Implémentation d'un Automate Cellulaire 2D de Fredkin sur Arduino : Conception d'une Interface Graphique et Analyse des Défis Techniques

Présenté par **MAKHEZER Mohamed Anis**

ALLOUCHE Mohamed Abdelmalek

Jury : **Sieiler Loïc**
Habert Olivier

Année Universitaire : **2024 – 2025**

Rapport de projet de Master 1 EEA – Mesure et traitement de l'information

I. Remerciement

Nous tenons à exprimer notre profonde gratitude à M. Siéler, notre tuteur, pour son soutien constant, ses conseils avisés et son engagement tout au long de ce projet. Ses remarques constructives et son expertise ont grandement contribué à la réussite de notre travail.

Nous remercions également l'ensemble des enseignants et personnels de l'Université de Lorraine, et plus particulièrement l'UFR Scifa, pour la qualité de leur enseignement et pour avoir créé un environnement stimulant et propice à l'apprentissage.

Nos remerciements vont aussi à nos proches et à nos collègues pour leur soutien moral et leur patience durant les périodes intenses de travail. Leur encouragement nous a permis de surmonter les difficultés et de mener ce projet à bien.

Enfin, nous adressons nos sincères remerciements à toutes les personnes qui, de près ou de loin, ont contribué à la réalisation de ce rapport et à l'avancement de notre projet.

MAKHEZER Mohamed Anis – ALLOUCHE Mohamed Abdelmalek

Table d’illustration :

II. Introduction :

a) Contexte et justification :

Les automates cellulaires représentent un domaine passionnant situé à la croisée des mathématiques et de l'informatique théorique. Ce sont des systèmes discrets constitués d'une grille de cellules qui évoluent simultanément selon des règles locales simples. Grâce à cette approche, il est possible de décrire de manière rigoureuse une grande variété de phénomènes complexes. Leur capacité à générer des dynamiques globales à partir d'interactions élémentaires en fait un outil précieux pour étudier comment des règles simples peuvent donner naissance à une diversité de comportements riches et parfois imprévisibles. Par exemple, on peut simuler la propagation d'épidémies, la formation de cristaux ou encore modéliser certains systèmes biologiques et physiques, parmi d'autres applications.

Dans le cadre de notre projet en Master 1 EEA, nous avons décidé de nous intéresser à un automate cellulaire particulier : celui de Fredkin. Cet automate se distingue par une règle de transition basée sur un compteur de parité. Autrement dit, pour chaque cellule, le nombre de cellules actives dans son voisinage détermine son état futur. Malgré sa simplicité de formulation, cette règle a la capacité de générer des comportements très riches, ce qui en fait un sujet d'étude particulièrement captivant.

Le projet s'appuie sur une plateforme Arduino, avec une carte Arduino Mega 2560 associée à un écran TFT de 3,5 pouces. Ce choix de matériel permet de concrétiser les concepts théoriques sur un support interactif, grâce à la mise en place d'une interface graphique qui offre une visualisation en temps réel de l'évolution de l'automate. Par ailleurs, des contraintes techniques importantes – telles que la synchronisation des données et la mise en œuvre d'un système d'enregistrement des logs – viennent ajouter une dimension stimulante et rigoureuse à ce projet.

Ce rapport a pour objectif de détailler la démarche adoptée pour concevoir et réaliser cette interface graphique dédiée à la simulation de l'automate cellulaire de Fredkin. Nous y exposerons les fondements théoriques, les choix techniques effectués ainsi que les défis rencontrés tout au long du processus.

b) Objectifs du projet :

Conception de l'interface graphique

• Élaborer une interface qui offre une vision claire et dynamique de l'évolution de l'automate, permettant à l'utilisateur d'observer en temps réel la transition des états des cellules.

• Utiliser des éléments visuels variés – schémas, diagrammes et animations – pour illustrer de manière intuitive les changements d'état et faciliter la compréhension du processus de transition.

• Mettre en œuvre un système ergonomique qui simplifie la prise en main des paramètres de simulation (choix du type de Fredkin, nombre de générations, configuration initiale, etc.)

Gestion de l'affichage et synchronisation

• Assurer un affichage optimal en garantissant que les mises à jour de l'état de la grille s'effectuent sans délai perceptible, pour une visualisation fluide des événements.

• Déployer des mécanismes de synchronisation efficaces afin de coordonner l'envoi des données depuis la carte Arduino avec leur réception par l'interface graphique, assurant ainsi une cohésion parfaite durant la simulation.

• Optimiser le flux de données pour éviter toute saturation ou décalage qui pourrait altérer la fluidité de l'affichage.

Enregistrement et suivi des logs

• Implémenter une solution robuste pour automatiser l'archivage des logs, garantissant la conservation d'une trace complète et détaillée des différentes étapes et générations de la simulation.

• Offrir à l'utilisateur la possibilité de consulter et d'exporter ces logs, afin de faciliter l'analyse ultérieure du comportement de l'automate et d'identifier d'éventuels problèmes techniques ou pistes d'amélioration.

Validation et tests

• Développer une série de tests pour vérifier la bonne exécution de la simulation sur la plateforme Arduino, en particulier pour valider la synchronisation et la performance de l'interface graphique.

Ces objectifs, articulés autour de la création d'un outil visuel performant, mettent en lumière l'importance de rendre interactif et accessible un concept complexe, tout en répondant aux contraintes techniques inhérentes à l'implémentation sur une plateforme embarquée comme Arduino. Ce projet se propose ainsi de démontrer que, grâce à une conception rigoureuse et à l'intégration de solutions logicielles adaptées, il est possible de transposer avec succès les principes théoriques des automates cellulaires en une application concrète et pédagogique.

c) Organisation du rapport :

Le présent document est organisé de manière à guider le lecteur à travers chaque étape du projet de façon logique et progressive. Dans un premier temps, nous introduisons les automates cellulaires en exposant leurs fondements théoriques, leur évolution historique et en illustrant quelques exemples emblématiques. Nous consacrons ensuite une section dédiée aux particularités de l'automate de Fredkin, en mettant en lumière ses règles de transition distinctives et ses diverses applications potentielles.

