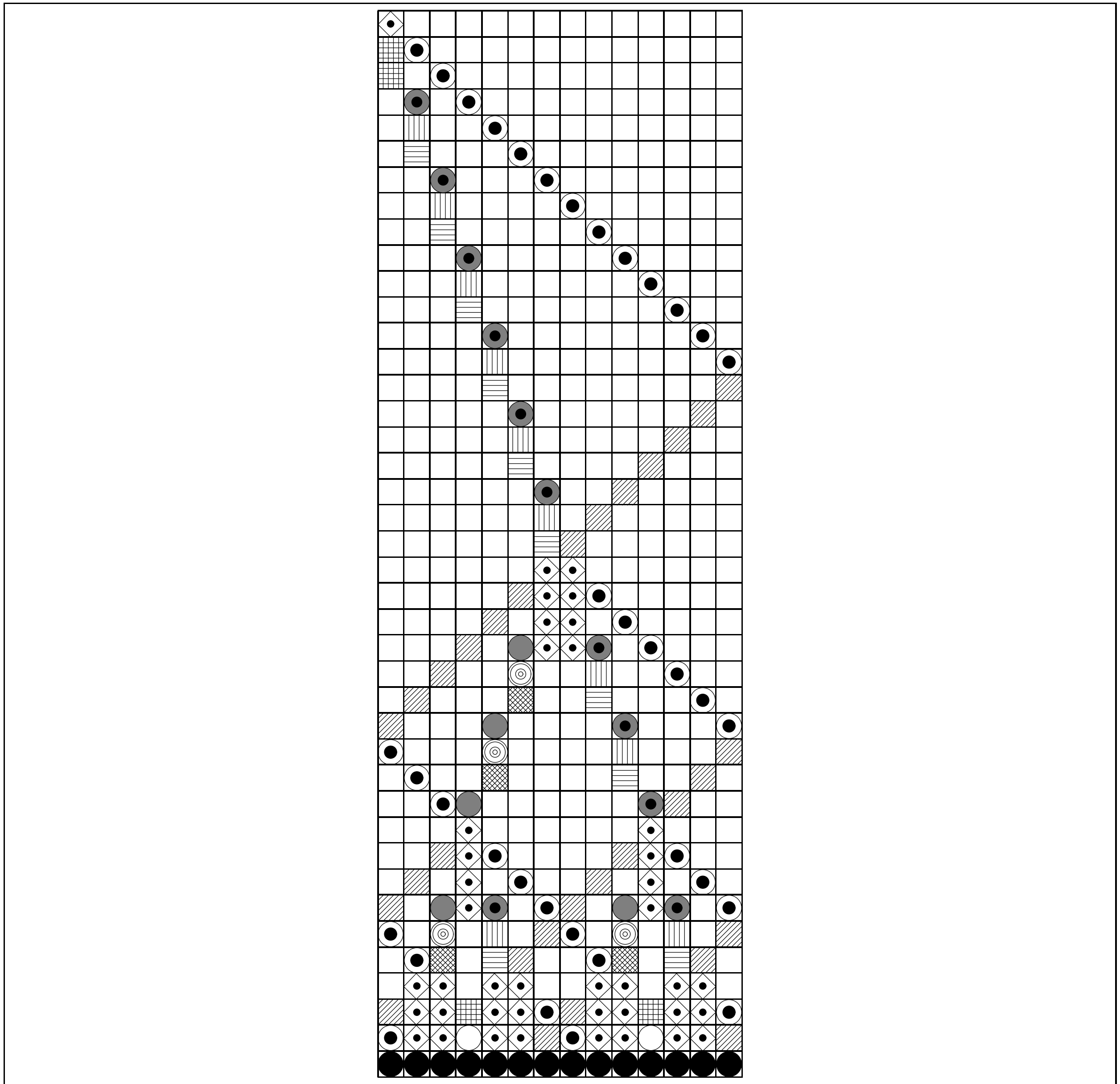


Fig. B.2 - *Synchronisation d'une ligne de longueur 14 avec 13 états*

Annexe C

ACLiF synchronisant en temps $3n + O(\log n)$ avec 7 états

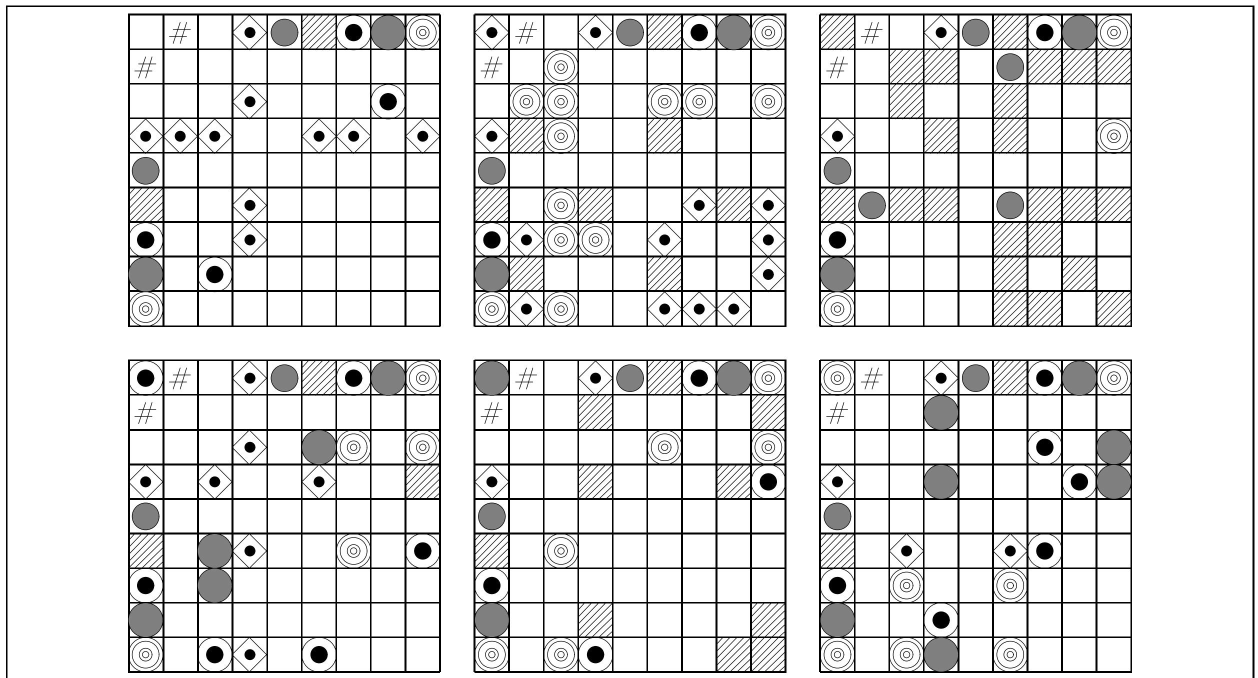


FIG. C.1 - *Automate synchronisant avec 7 états (Yunès)*

(.,.,-)↦(.)	(.,.,.)↦(.)	(.,.,X)↦(X)	(.,.,G)↦(.)	(.,.,a)↦(.)	(.,.,b)↦(a)
(.,.,d)↦(.)	(X,.,-)↦(X)	(X,.,.)↦(X)	(X,.,G)↦(X)	(X,.,a)↦(X)	(X,.,d)↦(X)
(G,.,.)↦(.)	(G,.,X)↦(X)	(G,.,a)↦(.)	(G,.,d)↦(.)	(a,.,.)↦(.)	(a,.,X)↦(X)
(a,.,G)↦(.)	(a,.,d)↦(.)	(b,.,.)↦(a)	(d,.,-)↦(.)	(d,.,.)↦(.)	(d,.,G)↦(.)
(d,.,a)↦(.)	(-,X,.)↦(d)	(.,X,-)↦(d)	(.,X,.)↦(d)	(.,X,G)↦(d)	(.,X,a)↦(d)
(.,X,d)↦(d)	(X,X,-)↦(G)	(X,X,.)↦(d)	(X,X,G)↦(G)	(G,X,.)↦(d)	(G,X,X)↦(G)
(G,X,a)↦(X)	(G,X,b)↦(G)	(G,X,d)↦(X)	(a,X,-)↦(X)	(a,X,.)↦(d)	(a,X,X)↦(d)
(a,X,G)↦(X)	(a,X,d)↦(X)	(b,X,-)↦(G)	(b,X,G)↦(G)	(b,X,d)↦(X)	(d,X,-)↦(X)
(d,X,.)↦(d)	(d,X,G)↦(X)	(d,X,a)↦(X)	(d,X,b)↦(X)	(-,G,.)↦(G)	(-,G,X)↦(G)
(-,G,G)↦(F)	(-,G,a)↦(G)	(-,G,b)↦(G)	(-,G,d)↦(G)	(.,G,.)↦(G)	(.,G,G)↦(G)
(X,G,X)↦(G)	(X,G,G)↦(G)	(X,G,d)↦(d)	(G,G,-)↦(F)	(G,G,.)↦(G)	(G,G,X)↦(G)
(G,G,G)↦(F)	(G,G,a)↦(G)	(G,G,b)↦(G)	(G,G,d)↦(G)	(a,G,G)↦(G)	(a,G,a)↦(G)
(b,G,G)↦(G)	(b,G,b)↦(G)	(d,G,G)↦(G)	(d,G,a)↦(G)	(d,G,d)↦(G)	(.,a,X)↦(X)
(.,a,G)↦(b)	(.,a,a)↦(d)	(.,a,d)↦(d)	(X,a,.)↦(X)	(X,a,G)↦(X)	(X,a,d)↦(G)
(G,a,.)↦(b)	(G,a,X)↦(X)	(G,a,a)↦(d)	(G,a,d)↦(a)	(a,a,.)↦(b)	(d,a,.)↦(a)
(d,a,X)↦(X)	(d,a,G)↦(a)	(-,b,X)↦(G)	(-,b,d)↦(G)	(.,b,G)↦(d)	(.,b,d)↦(d)
(X,b,X)↦(G)	(X,b,G)↦(.)	(X,b,b)↦(G)	(X,b,d)↦(a)	(G,b,.)↦(d)	(G,b,X)↦(.)
(b,b,X)↦(G)	(b,b,d)↦(G)	(d,b,.)↦(d)	(d,b,X)↦(a)	(d,b,b)↦(G)	(d,b,d)↦(G)
(-,d,X)↦(b)	(.,d,X)↦(.)	(.,d,a)↦(a)	(.,d,b)↦(.)	(.,d,d)↦(b)	(X,d,-)↦(.)
(X,d,.)↦(.)	(X,d,X)↦(b)	(X,d,G)↦(.)	(X,d,a)↦(.)	(X,d,b)↦(a)	(X,d,d)↦(b)
(G,d,.)↦(X)	(G,d,X)↦(.)	(G,d,G)↦(X)	(G,d,a)↦(a)	(G,d,b)↦(.)	(a,d,.)↦(d)
(a,d,X)↦(.)	(a,d,G)↦(d)	(b,d,.)↦(.)	(b,d,X)↦(a)	(b,d,G)↦(.)	(d,d,.)↦(d)
(d,d,X)↦(b)	(d,d,G)↦(d)				