La deuxième partie se focalise sur l'état de l'art et l'analyse des travaux antérieurs, ce qui permet de situer notre démarche méthodologique dans un contexte plus large et de justifier les choix techniques effectués. Enfin, la troisième section décrit de manière détaillée la mise en œuvre du projet. Nous y présentons le cahier des charges, le choix du matériel, le développement de l'interface graphique ainsi que les différentes étapes de réalisation, en abordant les problématiques rencontrées et les solutions apportées.

La démarche adoptée repose sur une stratégie itérative et expérimentale, combinant une analyse théorique approfondie et des essais pratiques sur la plateforme Arduino. Cette approche a permis d'ajuster progressivement la conception et de vérifier individuellement chaque composant du système avant de les intégrer dans l'ensemble final.

III. Théorie des Automates Cellulaires :

a) Définition générale :

Un automate cellulaire est un modèle mathématique et informatique permettant de simuler des systèmes dynamiques complexes à partir de règles locales simples. Développé initialement par John von Neumann et Stanislaw Ulam dans les années 1940, ce concept repose sur une grille de cellules pouvant être unidimensionnelle, bidimensionnelle ou tridimensionnelle, où chaque cellule adopte un état parmi un ensemble fini, comme actif/inactif ou vivant/mort (Neumann, 1966)

L'évolution de ces cellules s'effectue de manière synchronisée : à chaque étape temporelle, chaque cellule met à jour son état en fonction de son propre état et de celui de ses voisines. Cette simultanéité garantit que l'évolution du système repose uniquement sur l'état précédent, sans interférences dues à des mises à jour successives désynchronisées, un principe clé étudié notamment par Stephen Wolfram dans ses recherches sur les automates cellulaires élémentaires. (Stephen Wolfram)

Les règles locales définissent les transformations de chaque cellule en fonction de son voisinage. Par exemple, une cellule peut devenir active si un certain nombre de voisines sont actives ou, à l'inverse, rester inactive si ce seuil n'est pas atteint. Lorsqu'elles sont appliquées à grande échelle et sur plusieurs générations, ces règles donnent naissance à des comportements émergents, où des structures complexes apparaissent à partir d'interactions élémentaires, un phénomène exploré par Christopher Langton dans ses travaux sur la complexité computationnelle des automates cellulaires (Langton, 1986; Mitchelle, 2009)

Ce phénomène, central dans la théorie des systèmes complexes, permet d'expliquer comment des modèles simples peuvent générer des comportements globaux riches et variés. Les automates cellulaires ont ainsi été largement utilisés pour modéliser des phénomènes naturels et artificiels, comme la croissance des cristaux, la propagation d'épidémies, l'évolution des écosystèmes et même certains phénomènes sociaux (Mitchelle, 2009)

En résumé, un automate cellulaire se caractérise par :

- une structure en grille,
- des états discrets,
- des règles locales appliquées de manière synchronisée,

Bien que basés sur des principes simples, ces modèles offrent un outil puissant pour explorer et comprendre une grande variété de systèmes naturels et artificiels.

b) Historique et évolution :

L'histoire des automates cellulaires remonte aux années 1940, lorsque Stanislaw Ulam et John von Neumann ont exploré les principes de la complexité et de la reproduction à partir de règles simplifiées. Ulam, influencé par la formation des structures cristallines, a utilisé des grilles pour simuler ces phénomènes (Ulam, 1952). De son côté, von Neumann, motivé par la recherche sur l'autoréplication, a élaboré un modèle mathématique capable de reproduire son propre état (Neumann, 1966). Ces travaux menés à Los Alamos ont posé les bases du concept des automates cellulaires en démontrant comment des interactions locales pouvaient générer une organisation globale.

Dans les années 1960, le concept a été affiné et popularisé, notamment grâce aux recherches de John Conway, qui a introduit dans les années 1970 le célèbre Jeu de la Vie (Gardner, 1970) .Cet automate cellulaire bidimensionnel, basé sur des règles simples, a révélé des comportements émergents d'une grande complexité. Ce modèle a marqué un tournant en illustrant de manière spectaculaire comment des structures dynamiques pouvaient apparaître spontanément à partir de règles élémentaires, attirant ainsi l'attention des scientifiques et du grand public (Elwyn R. Berlekamp, 1982).

Dans les années 1980, Stephen Wolfram a mené une étude systématique des automates cellulaires en explorant leurs comportements par des expérimentations numériques massives. Dans son ouvrage *A New Kind of Science*, il propose une classification des automates en plusieurs catégories, distinguant les systèmes stables, cycliques, chaotiques et complexes (Stephen Wolfram). Ses travaux ont mis en évidence la diversité des dynamiques observables dans ces modèles, révélant leur potentiel en tant qu'outils de modélisation dans des domaines aussi variés que la physique, la biologie et les sciences sociales (Ilachinski, 2001)

L'évolution des automates cellulaires illustre ainsi un cheminement intellectuel remarquable : d'un concept visant à comprendre la reproduction des structures naturelles à un outil de simulation puissant permettant d'analyser des phénomènes dynamiques complexes. Des premières recherches de Ulam et von Neumann, à l'impact révolutionnaire du Jeu de la Vie de Conway, jusqu'aux contributions théoriques et appliquées de Wolfram, chaque avancée a enrichi ce champ d'étude.

Aujourd'hui, les automates cellulaires continuent d'être exploités dans des domaines aussi divers que la biologie, la physique, la cryptographie et l'art génératif. L'héritage des pionniers se perpétue à travers des applications novatrices qui confirment qu'à partir de règles locales simples, il est possible d'engendrer une complexité émergente d'une richesse inouïe, contribuant à une meilleure compréhension des systèmes naturels et artificiels (Mitchelle, 2009)

c) Exemples classiques :

Pour illustrer les principes des automates cellulaires, nous examinerons d'abord des exemples unidimensionnels avant d'aborder un modèle classique en deux dimensions

Automates unidimensionnel:

Les automates cellulaires unidimensionnels sont des systèmes dans lesquels chaque cellule interagit avec ses voisines immédiates suivant une règle déterminée. Parmi les plus étudiés, trois règles se distinguent par leurs propriétés uniques et leur impact sur la recherche en complexité computationnelle.