TAB. C.1 - *Solution de Minsky (134 transitions)*

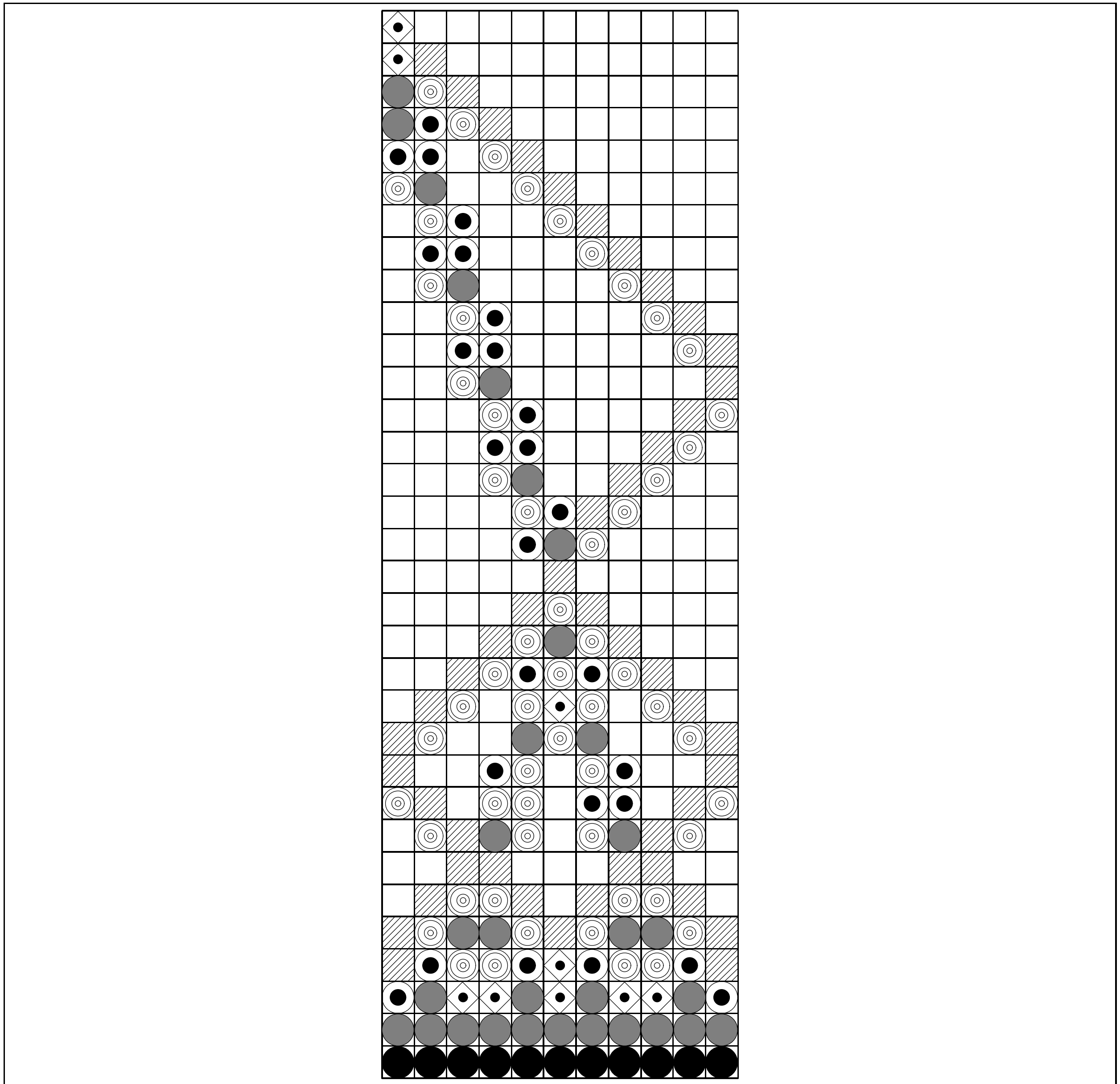


FIG. C.2 - *Synchronisation d'une ligne de longueur 11 avec 7 états*

FIG. C.3 - *Synchronisation d'une ligne de longueur 12 avec 7 états*

Annexe D

ACLiF synchronisant en temps $3n - O(\log n)$ avec 7 états

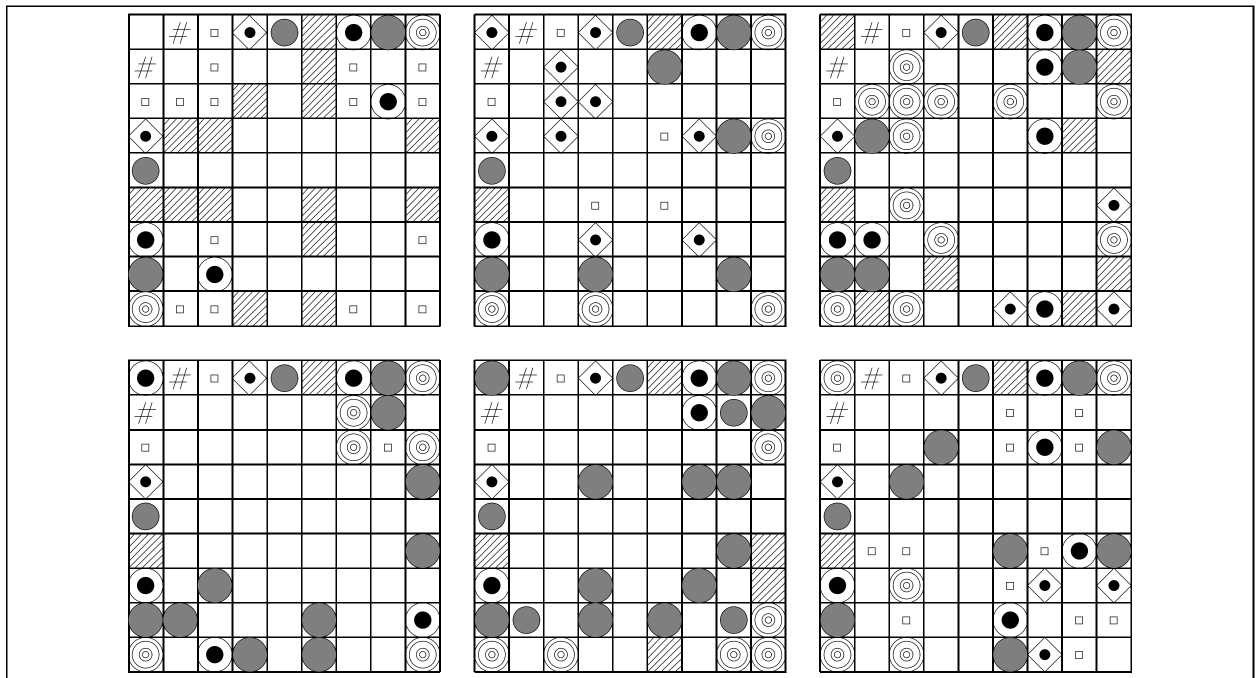


FIG. D.1 - *Automate synchronisant avec 7 états (Yunès)*

(-,.,.)↦(.)	(-,.,Z)↦(Z)	(-,.,A)↦(.)	(-,.,d)↦(.)	(.,.,-)↦(.)	(.,.,.)↦(.)
(.,.,X)↦(Z)	(.,.,Z)↦(Z)	(.,.,A)↦(.)	(.,.,C)↦(A)	(.,.,d)↦(.)	(X,.,-)↦(Z)
(X,.,.)↦(Z)	(X,.,d)↦(Z)	(Z,.,-)↦(Z)	(Z,.,.)↦(Z)	(Z,.,Z)↦(Z)	(Z,.,d)↦(Z)
(A,.,.)↦(.)	(A,.,Z)↦(Z)	(A,.,d)↦(.)	(C,.,.)↦(A)	(d,.,-)↦(.)	(d,.,.)↦(.)
(d,.,X)↦(Z)	(d,.,Z)↦(Z)	(d,.,A)↦(.)	(d,.,d)↦(.)	(-,X,.)↦(X)	(-,X,Z)↦(C)
(.,X,.)↦(X)	(.,X,X)↦(X)	(X,X,.)↦(X)	(X,X,Z)↦(.)	(X,X,A)↦(X)	(X,X,C)↦(C)
(X,X,d)↦(d)	(Z,X,X)↦(.)	(Z,X,Z)↦(.)	(A,X,X)↦(X)	(A,X,A)↦(X)	(C,X,X)↦(C)
(C,X,C)↦(C)	(d,X,X)↦(d)	(d,X,d)↦(d)	(-,Z,.)↦(d)	(-,Z,A)↦(A)	(-,Z,C)↦(C)
(-,Z,d)↦(Z)	(.,Z,-)↦(d)	(.,Z,.)↦(d)	(.,Z,X)↦(d)	(.,Z,Z)↦(d)	(.,Z,d)↦(d)
(X,Z,-)↦(C)	(X,Z,.)↦(d)	(X,Z,A)↦(A)	(X,Z,C)↦(Z)	(Z,Z,.)↦(d)	(Z,Z,d)↦(X)
(A,Z,-)↦(A)	(A,Z,X)↦(d)	(A,Z,d)↦(d)	(C,Z,-)↦(C)	(C,Z,X)↦(Z)	(C,Z,d)↦(Z)
(d,Z,-)↦(Z)	(d,Z,.)↦(d)	(d,Z,Z)↦(X)	(d,Z,A)↦(A)	(d,Z,C)↦(Z)	(d,Z,d)↦(X)
(-,A,A)↦(d)	(-,A,C)↦(C)	(.,A,A)↦(d)	(.,A,C)↦(.)	(.,A,d)↦(d)	(X,A,d)↦(C)
(Z,A,d)↦(C)	(A,A,.)↦(C)	(C,A,-)↦(C)	(C,A,Z)↦(C)	(C,A,d)↦(A)	(d,A,.)↦(A)
(d,A,X)↦(C)	(d,A,Z)↦(C)	(d,A,d)↦(d)	(-,C,A)↦(A)	(-,C,C)↦(F)	(-,C,d)↦(C)
(.,C,d)↦(d)	(X,C,X)↦(C)	(X,C,A)↦(C)	(X,C,C)↦(C)	(Z,C,C)↦(C)	(Z,C,d)↦(Z)
(A,C,X)↦(C)	(A,C,A)↦(C)	(A,C,d)↦(Z)	(C,C,-)↦(F)	(C,C,X)↦(C)	(C,C,Z)↦(C)
(C,C,C)↦(F)	(C,C,d)↦(d)	(d,C,.)↦(d)	(d,C,Z)↦(Z)	(d,C,C)↦(d)	(d,C,d)↦(d)
(-,d,Z)↦(.)	(-,d,C)↦(.)	(.,d,X)↦(C)	(.,d,Z)↦(.)	(.,d,A)↦(A)	(.,d,C)↦(.)
(.,d,d)↦(C)	(X,d,.)↦(C)	(Z,d,-)↦(.)	(Z,d,.)↦(.)	(Z,d,Z)↦(C)	(Z,d,A)↦(.)
(Z,d,C)↦(A)	(Z,d,d)↦(C)	(A,d,.)↦(d)	(A,d,Z)↦(.)	(A,d,A)↦(X)	(A,d,d)↦(X)
(C,d,.)↦(.)	(C,d,Z)↦(A)	(C,d,C)↦(.)	(C,d,d)↦(.)	(d,d,.)↦(d)	(d,d,Z)↦(C)
(d,d,A)↦(X)	(d,d,C)↦(.)				

TAB. D.1 - *Solution de Minsky (134 transitions)*

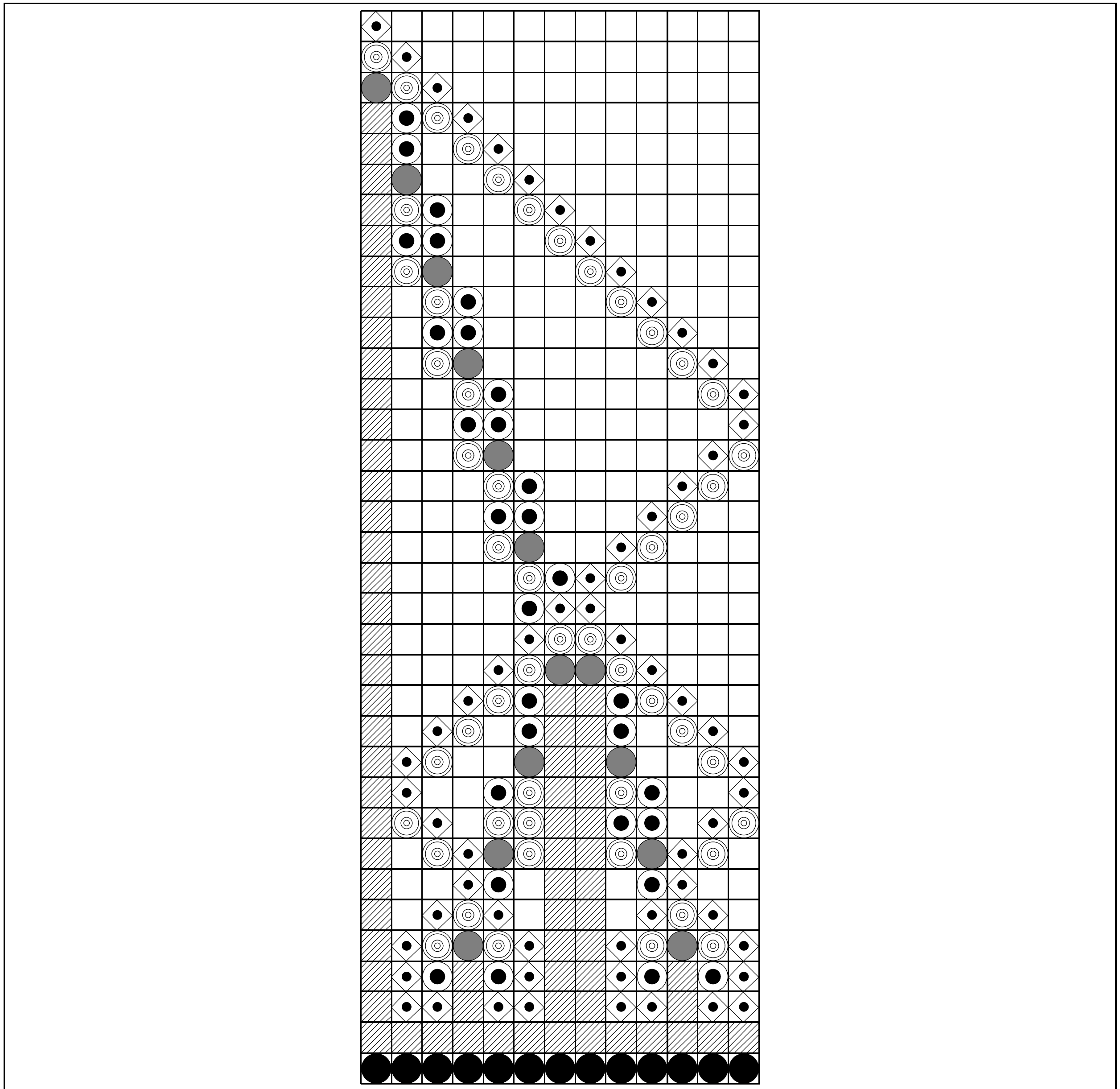


Fig. D.2 - *Synchronisation d'une ligne de longueur 13 avec 7 états*

Annexe E

Programme de Balzer

```
/**
        Programme de verification qu'il existe/n'existe pas de solution
        de synchronisation en temps minimal (pour un nombre d'etats fixe).
        Methode: Backtracking
**/
#include <stdio.h>

/* Parametres de calcul */
#define VERSION "Back. v1.0 par JBY.\n"
#define COMMENT "Un algorithme de BackTracking pour les AC.\n"
#define ALL                      /* Fournir toutes les solutions si possible !!! */
#define AFF_SYNC                 /* Affichage temps reel des solutions partielles */
#undef AFF_SYNC
#define INC                      /* Autorise le comptage des tables testees */

#ifdef AFF_SYNC                  /* Si on est en affichage temps reel */
#define STOP                     /* On s'arrete a chaque fois ? */
#undef STOP
#define SAUT 1                   /* Vision en temps reel mais pas de toutes */
#endif AFF_SYNC

#ifdef INC                       /* Si on decide de compter, qu'est ce qu'on compte ? */
#define AFF_COMPTE               /* Autorise l'affichage du decompte final */
#define COMPTEUR                 /* Permet le decompte temps reel */
#ifdef COMPTEUR
#define SAUT_COMPTEUR 10000      /* On decompte en temps reel mais quand meme */
#endif COMPTEUR
#endif INC

/* Definition des constantes du calcul */
#define N_ETATS 4                /* Nombre maximal d'etats */
#define MIN 2                    /* Plus petite longueur de ligne */
#define MAX 9                    /* Plus grande longueur de ligne */

/* Definition des etats */
#define INCONNU -1
```

```
#define BORD 0                                    /* Etat de BORD */
#define LATENT 1                                  /* Etat LATENT */
#define DEPART 2                                  /* Etat de DEPART */
#define FEU N_ETATS                               /* Etat de FEU */

#define FDT(x) (2*x-1)                            /* Fin Des Temps */

char Table[N_ETATS][N_ETATS][N_ETATS],            /* Table de transitions */
Masque[N_ETATS+1];                     /* Isomorphisme des etats non-utilises */
long compteur1, compteur2;                  /* Ben on peut compter beaucoup */
FILE *fichier;
char Dummy[MAX-MIN+1][FDT(MAX)][MAX+2],*Espace[MAX-MIN+1][FDT(MAX)];
#ifdef SAUT
long compteur;
#endif SAUT

void init(longueur)                          /* Initialisation d'une ligne */
int longueur;
{
        register int i;

        for (i=0; i<FDT(longueur); i++)
                Espace[longueur-MIN][i][-1] = Espace[longueur-MIN][i][longueur] = BORD;
        Espace[longueur-MIN][0][0] = DEPART;
        for (i=1; i<longueur; i++)
                Espace[longueur-MIN][0][i] = LATENT;
}

void affiche_solution()                      /* Affiche une table de transition */
{
        register int i, j, k;

        fputs("\nLa table suivante synchronise toutes les lignes de\n",fichier);
        fputs("longueur comprise entre ",fichier);
        fprintf(fichier,"%d et %d avec %d etats.\n",MIN,MAX,N_ETATS);
#ifdef AFF_COMPTE
        fprintf(fichier,"%9d%09d eme table calculee.\n",compteur2,compteur1);
#endif AFF_COMPTE
        for (j=1; j<N_ETATS; j++) {
                putc('a'+j,fichier);
                putc('|',fichier);
                for (i=0; i<N_ETATS; i++) putc('a'+i,fichier);
                putc(' ',fichier);
        }
        putc('\n',fichier);
        for (j=1; j<N_ETATS; j++) {
                putc('-',fichier);
                putc('+',fichier);
                for (i=0; i<N_ETATS; i++) putc('-',fichier);
                putc(' ',fichier);
        }
        putc('\n',fichier);
        for (i=0; i<N_ETATS; i++) {
```

```
            for (j=1; j<N_ETATS; j++) {
                putc('a'+i,fichier);
                putc('|',fichier);
                for (k=0; k<N_ETATS; k++)
                    putc(Table[i][j][k]==INCONNU?' ':'a'+Table[i][j][k],fichier);
                putc(' ',fichier);
            }
            putc('\n',fichier);
        }
        putc('\n',fichier);

#ifdef ALL
        return;
#else ALL
        fclose(fichier);
        exit(0);
#endif ALL
}

#ifdef INC
void inc()                         /* Permet d'avoir un compteur jusqu'a 2^32 * 10^9 */
{
        if (++compteur1%1000000000 == 0) {
            ++compteur2;
            compteur1 = 0;
        }
#ifdef COMPTEUR
        if ( compteur1%SAUT_COMPTEUR == 0) {          /* On l'affiche pas a tout les coups */
            fputs("\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b",stdout);
            printf("%9d%09d",compteur2,compteur1);
            fflush(stdout);
        }
#endif COMPTEUR
        return;
}
#endif INC

void affiche_det(time,position,longueur)                          /* Affiche un DET */
int time, position, longueur;
{
        register int i, j, c;

        if (fichier==stdout)
            fputs("\033[2J\033[0;0H",fichier);              /* Terminal dependant (VT100) */
        else
            putc('\n',fichier);
        for (i=0; i<time; i++) {
            putc('|',fichier);
            for (j=0; j<longueur; j++)
                putc((c='a'+Espace[longueur-MIN][i][j])=='b'?' ':c,fichier);
            putc('|',fichier);
            putc('\n',fichier);
        }
```

```
        putc('|',fichier);
        for (j=0; j<position; j++)
        putc((c='a'+Espace[longueur-MIN][time][j])=='b'?' ':c,fichier);
        putc('|',fichier);
        putc('\n',fichier);
        return;
}

#ifdef AFF_SYNC
void affiche_solution_partielle(longueur)
int longueur;
{
        int k;

        if (++compteur ≠ SAUT) return;                  /* On va pas tout afficher non ? */
        compteur = 0;
        fprintf(fichier,"\nSolution partielle");
        affiche_det(FDT(longueur)-1,longueur,longueur);
#ifdef STOP
        gets((char *)&k);                                /* Il vaut mieux frapper RETURN */
#endif STOP
        return;
}
#endif AFF_SYNC

/* On essaye de remplir le diagramme espace-temps */
void test(time, longueur, position)
int time, longueur, position;
{
        register char *p, i, state;