- Règle 30 :

La règle 30, introduite et étudiée par Stephen Wolfram, est connue pour son comportement chaotique émergent à partir de configurations initiales simples (Stephen Wolfram). Bien que définie par une transformation binaire élémentaire, son application itérative génère des motifs imprévisibles et non répétitifs, ce qui en fait un exemple marquant de l'impact des interactions locales sur la dynamique globale des systèmes.

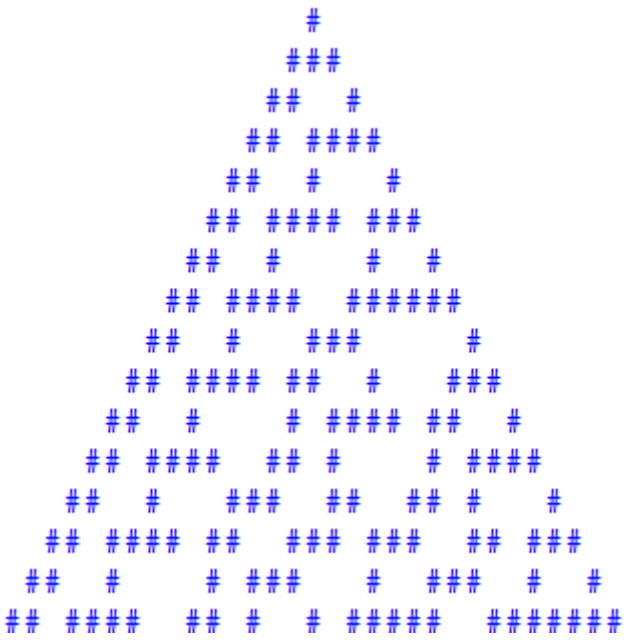

Figure 1 test python AC règle 30

- Règle 90 :

La règle 90 se distingue par la génération de motifs fractals et symétriques, démontrant ainsi comment une règle locale appliquée uniformément peut produire des structures autosimilaires à différentes échelles (Ilachinski, 2001). Cette propriété en fait un modèle d'étude précieux pour comprendre la formation de motifs naturels complexes à partir de règles élémentaires.

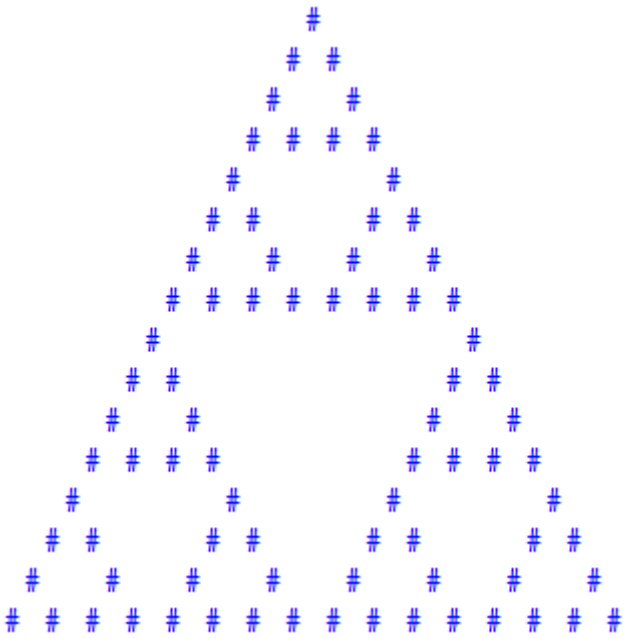

Figure 2 test python AC règle 90

- Règle 110 :

La règle 110 occupe une place exceptionnelle dans l'étude des automates cellulaires, car elle est démontrée comme étant Turing complète, c'est-à-dire capable de simuler n'importe quel

Figure 3 test python règle 110

calcul réalisable par un ordinateur universel (Cook, 2004) . Bien que son fonctionnement repose sur des transformations locales simples, elle illustre le potentiel des systèmes computationnels émergents et alimente la recherche sur l'origine de la complexité algorithmique dans les systèmes naturels et artificiels.

Exemple bidimensionnel :

Le Jeu de la Vie :

Le Jeu de la Vie, inventé par John Conway en 1970, est sans doute le modèle d'automate cellulaire bidimensionnel le plus emblématique (Gardner, 1970) . Ce système repose sur une grille où chaque cellule évolue selon des règles simples qui tiennent compte du nombre de cellules voisines en vie. Une cellule peut apparaître, survivre ou disparaître selon des critères définis, illustrant comment des motifs dynamiques et des structures autoorganisées émergent de règles élémentaires (Elwyn R. Berlekamp, 1982).

Le Jeu de la Vie est particulièrement fascinant en raison de sa capacité à produire des comportements complexes et imprévisibles. Certains motifs, tels que les oscillateurs et les vaisseaux**,** démontrent comment des structures peuvent persister et interagir sur de longues périodes, ce qui le rapproche des systèmes biologiques et des processus d'auto-organisation observés dans la nature (Mitchelle, 2009).