        register int flag;                  /* Indicateur utilise pour eliminer les solutions */
        /* isomorphes */

        flag = 0;                                    /* On n'a pas encore utilise d'etat vierge */

#ifdef INC
        inc();                                        /* Si il faut compter, et bien comptons */
#endif INC

loop2:
        if (time == FDT(longueur)-1) {                           /* C'est la fin des temps ? */
loop3:
                /* On pointe sur l'endroit interessant du DET */
                p = &(Table[Espace[longueur-MIN][time-1][position-1]]
                    [Espace[longueur-MIN][time-1][position]]
                    [Espace[longueur-MIN][time-1][position+1]]);
                switch(*p) {
                case INCONNU:                      /* Alors on peut essayer d'y mettre le FEU */
                        *p = Espace[longueur-MIN][time][position] = FEU;
                        Masque[FEU]++;
                        /* Peut-etre a t'on fini de synchroniser ? */
                        if (++position ≥ longueur) {
#ifdef AFF_SYNC
```

```
                                affiche_solution_partielle(longueur);
#endif AFF_SYNC
                                /* Voila une solution de plus ? */
                                if (++longueur > MAX)
                                        affiche_solution();
                                else {                              /* Ligne suivante */
                                        init(longueur);
                                        test(1,longueur,0);
                                }
                        }
                        else
                                test(time,longueur,position);
                        *p = INCONNU;
                        Masque[FEU]--;
                        return;
                case FEU:                               /* On n'a pas vraiment le choix */
                        Espace[longueur-MIN][time][position] = *p;
                        if (++position >= longueur) {
#ifdef AFF_SYNC
                                affiche_solution_partielle(longueur);
#endif AFF_SYNC
                                if (++longueur > MAX) {
                                        affiche_solution();
                                        return;
                                }
                                init(longueur);
                                position = 0;
                                time = 1;
                                goto loop;
                        }
                        goto loop3;
                default:                                    /* C'est pas bon du tout */
                        return;
                }
        }
loop:                                                   /* C'est pas la fin des temps */
        /* On pointe sur l'endroit interessant du DET */
        p = &(Table[Espace[longueur-MIN][time-1][position-1]]
            [Espace[longueur-MIN][time-1][position]]
            [Espace[longueur-MIN][time-1][position+1]]);
        switch(*p) {
        case INCONNU:                                   /* Bon ben faut tout essayer */
                for (state=LATENT; state<FEU; state++) {
                        if (!Masque[state])
                                if (flag) continue;
                                else flag = 1;
                        *p=Espace[longueur-MIN][time][position]=state;
                        Masque[state]++;
                        if (position+1 >= longueur)
                                test(time+1,longueur,0);
                        else
                                test(time,longueur,position+1);
                        Masque[state]--;
```

```
                }
                *p = INCONNU;
                return;
        case FEU:                                   /* Il vaut mieux ne pas faire FEU ici */
                return;
        default:                                    /* Il n'y a pas beaucoup de choix */
                Espace[longueur-MIN][time][position] = *p;
                if (++position ≥ longueur) {
                        time++;
                        position = 0;
                        goto loop2;
                }
                goto loop;
        }
}

void main(argc,argv)                                /* C'est le principal, non ? */
int argc;
char *argv[];
{
        int i, j, k;

        switch (argc) {
        case 1:
                fichier = stdout;
                break;
        case 2:
                fichier = fopen(argv[1],"a");
                if (fichier == NULL) {
                        printf("Erreur d'ouverture %s\n",argv[1]);
                        exit(1);
                }
                break;
        default:
                printf("Erreur d'utilisation : %s [file]\n",argv[0]);
                exit(1);
        }

        fputs(VERSION,fichier);
        fputs(COMMENT,fichier);
        fprintf(fichier,"Calcul avec %d etat.\n",N_ETATS);
        fprintf(fichier,"Longueurs comprises entre %d et %d.\n",MIN,MAX);

        compteur1 = compteur2 = 0;

        for (i=0; i≤N_ETATS; i++) Masque[i] = 0;

        /* Initialise la table */
        for (i=0; i<N_ETATS; i++)
                for (j=0; j<N_ETATS; j++)
                        for (k=0; k<N_ETATS; k++)
                                Table[i][j][k] = INCONNU;
```

```
        /* Supprime les generations spontanees */
        Table[LATENT][LATENT][LATENT] = LATENT;
        Masque[LATENT]++;
        Table[BORD][LATENT][LATENT] = LATENT;
        Masque[LATENT]++;
        Table[LATENT][LATENT][BORD] = LATENT;
        Masque[LATENT]++;

        /* Initialise le diagramme espace-temps */
        for (j=MIN; j<=MAX; j++)
                for (i=0; i<FDT(MAX); i++)
                        Espace[j-MIN][i] = &(Dummy[j-MIN][i][1]);

        init(MIN);                              /* Preparation de la premiere ligne */

        test(1,MIN,0);                          /* Ben c'est parti */

        fputs("C'est termine.\n",fichier);

#ifdef AFF_COMPTE
        fprintf(fichier,"\n%9d%09d tables calculees\n",compteur2,compteur1);
#endif AFF_COMPTE
        fclose(fichier);
        exit(0);
}
```

Annexe F

Programme de Balzer modifié

```
/**
        etats_rec : BackTracking pour le test d'existence d'une fonction
                arbitraire de synchronisation
**/
#include <stdio.h>
#include <signal.h>

#define VERSION "BackTracking 2.0 (JBY) 28-04-92"

#define TOUTES_LES_SOLUTIONS                    /* Calcule-les toutes */

#define AFFICHE_TEMPS                           /* Affiche un tableau recapitulatif */

#define AFFICHE_ESPACE_TEMPS                    /* Affiche les solutions */

#define AFFICHE_TABLE                           /* Affiche les tables de transitions */

#define ETATS 3                                 /* Nombre d'etats */

#define MINLONG 2                               /* Plus petite longueur de ligne */
#define MAXLONG 8                               /* Plus grande longueur de ligne */

#if (ETATS < 3) || (ETATS > 10)
c'est une erreur
#endif

#define INCONNU -1
#define BORD 0
#define LATENT 1
#define DEPART 2
#define FEU ETATS

#define MINTIME(l) (l)

struct ligne {
        char *ligne;                            /* Configuration */
```

```
        struct ligne *suivante;                              /* Liste chainee */
        unsigned long int synchro;                           /* Nombre de solutions */
        };

char *malloc();

#if ETATS == 3
char TransD[ETATS+2] = { ' ', '$', '.', 'X', 'F' };
#else
#if ETATS == 4
char TransD[ETATS+2] = { ' ', '$', '.', 'X', 'Y', 'F' };
#else
#if ETATS == 5
char TransD[ETATS+2] = { ' ' ,'$', '.', 'X', 'Y', 'Z', 'F' };
#endif
#endif
#endif

struct ligne Esp[MAXLONG-MINLONG+1];
char Table[ETATS][ETATS][ETATS],*Trans;
unsigned long int Compteur_Table, Total_Alloue, Total_Synchro;

/**
        MYMALLOC : Permet de tenir des comptes sur la memoire allouee
**/
char *mymalloc(taille)
int taille;
{
        Total_Alloue += taille;
        return malloc(taille);
}

/**
        TRAPPE : 'Handler' de signaux permettant d'interroger le programme
                sur son etat
**/
void trappe(sig)
int sig;
{
register int i, j;
register struct ligne *l;

        printf("ATTENTION J'ai trappe le signal %d\n",sig);

        printf("J'ai compte %d tables\n",Compteur_Table);
        printf("J'ai alloue %d octets\n",Total_Alloue);
        printf("Il y a %d solutions de synchronisation ",Total_Synchro);
        printf("pour les lignes de longueur %d a %d\n",MINLONG,MAXLONG);

        for(j=MINLONG; j<=MAXLONG; j++) {
                l = &(Esp[j-MINLONG]);
                i = 1;
                printf("Longueur %d\n",j);
```

```
                while (l→suivante≠NULL) {
                        if (l→synchro) printf("t=%5d ns=%5d\n",i,l→synchro);
                        i++;
                        l = l→suivante;
                }
                if (l→synchro) printf("t=%5d ns=%5d\n",i,l→synchro);
                printf("\n");
        }
        fflush(stdout);
        if (sig==SIGBUS || sig==SIGSEGV)
                exit(1);                        /* Tout de meme, ce ne serait pas serieux */
        signal(sig,trappe);                                     /* A la prochaine fois !!! */
        return;
}

/**
        PW : Puissance
**/
long int pw(i,ex)
int i, ex;
{
        if (ex==1)
                return i;
        else {
        long int resultat;
                resultat = pw(i,ex>>1);
                return resultat*resultat*((ex&1)==0?1:i);
        }
}

/**
        AFFICHE_SOLUTION : Affiche une solution de synchronisation
**/
void affiche_solution()
{
register int lg, i, j, k, t, compteur;
register struct ligne *ligne, *tl[MAXLONG-MINLONG+1];
register char *p;

        Total_Synchro++;

#ifdef AFFICHE_TEMPS
        puts("Affichage des temps de synchronisation\n");
        for (i=0,t=0,p=(char *)Table; i<sizeof(Table); i++,p++)
                if (*p≠INCONNU) t++;
        printf("Il y a %d transitions\n",t);
        printf("longueur = ");
        for (lg=MINLONG; lg≤MAXLONG; lg++) printf("%5d ",lg);
        printf("\ntemps = ");
        for (lg=MINLONG; lg≤MAXLONG; lg++) {
                ligne = &(Esp[lg-MINLONG]);
                t = 1;
                while (ligne→ligne[1]≠FEU) {
```

```
				t++;
				ligne = ligne→suivante;
			}
			printf("%5d ",t);
		}
		puts("\n");
#endif AFFICHE_TEMPS

#ifdef AFFICHE_ESPACE_TEMPS
		puts("Affichage de la solution\n");
		for (lg=MINLONG; lg≤MAXLONG; lg++)
			tl[lg-MINLONG] = &(Esp[lg-MINLONG]);
		compteur = MAXLONG-MINLONG+1;
		while (compteur) {
			for (lg=MINLONG; lg≤MAXLONG; lg++) {
				if (tl[lg-MINLONG]==NULL)
					for (i=1; i≤lg; i++) putchar(' ');
				else {
					for (i=1; i≤lg; i++)
						putchar(Trans[tl[lg-MINLONG]→ligne[i]]);
					if (tl[lg-MINLONG]→ligne[1]==FEU) {
						tl[lg-MINLONG]=NULL;
						compteur--;
					}
					else
						tl[lg-MINLONG] = tl[lg-MINLONG]→suivante;
				}
				putchar(' ');
			}
			putchar('\n');
		}
		puts("");
#endif AFFICHE_ESPACE_TEMPS

#ifdef AFFICHE_TABLE
		puts("Affichage de la table de transition\n");
		for (i=LATENT; i<FEU; i++) {
			putchar(Trans[i]); putchar('|');
				for (j=BORD; j<FEU; j++) putchar(Trans[j]);
			putchar(' '); putchar(' ');
		}
		putchar('\n');
		for (i=LATENT; i<FEU; i++) {
			putchar('-'); putchar('+');
			for (j=BORD; j<FEU; j++) putchar('-');
			putchar(' '); putchar(' ');
		}
		putchar('\n');
		for (i=BORD; i<FEU; i++) {
			for (j=LATENT; j<FEU; j++) {
				putchar(Trans[i]); putchar('|');
				for (k=BORD; k<FEU; k++) putchar(Trans[Table[i][j][k]]);
				putchar(' '); putchar(' ');
```

```
                }
                putchar('\n');
        }
        puts("");
#endif AFFICHE_TABLE
        puts("****************************");
}

/**
        CALCUL : Parcours du diagramme espace-temps
**/
void calcul(tps,pos,lg,tsync,l)
unsigned long int tps, pos, lg, tsync;
struct ligne *l;
{
register char *c;
register unsigned long int ts;

recommence:
        if (l->suivante==NULL) {                        /* On s'alloue de l'espace si necessaire */
                register int i;

                l->suivante = (struct ligne *)mymalloc(sizeof(struct ligne));
                if (l->suivante==NULL) {
                        fprintf(stderr,"Erreur d'allocation\n");
                        exit(1);
                }
                l->suivante->ligne = mymalloc(lg+2);
                if (l->suivante->ligne==NULL) {
                        fprintf(stderr,"Erreur d'allocation\n");
                        exit(1);
                }
                l->suivante->ligne[0] = l->suivante->ligne[lg+1] = BORD;
                for (i=1; i<lg+1; i++) l->suivante->ligne[i] = INCONNU;
                l->suivante->suivante = NULL;
        }
        if (tps==tsync) {                               /* Il est temps de synchroniser */
suivante_synchro:
                c = Table[l->ligne[pos-1]][l->ligne[pos]]+l->ligne[pos+1];
                if (pos==1) {                                   /* Debut de la ligne */
                        switch(*c) {
                        case FEU:                               /* Ben il n'y a pas le choix */
                                l->suivante->ligne[pos] = *c;
                                pos++;