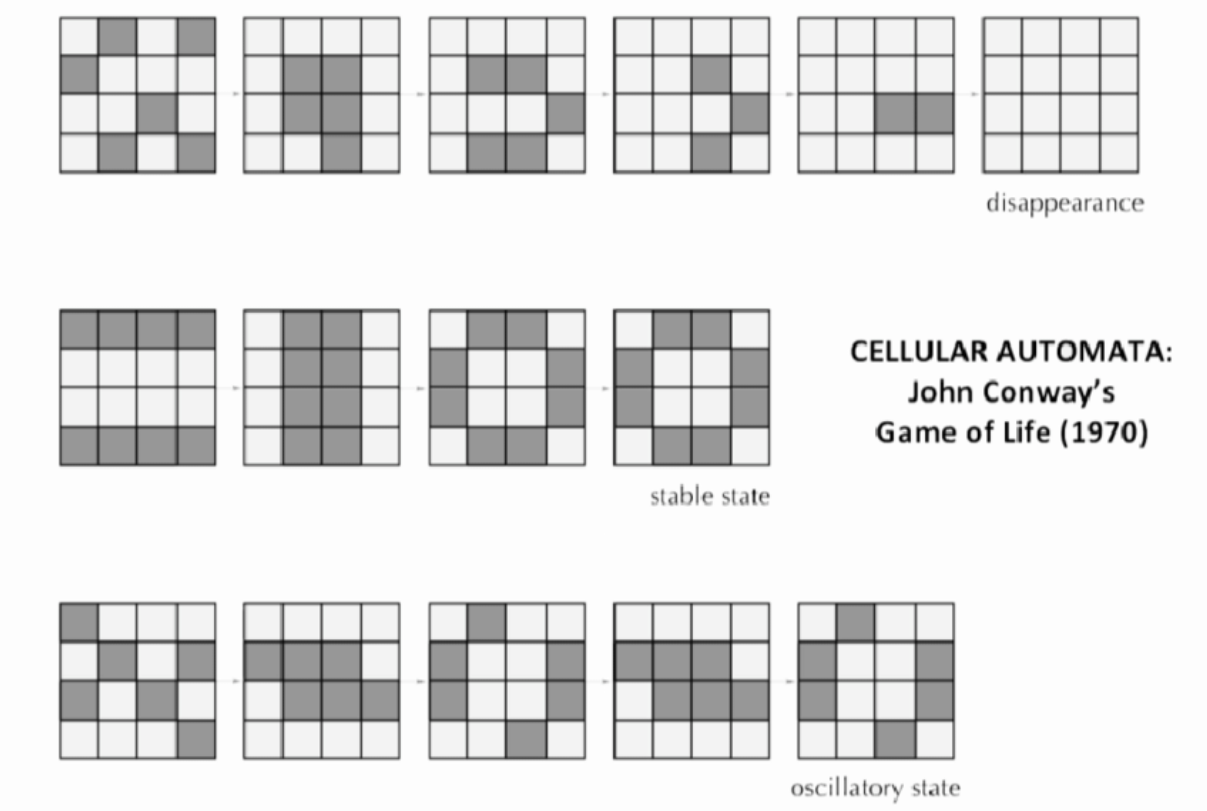

Figure 4e « jeu de la vie » de John Conway [1970]. © Irene Cazzaro.

Conclusion

Ces exemples démontrent que, qu'il s'agisse d'automates cellulaires unidimensionnels ou bidimensionnels, des règles simples peuvent conduire à l'émergence de structures et de dynamiques d'une complexité fascinante. Ils rappellent ainsi que la complexité observable dans de nombreux systèmes naturels et artificiels peut être issue d'interactions locales élémentaires, offrant ainsi des perspectives riches pour la modélisation de phénomènes complexes en biologie, en physique et en informatique.

IV. Automates Cellulaires de Fredkin

a) Spécificités de l'automate de Fredkin :

L'automate de Fredkin se distingue par une approche originale reposant sur un compteur de parité, une idée développée par Edward Fredkin, pionnier dans le domaine de l'informatique théorique (Fredkin, 1982). Son fonctionnement est basé sur un principe simple : pour chaque cellule, on examine le nombre de cellules actives dans son voisinage. Si ce nombre est impair, la cellule devient active lors de l'itération suivante ; sinon, elle reste ou devient inactive.

Ce mécanisme – qui repose uniquement sur la parité du voisinage – permet d'obtenir une dynamique non triviale à partir d'une règle élémentaire. Bien que la règle soit simple, son application répétée à l'ensemble d'une grille de cellules conduit à des évolutions riches et imprévisibles (Margolus T. T., 1987). En effet, le compteur de parité agit comme un filtre qui, en fonction du nombre de cellules actives dans l'environnement immédiat, oriente l'état futur de chaque cellule tout en maintenant une symétrie sous-jacente dans l'évolution globale du système.

L'une des spécificités majeures de cet automate réside dans sa conservation de l'information : contrairement à d'autres automates cellulaires où l'information peut se perdre au fil des générations, l'automate de Fredkin est un système réversible qui préserve l'état initial et permet, en théorie, de retrouver les étapes précédentes à partir d'un état donné (Morita, 1998). Cette propriété le rend particulièrement intéressant pour des applications en logique réversible, en informatique quantique et en modélisation de systèmes physiques conservatifs (Margolus N. , 1984).

Ainsi, l'automate de Fredkin, par sa règle basée sur la parité, illustre parfaitement comment des interactions locales élémentaires peuvent engendrer une complexité émergente. Il constitue un modèle élégant et puissant pour explorer des phénomènes liés à l'auto-organisation, au calcul parallèle et aux systèmes dynamiques conservatifs.

b) Différenciation des types :

Dans le cadre de l'automate de Fredkin, deux configurations de voisinage se distinguent, chacune influençant la dynamique du système de manière spécifique .

Configurations de voisinage :

Type 1 – Voisinage von Neumann :

Dans cette configuration, chaque cellule est influencée uniquement par ses quatre voisins

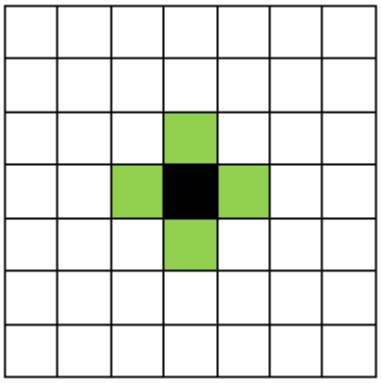

Figure 5 Voisinage von Neumann

orthogonaux : ceux situés au nord, sud, est et ouest. Ce type de voisinage, souvent appelé « voisinage à 4 cellules », limite les interactions aux directions cardinales et permet d'observer des motifs plus structurés et prévisibles (Ilachinski, 2001). La règle basée sur le compteur de parité s'applique en ne considérant que ces quatre voisins immédiats, ce qui conduit généralement à une évolution plus régulière et organisée.

Type 2 – Voisinage de Moore :

Ici, l'influence s'étend aux huit cellules adjacentes, englobant à la fois les voisins orthogonaux et diagonaux. Ce « voisinage de Moore » capture une gamme d'interactions plus large, rendant le système plus sensible aux configurations locales complexes (Stephen Wolfram). En appliquant la règle de transition basée sur la parité à cet ensemble élargi, le système tend à exhiber des dynamismes plus variés et imprévisibles, favorisant l'apparition de structures émergentes et de comportements non linéaires (Ilachinski, 2001).