                                goto suivante_synchro;
                        case INCONNU:                   /* Bon on va essayer de mettre le FEU */
                                *c = l->suivante->ligne[pos] = FEU;
                                Compteur_Table++;
                                calcul(tps,pos+1,lg,tsync,l);
                                *c = INCONNU;
                                return;
                        default:                                /* C'est pas la peine de continuer */
                                return;
```

```
				}
			}
			else if (pos==lg) {						/* Fin de la ligne */
				switch(*c) {
				case FEU:				/* Haha ! Ca marche. Suivante ! */
					l→suivante→ligne[pos] = *c;
					l→suivante→synchro++;
					if (lg==MAXLONG) {
						affiche_solution();
#ifdef TOUTES_LES_SOLUTIONS
						return;
#else TOUTES_LES_SOLUTIONS
						exit(0);
#endif TOUTES_LES_SOLUTIONS
					}
					lg++;
					for (ts=MINTIME(lg); ts≤pw(ETATS,lg); ts++)
						calcul(2,1,lg,ts,Esp+lg-MINLONG);
					return;
				case INCONNU:				/* Y'a plus qu'a mettre le FEU. */
					*c = l→suivante→ligne[pos] = FEU;
					Compteur_Table++;
					l→suivante→synchro++;
					if (lg==MAXLONG) {
						affiche_solution();
#ifdef TOUTES_LES_SOLUTIONS
						*c = INCONNU;
						return;
#else TOUTES_LES_SOLUTIONS
						exit(0);
#endif TOUTES_LES_SOLUTIONS
					}
					lg++;
					for (ts=MINTIME(lg); ts≤pw(ETATS,lg); ts++)
						calcul(2,1,lg,ts,Esp+lg-MINLONG);
					*c = INCONNU;
					return;
				default:				/* Y'a plus rien d'interessant */
					return;
				}
			}
			else {							/* Milieu de ligne */
				switch(*c) {
				case FEU:				/* Bien, ca brule presque partout */
					l→suivante→ligne[pos] = FEU;
					pos++;
					goto suivante_synchro;
				case INCONNU:			/* On met le FEU ici, et on continue */
					*c = l→suivante→ligne[pos] = FEU;
					Compteur_Table++;
					calcul(tps,pos+1,lg,tsync,l);
					*c = INCONNU;
					return;
```

```
                        default:                               /* Un coup pour rien */
                                return;
                        }
                }
        }
suivante_non_synchro:                          /* C'est pas l'heure de synchroniser */
        c = Table[l→ligne[pos-1]][l→ligne[pos]]+l→ligne[pos+1];
        switch (*c) {
        case INCONNU:                    /* Bon ben on essaye les transitions possibles */
                if (pos==lg) {
                        for (*c=LATENT; (*c)<FEU; (*c)++) {
                                l→suivante→ligne[pos] = *c;
                                /* On pourrait verifier l'abscence de cycle */
                                /* Mais c'est un peu cher, non ? */
                                Compteur_Table++;
                                calcul(tps+1,1,lg,tsync,l→suivante);
                        }
                }
                else {
                        for (*c=LATENT; (*c)<FEU; (*c)++) {
                                l→suivante→ligne[pos] = *c;
                                Compteur_Table++;
                                calcul(tps,pos+1,lg,tsync,l);
                        }
                }
                *c = INCONNU;
                return;
        case FEU:                               /* C'est pas le moment d'y mettre le FEU */
                return;
        default:                         /* Bon ben on continue avec le reste de la ligne */
                l→suivante→ligne[pos] = *c;
                if (pos==lg) {
                        tps++; pos=1; l = l→suivante;
                        goto recommence;
                }
                pos++;
                goto suivante_non_synchro;
        }
}

/**
        USAGE : Comment ca marche ?
**/
void usage(help_flag,com)
int help_flag;
char *com;
{
        if (help_flag==0)
                printf("Usage is %s [-h]\n",com);
        else {
                fputs(com,stdout);
                puts(" -h affiche cette aide");
        }
```

```
}

/**
        MAIN : Rien de plus important
**/
void main(argc,argv)
int argc;
char *argv[];
{
extern char *optarg;
extern int optind;
register int c, j, ii, jj, kk;
unsigned long int i, ts, lg;
struct ligne *l;

        puts(VERSION);
        /**
                Initialisations
        **/
        Trans = TransD+1;
        Compteur_Table = Total_Alloue = Total_Synchro = 0;
        lg = MINLONG;
        for (ii=0; ii<ETATS; ii++)                                  /* On vide la table */
                for (jj=0; jj<ETATS; jj++)
                        for (kk=0; kk<ETATS; kk++)
                                Table[ii][jj][kk] = INCONNU;
        Table[BORD ][LATENT][LATENT] = LATENT;
        Table[LATENT][LATENT][LATENT] = LATENT;
        Table[LATENT][LATENT][BORD ] = LATENT;
        for (ii=MINLONG; ii≤MAXLONG; ii++) {                        /* Initialisation du DET */
                Esp[ii-MINLONG].ligne = mymalloc(ii+2);
                Esp[ii-MINLONG].ligne[0] = BORD;
                Esp[ii-MINLONG].ligne[1] = DEPART;
                for (jj=2; jj≤ii; jj++)
                        Esp[ii-MINLONG].ligne[jj] = LATENT;
                Esp[ii-MINLONG].ligne[ii+1] = BORD;
                Esp[ii-MINLONG].suivante = NULL;
        }

        do {                                        /* Detection des arguments et traitement */
                while ((c=getopt(argc,argv,"h"))≠EOF) {
                        switch(c) {
                                case 'h':
                                        usage(1,argv[0]);
                                        exit(1);
                                case '?':
                                default:
                                        usage(0,argv[0]);
                                        exit(1);
                        }
                }
        if (argc==optind) break;
        optind++;
```

```
    } while (1);

    printf("Commande ");
    for (i=0; i<argc; i++) printf("%s ",argv[i]);
    printf("\n");

    for (i=1; i≤NSIG; i++) if (i≠SIGTSTP) signal(i,trappe);

    for (ts=MINTIME(lg); ts≤pw(ETATS,lg); ts++)
        calcul( 2L                                  /* temps */,
                1L                                  /* position */,
                lg                                  /* longueur */,
                ts                                  /* temps de synchronisation */,
                Esp                                 /* Espace de travail */);

    printf("J'ai compte %d tables\n",Compteur_Table);
    printf("J'ai alloue %d octets\n",Total_Alloue);
    printf("Il y a %d solutions de synchronisation ",Total_Synchro);
    printf("pour les lignes de longueur %d a %d\n",MINLONG,MAXLONG);

    for(j=MINLONG; j≤MAXLONG; j++) {
        l = &(Esp[j-MINLONG]);
        i = 1;
        printf("Longueur %d\n",j);
        while (l→suivante≠NULL) {
            if (l→synchro) printf("t=%5d ns=%5d\n",i,l→synchro);
            i++;
            l = l→suivante;
        }
        if (l→synchro) printf("t=%5d ns=%5d\n",i,l→synchro);
        printf("\n");
    }
    exit(0);
}
```