Ainsi, la principale différence entre ces deux approches réside dans l'étendue du voisinage pris en compte : le type 1 se limite aux quatre voisins immédiats, tandis que le type 2 inclut huit voisins, ce qui augmente la complexité des interactions et enrichit la dynamique du système.

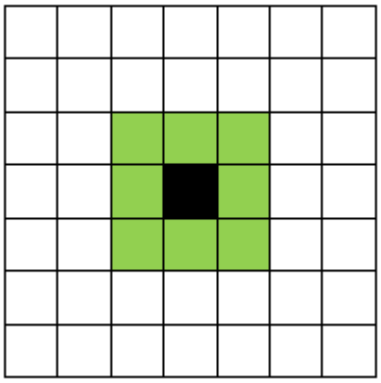

Figure 6 Voisinage de Moore

Règles de transition et fonctionnement

Le mécanisme de transition dans l'automate de Fredkin repose sur un compteur de parité, appliqué à l'ensemble des cellules voisines de chaque cellule (Fredkin, 1982).

Calcul du compteur de parité :

On examine les cellules voisines selon le voisinage von Neumann (4 voisins) ou Moore (8 voisins). On compte le nombre de cellules actives (état = 1) dans ce voisinage.

Application de la règle de parité :

Si le nombre total de cellules actives est impair, la cellule devient active ou reste active.

Si le nombre est pair, la cellule devient inactive ou reste inactive (Langton, 1986).

Mise à jour synchronisée :

Chaque cellule évalue son état simultanément avec l'ensemble du système.

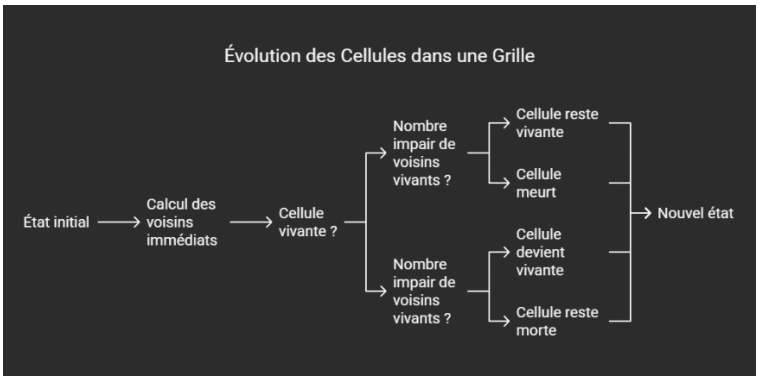

Figure 7 evolution des cellules dans une grille

Ce caractère synchronisé favorise l'émergence de motifs dynamiques et auto-organisés qui peuvent évoluer de manière complexe et surprenante (Sipper, 1997)

Bien que cette règle repose sur une logique simple, elle donne lieu à des évolutions riches et variées. Elle démontre ainsi comment, à partir de règles locales élémentaires, il est possible de générer des comportements globaux complexes, illustrant un principe clé des systèmes dynamiques et de la complexité émergente.

c) Applications et utilité :

Les automates cellulaires, grâce à leur capacité à transformer des règles locales simples en comportements globaux complexes**,** se prêtent à une grande variété d'applications dans des domaines aussi divers que la physique, la biologie et l'informatique (Droz, 1998).

Simulation de systèmes complexes :

Les automates cellulaires sont couramment utilisés pour modéliser des dynamiques complexes, telles que la propagation d'épidémies, la diffusion de chaleur ou encore la croissance de structures cristallines (Macia, 2005) . Ces modèles permettent d'étudier l'émergence de motifs et de comportements imprévus, offrant ainsi un outil puissant pour la recherche en sciences naturelles et en ingénierie.

Modélisation de phénomènes physiques ou biologiques :

Dans le domaine de la physique, les automates cellulaires servent à simuler des processus de diffusion, des réactions chimiques, ou encore des systèmes de particules (Chopard & Droz, 1998, p. 82, livre). En biologie, ils permettent d'illustrer des processus tels que la

morphogenèse ou la dynamique des populations, aidant ainsi à mieux comprendre l'auto-organisation et l'évolution des systèmes vivants (Mitchelle, 2009).

Cryptographie :

Certains automates cellulaires, grâce à leur capacité à générer des séquences pseudo-aléatoires complexes à partir de règles déterministes, trouvent des applications en cryptographie (Fredkin, 1982). Ils sont exploités pour concevoir des générateurs de nombres aléatoires ou pour élaborer des systèmes de chiffrement, garantissant ainsi un niveau de sécurité basé sur la complexité émergente.

Le choix de l'automate de Fredkin pour ce projet est particulièrement pertinent. Sa règle de transition, fondée sur un compteur de parité, permet de produire des comportements à la fois simples et riches, tout en restant suffisamment intuitif pour être implémenté sur une plateforme embarquée telle que l'Arduino (Fredkin, 1982). En outre, cette approche offre un excellent compromis entre la simplicité des interactions locales et l'émergence de dynamiques globales complexes, ce qui en fait un outil pédagogique et de recherche idéal pour explorer les principes de l'auto-organisation.

Ainsi, l'automate de Fredkin se positionne comme un choix judicieux pour développer une interface graphique interactive qui non seulement illustre ces phénomènes, mais permet également d'expérimenter et d'analyser la synchronisation, la gestion de l'affichage et l'enregistrement des logs dans un environnement réel (Macia, 2005).

V. État de l'Art et Travaux Précédents

a) Revue des simulateurs existants :

Dans le domaine de la simulation des automates cellulaires, plusieurs logiciels et environnements se distinguent par leurs fonctionnalités et leur ergonomie, certains offrant même des modules spécifiques pour l'automate de Fredkin (Stephen Wolfram)

1. Golly

- Golly est un logiciel open source reconnu pour la simulation d'automates cellulaires, notamment pour le Jeu de la Vie (Gosper, 2005)
- Bien qu'il ne soit pas exclusivement dédié à l'automate de Fredkin, il permet de configurer et expérimenter une multitude de règles, incluant celles basées sur un compteur de parité.

- Sa flexibilité et ses fonctionnalités avancées en font un outil de référence pour les chercheurs explorant les comportements émergents des automates cellulaires.

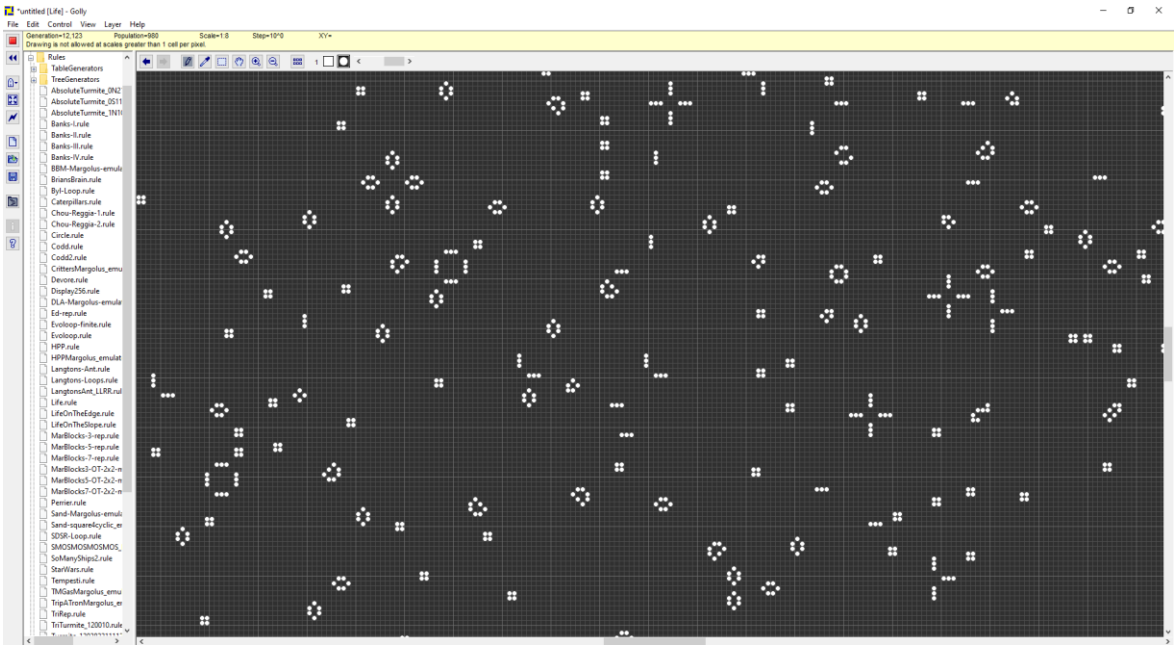

Figure 8 Logiciel Golly

2. Logiciel FiatLux

- Développé par Nazim Fatès, FiatLux est un simulateur utilisé dans des environnements de recherche, notamment au laboratoire LORIA à Nancy (Fatès, 2010)

- Il se distingue par une interface intuitive et des fonctionnalités avancées, facilitant l'expérimentation avec différents types d'automates cellulaires, dont Fredkin.

- Ce logiciel permet une visualisation claire des dynamiques et offre des options de paramétrage détaillées, permettant de tester différentes configurations et d'analyser les résultats en temps réel.

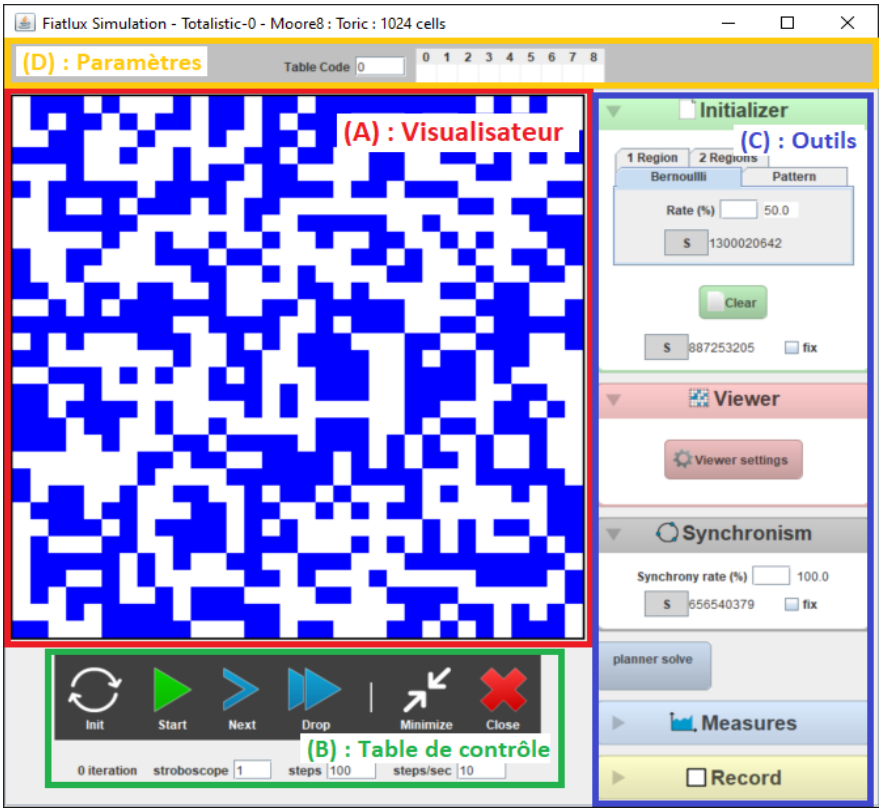

Figure 9 Logiciel FiatLux

3. Simulateur de l'Université du Mans

- Il s'agit d'une application en ligne permettant de tester et visualiser divers automates cellulaires directement via un navigateur (Mans, 2015)Bien que non exclusivement dédié à Fredkin, il offre des options avancées permettant de simuler plusieurs règles

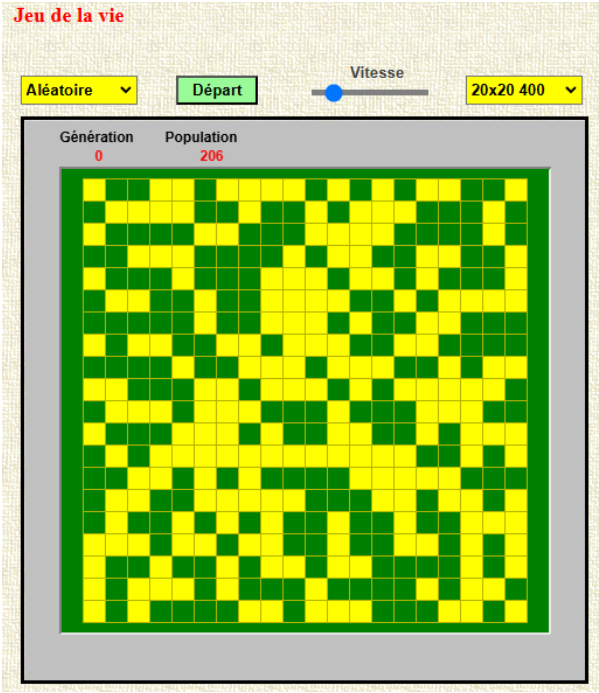

Figure 10 Simulateur Université Le Mans

et d'observer leurs comportements dynamiques.