Bibliographie

[1] Adamides, E., Tsalides, P., and Thanailakis, A. Synchronization of asynchronous concurrent process using cellular automata. *Parallel Computing 11* (1989), 163–169.

[2] Balzer, R. Search for a solution: A case study. Tech. rep., RAND Corporation,????

[3] Balzer, R. An 8–state minimal time solution to the firing squad synchronization problem. *Information and Control 10* (1967), 22–42.

[4] Culik, K. Variations of the firing squad problem and applications. *Information Processing Letters 30* (1989), 153–157.

[5] Fischer, P. C. Generation of primes by a one-dimensional real-time iterative array. *Journal of the Association for Computing Machinery 12* (July 1965), 388–394.

[6] Grasselli, A. Synchronization of cellular arrays: The firing squad problem in two dimensions. *Information and Control 28* (1975), 113–124.

[7] Hardy, G. H., and Wright, E. *An Introduction to the Theory of Numbers*, quatrième ed. Oxford Press, 1975.

[8] Herman, G., and Liu, W. The daughter of celia, the french flag, and the firing squad. *Simulation 21* (1973), 33–41.

[9] Herman, G., Liu, W., Rowland, S., and Walker, A. Synchronization of growing cellular automata. *Information and Control 25* (1974), 103–122.

[10] Jen, E. Cylindrical cellular automata. *Communications in Mathematical Physics* (1988), 569–590.

[11] Jen, E. Linear cellular automata and recurring sequences in finite fields. *Communications in Mathematical Physics* (1988), 13–28.

[12] Jiang, T. The synchronization of nonuniform networks of finite automata (extended abstract). *FOCS* (1989), 376–381.

[13] Jiang, T. The synchronization of nonuniform networks of finite automata. *Information and Control 97* (1992), 234–261.

[14] KNUTH, D. P. *Fundamental Algorithms*, second ed., vol. 1 of *The Art of Computer Programming.* Prentice-Hall, 1973.

[15] KNUTH, D. P. *Sorting and Searching*, first ed., vol. 3 of *The Art of Computer Programming.* Prentice-Hall, 1973.

[16] KNUTH, D. P. *Semi-Numerical Algorithms*, second ed., vol. 2 of *The Art of Computer Programming.* Prentice-Hall, 1981.

[17] KOBAYASHI, K. The firing squad synchronisation problem for two-dimensional arrays. *Information and Control 34* (1977), 177–197.

[18] KOBAYASHI, K. The firing squad synchronization problem for a class of poly-automata networks. *Journal of Computer and System Sciences 17* (1978), 300–318.

[19] KOBAYASHI, K. On the minimal firing time of the firing squad synchronization problem for poly-automata networks. *Theoretical Comput. Sci. 7* (1978), 149–167.

[20] MAZOYER, J. A six states minimal time solution to the firing squad synchronization problem. *Theoretical Comput. Sci. 50* (1987), 183–238.