4. Environnements de calcul : Mathematica et MATLAB

- Mathematica et MATLAB sont des plateformes puissantes qui permettent de programmer des automates cellulaires personnalisés et d'exploiter des outils de visualisation avancés (Research, 2020)
- Ces environnements offrent une grande flexibilité pour implémenter des règles spécifiques, telles que celles basées sur le compteur de parité de Fredkin.
- Toutefois, leur utilisation nécessite des compétences en programmation, ce qui peut représenter une barrière pour les utilisateurs recherchant une prise en main rapide.

```
In[1]:= funLife=FunctionCompile[
          Function[{Typed[board,TypeSpecifier["PackedArray"]["MachineInteger",2]]},
            Module[{dims,width,height,current,neighborCount,yIdx,xIdx},
            dims=Dimensions[board];
            width=dims[[2]];
            height=dims[[1]];
            Table[
              neighborCount=0;
              Do[
                yIdx=yy+yOffset;
                xIdx=xx+xOffset;
                If[yIdx>0&&yIdx≤height&&xIdx>0&&xIdx≤width,
                  neighborCount+=board[[yIdx,xIdx]]],
                {yOffset,-1,1},
                {xOffset,-1,1}];
              current=board[[yy,xx]];
              If[neighborCount===3||(current===1&&neighborCount===2),
                1,
                0],
              {yy,height},
              {xx,width}]]]]

Out[1]= CompiledCodeFunction[
          Code  Signature: {TypeSpecifier[PackedArray][Integer64, 2]} → TypeSpecifier[PackedArray][Integer64, 2]  ]
```

Créez un état initial.

```
In[2]:= board = RandomInteger[1, {100, 100}];
```

Utilisez Dynamic pour jouer au jeu en temps réel.

```
In[3]:= Dynamic[ArrayPlot[board = funLife[board]]]
```

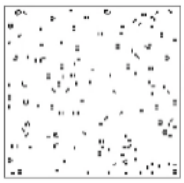

Figure 11 Mathematica & Matlab

Critère	Golly	FiatLux	Université Du Mans	Mathematica/MATLAB
Accessibilité & Interface	Facile d'utilisation, Open source	Interface conviviale et détaillée	Accessible en ligne, interface intuitive	Courbe d'apprentissage élevée, nécessite des compétences en programmation
Spécificité Fredkin	Configuration possible mais non native	Supporte Fredkin avec des modules spécifiques	Compatible avec diverses règles	Programmation personnalisée requise

Flexibilité	Simule plusieurs règles personnalisables	Outil avancé pour la recherche	Outil de démonstration rapide	Extrêmement flexible, adapté aux analyses complexes

(Bialas, 2012)

b) Analyse critique :

Les simulateurs existants présentent plusieurs avantages notables en matière de richesse fonctionnelle et d'ergonomie. Par exemple, des outils comme Golly et FiatLux offrent une interface conviviale qui permet une exploration rapide et intuitive des automates cellulaires (Gosper, 2005); (Fatès, 2010). Ces environnements permettent de visualiser en temps réel l'évolution des grilles et proposent de nombreuses options de paramétrage, ce qui facilite la compréhension des dynamiques émergentes.

De plus, l'utilisation d'environnements de calcul comme Mathematica ou MATLAB permet une flexibilité remarquable et une personnalisation poussée, offrant ainsi des capacités d'analyse quantitative approfondie ((Research, 2020) ; (The MathWorks, 2020). Leur puissance de calcul et leurs outils de visualisation avancés en font des solutions idéales pour les chercheurs souhaitant explorer des modèles complexes.

Limites des simulateurs existants **:**

Cependant, ces approches ne sont pas exemptes de limitations.

- La plupart de ces simulateurs sont conçus pour être polyvalents et ne se concentrent pas spécifiquement sur l'automate de Fredkin. Bien qu'ils offrent la possibilité de configurer des règles personnalisées, ils ne fournissent pas de modules optimisés pour étudier le compteur de parité caractéristique de Fredkin (Durand, 2018)
- Les environnements comme Mathematica et MATLAB, bien que puissants, nécessitent une expertise en programmation et une compréhension avancée des scripts, ce qui peut constituer un obstacle pour les utilisateurs cherchant une solution plus immédiate et ergonomique (Stephen Wolfram)

Positionnement du projet :

C’est dans ce contexte que notre projet se positionne avantageusement. En se concentrant exclusivement sur l’automate de Fredkin, notre interface graphique a été conçue pour répondre précisément aux besoins d’exploration de ce modèle particulier.

- Nous avons mis un accent particulier sur la fluidité de l’affichage et sur la synchronisation entre la carte Arduino et l'interface, afin d’assurer une visualisation en temps réel sans décalage, un aspect souvent négligé dans les solutions généralistes.
- De plus, notre système intègre un module d’enregistrement des logs robuste, permettant d’analyser et d’exporter facilement les données de simulation, ce qui facilite le diagnostic et l’optimisation du comportement de l’automate.

Ainsi, bien que les approches existantes offrent une grande polyvalence et une palette d’options riches, notre projet se distingue par sa spécialisation et sa simplicité d’utilisation pour l’étude de l’automate de Fredkin.