[21] MAZOYER, J. The linear firing squad synchronization problem: Some new solutions and applications. Tech. rep., Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1989.

[22] MAZOYER, J. A minimal time solution to the firing squad synchronization problem with only one bit of information exchanged. Tech. Rep. 89.03, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, avril 1989.

[23] MAZOYER, J. Synchronization of two interacting finite automata. Tech. rep., Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1989.

[24] MAZOYER, J. Synchronization of a line of finite automata with non-uniform delays. Tech. rep., Laboratoire de l'Informatique du parallélisme, Ecole Normale Supérieure de Lyon, 1992.

[25] MINSKY, M. *Computation : Finite and Infinite Machines.* Prentice-Hall, 1967.

[26] MOORE, E., AND LANGDON, G. A generalized firing squad problem. *Information and Control 12* (1968), 212–220.

[27] MOORE, E. E. *Sequential machines, Selected papers.* Addison Wesley, 1964.

[28] REIMEN, N. Superposable treillis automata. Tech. Rep. 92.11, Institut Blaise Pascal, L.I.T.P. 2, Place Jussieu 75005 Paris (France), février 1992.

[29] ROMANI, F. Cellular automata synchronization. *Information Sciences 10* (1976), 299–318.

[30] Romani, F. The parallelism principle: Speeding up the cellular automata synchronization. *Information and Control 36* (1978), 245–255.

[31] Rosenstiehl, P., Fiskel, J., and Holliger, A. *Intelligent Graphs: Networks of Finite Automata capable of Solving Graph Problems.* Graph Theory and Computing (R.C. Read Ed.) Academic Press, 1972.

[32] Shinahr, I. Two and three dimensional firing squad synchronization problems. *Information and Control 24* (1974), 163–180.

[33] Szwerinski, H. Time-optimal solution of the firing-squad synchronization problem for n-dimensional rectangles with the general at an arbitrary position. *Theoretical Comput. Sci. 19* (1982), 305–320.

[34] Terrier, V. *Temps réel sur automates cellulaires.* PhD thesis, Ecole Normale Supérieure de Lyon, 1991.

[35] Varshavsky, V., Marakhovsky, V., and Peshansky, V. Synchronization of interacting automata. *Mathematical System Theory 4 n.3* (1969), 212–230.

[36] Waksman, A. An optimum solution to the firing squad synchronization problem. *Information and Control 9* (1966), 66–78.

[37] Wolfram, S., Martin, O., and Odlyzko, A. M. Algebraic properties of cellular automata. *Communications in Mathematical Physics* (1984), 219–259.

[38] Yunès, J.-B. Seven states solutions to the firing squad synchronization problem. *Theorical Computer Science* (1992). à paraître.

Table des figures

1.1 Solution de type Minsky 15
1.2 Solution de type Waksman-Balzer 16
1.3 Solution de type Mazoyer 17

2.1 Execution of an LCA with 13 states and 14 cells 22
2.2 Execution of an LCA with 7 states and 11 cells 28
2.3 Execution of an LCA with 7 states and 12 cells 29
2.4 Execution of an LCA with 7 states and 13 cells 32

3.1 Longueurs de cycles 43
3.2 Longueur des cycles calculés par `Xor1` 47
3.3 Equivalence entre un ACLiF et un ACT 48
3.4 Exemples de configurations atomiques 51
3.5 Itération géométrique 54
3.6 Automates cellulaires 57
3.7 Exemples 59
3.8 Construction d'un treillis 60

A.1 Automate synchronisant avec 9 états 68
A.2 Synchronisation d'une ligne de longueur 23 avec 9 états 69

B.1 Automate synchronisant avec 13 états (Yunès) 71
B.2 Synchronisation d'une ligne de longueur 14 avec 13 états 73

C.1 Automate synchronisant avec 7 états (Yunès) 75
C.2 Synchronisation d'une ligne de longueur 11 avec 7 états 77
C.3 Synchronisation d'une ligne de longueur 12 avec 7 états 78

D.1 Automate synchronisant avec 7 états (Yunès) 79
D.2 Synchronisation d'une ligne de longueur 13 avec 7 états 81

Liste des tableaux

2.1 Propagation of signal $\mathcal{S}_1$ 23
2.2 A bounce of signal $\mathcal{S}_1$ 23
2.3 Propagation of signal $\mathcal{S}_3$ 24
2.4 An “even” cut 25
2.5 An “odd” cut 25
2.6 An “optimized odd” cut 26
2.7 $\mathcal{S}_3$ with 4 states 27
2.8 The new $\mathcal{S}_1$ 30
2.9 $\mathcal{S}_3$ with 3 states 30
2.10 Transition table for a solution with 7 states 31
2.11 Transition table for another solution with 7 states 33

3.1 Nombre d'automates avec n états 40
3.2 L'état du problème 41
3.3 L'état du problème 41
3.4 Longueurs de cycles 44
3.5 Les 4 automates à long cycle 45
3.6 Longueur des cycles calculés par `Xor1` 46

A.1 Transitions de l'automate synchronisant avec 9 états (165 transitions) 67

B.1 Solution de Minsky (130 transitions) 72

C.1 Solution de Minsky (134 transitions) 76

D.1 Solution de Minsky (134 transitions) 80

Table des matières

1 La synchronisation d'une ligne de fusiliers **5**
1.1 Définitions . . . 6
1.2 Généralisations du FSSP . . . 9
1.2.1 dans les lignes d'automates . . . 9
1.2.2 dans des réseaux de topologie régulière . . . 11
1.2.3 dans des graphes quelconques . . . 12
1.3 Les différentes méthodes de synchronisation . . . 12
1.3.1 La méthode de Minsky . . . 12
1.3.2 La méthode de Waksman-Balzer . . . 13
1.3.3 La méthode de Mazoyer . . . 14

2 Une approche descendante **19**
2.1 Introduction . . . 19
2.2 Definitions . . . 20
2.2.1 Practical definitions . . . 20
2.3 MINSKY's solution . . . 20
2.4 A 13-states solution . . . 21
2.4.1 How many states? . . . 21
2.4.2 How to propagate $\mathcal{S}_1$? . . . 23
2.4.3 How to propagate $\mathcal{S}_3$? . . . 23
2.4.4 Bounces . . . 24
2.4.5 Cuts . . . 24
2.4.6 Ignition of signals . . . 26
2.4.7 Synchronization . . . 26

2.4.8 The synchronization time 26

2.4.9 Thickness 27

2.5 A 7-states solution 27

2.5.1 Can we eliminate the auxiliary states? 27

2.5.2 $\mathcal{S}_3$ with fewer states? 27

2.5.3 $\mathcal{S}_1$ with fewer states? 27

2.5.4 A better way to construct $\mathcal{S}_3$ 30

2.5.5 The synchronization time 30

2.5.6 Note 30

2.5.7 Thickness 31

2.6 Another 7-states solution 31

2.6.1 The synchronization time 33

2.7 Verifying Balzer's conditions 34

2.8 Some mathematical results 35

3 Une approche ascendante **39**

3.1 Un nouveau programme de *Balzer* 40

3.2 Un programme de *Balzer* modifié 41

3.3 Observation des résultats du programme de Balzer modifié 43

3.4 La loi `Xor1` 45

3.4.1 Equivalence avec un automate torique 46

3.4.2 Equivalence avec des automates treillis 56

3.4.3 Pavage de l'espace-temps 59

3.4.4 Conclusion 64

3.5 La loi `Xor4` 64

3.5.1 Partitionnement de l'espace-temps 64

3.5.2 Equivalence avec la loi `Xor1` 65

3.5.3 Conclusion 66

A ACLiF synchronisant en temps $3n-3$ avec 9 états **67**

B ACLiF synchronisant en temps $3n+O(\log n)$ avec 13 états **71**

C ACLiF synchronisant en temps $3n + O(\log n)$ avec 7 états **75**

D ACLiF synchronisant en temps $3n - O(\log n)$ avec 7 états **79**

E Programme de Balzer **83**

F Programme de Balzer modifié **91**