- Ce positionnement permet de combler certaines lacunes des outils généralistes tout en proposant une solution didactique et performante, adaptée aux contraintes d’une plateforme embarquée telle qu’Arduino.

c) Présentation du travail de nos predecesseurs

Le projet précèdent , réalisé par Aicha BELHADDAD et Katia BEN MANSOUR, porte sur l’implémentation d’automates cellulaires bidimensionnels de Fredkin (types 1 et 2) sur une plateforme Arduino Mega 2560 associée à un écran TFT tactile de 3,5 pouces piloté par un contrôleur ILI9486 utilisant une communication parallèle 8 bits. Leur réalisation intègre directement la gestion de l’affichage graphique et tactile sur la carte Arduino, permettant à l'utilisateur de sélectionner différents états initiaux (ligne, croix, motif en 'F' ou configuration aléatoire) et de visualiser l’évolution de l'automate étape par étape. De plus, elles ont mis en place un système d’enregistrement des résultats sous forme de logs sur une carte SD intégrée, exportables pour une analyse ultérieure.

Notre projet, tout en reprenant les concepts fondamentaux des automates cellulaires bidimensionnels de Fredkin (types 1 et 2), se distingue fortement par son approche méthodologique et technique. Nous avons développé une solution hybride combinant l'environnement Arduino avec une interface graphique externe réalisée en Python grâce à la bibliothèque Tkinter. Cette architecture apporte une interactivité améliorée, permettant à l'utilisateur une manipulation plus aisée et un contrôle précis du déroulement des simulations. De plus, contrairement au projet précédent, où l'intégralité du processus se déroule uniquement sur la carte Arduino, notre approche offre une modularité et une souplesse accrues grâce au traitement externe des données.

Cette externalisation permet notamment de visualiser les résultats en temps réel sous forme graphique à l'aide de la bibliothèque matplotlib, offrant ainsi des possibilités d'analyse et d'interprétation plus avancées. Tandis que les projets antérieurs génèrent des logs locaux sur une carte SD, notre système étend cette fonctionnalité en permettant un enregistrement flexible et plus détaillé des états de l'automate cellulaire, avec une exploitation directe et immédiate sous forme graphique depuis l'ordinateur hôte.

En résumé, si les réalisations précédentes privilégient une approche autonome centrée sur Arduino, notre projet met en avant la modularité, l'extensibilité et l'interaction avec une interface utilisateur plus riche, facilitant une meilleure analyse des données et apportant une souplesse supplémentaire quant à l'intégration future d'autres fonctionnalités.

VI. Réalisation du Projet

a) Cahier des charges et spécifications techniques :

1.Matériel utilisé et contraintes techniques :

Le projet repose sur deux composants matériels principaux :

- **Arduino Mega 2560**

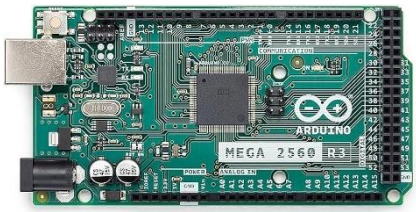

Figure 12 Arduino Mega 2560

Ce microcontrôleur, basé sur le processeur ATmega2560 fonctionnant à 16 MHz, offre une capacité de traitement suffisante pour gérer les mises à jour régulières d'un automate cellulaire. Il dispose d'une mémoire flash de 256 Ko et d'une SRAM limitée à 8 Ko, imposant ainsi une gestion rigoureuse des ressources pour stocker la grille, les états intermédiaires et les variables de contrôle. Le grand nombre de ports d'entrée/sortie disponibles facilite l'intégration de divers périphériques, notamment pour la communication avec l'écran TFT et, potentiellement, d'autres modules complémentaires.

- **Écran TFT 3,5 pouces**

Figure 13 Ecran TFT 3,5 pouces

Cet écran offre une résolution de 320 x 480 pixels, ce qui permet d'afficher une grille suffisamment détaillée pour visualiser l'évolution de l'automate.
Généralement piloté par des contrôleurs comme l'ILI9486, il peut gérer jusqu'à 65K couleurs, garantissant ainsi une distinction claire entre les différentes états (par exemple, une couleur pour les cellules actives et une autre pour les cellules inactives). La gestion du rafraîchissement de l'écran et la synchronisation avec l'envoi des données depuis l'Arduino sont essentielles pour éviter tout décalage, en particulier lors de simulations en temps réel.

Les contraintes techniques principales qui en découlent sont :
- **Mémoire limitée** : L'utilisation de 8 Ko de SRAM nécessite une optimisation du stockage des données et de l'affichage de la grille.
- **Vitesse de traitement** : Avec un processeur à 16 MHz, les algorithmes de mise à jour doivent être conçus de façon efficace pour assurer un cadencement fluide des générations de l'automate.
- **Résolution d'affichage** : La grille doit être dimensionnée de manière à tirer pleinement parti des 320 x 480 pixels disponibles sans compromettre la lisibilité, tout en respectant les contraintes de mémoire.

2.Exigences fonctionnelles de l'interface :

L'interface graphique du projet doit satisfaire plusieurs exigences fonctionnelles pour garantir une simulation fluide et interactive :

- **Affichage de l'automate**
  • La grille de l'automate doit être affichée de manière dynamique, en mettant à jour en temps réel les transitions d'état des cellules.
  • Des éléments visuels, tels que des schémas et des animations, doivent être utilisés pour illustrer les changements de la grille à chaque itération.

- **Sélection du mode Fredkin**
  • L’interface doit permettre de choisir entre les deux modes de l’automate de Fredkin (par exemple, Fredkin type 1 et Fredkin type 2) en fonction du voisinage (von Neumann ou Moore).
  • Cette sélection doit être intuitive, via des boutons ou un menu déroulant, pour faciliter la comparaison entre les deux configurations.
- **Enregistrement et suivi des logs**
  • Un système d’enregistrement automatique des logs doit être mis en place pour consigner les données de chaque génération (nombre de cellules actives/inactives, temps de calcul, etc.).

b) Choix des outils et bibliothèques :

Dans le cadre de ce projet, plusieurs bibliothèques ont été utilisées afin de répondre efficacement aux exigences de l'application développée. Du côté de Python, la bibliothèque `serial` a été employée pour gérer la communication série entre l'ordinateur et l'Arduino. Elle permet d'établir une connexion stable et fiable avec le matériel, essentielle à la transmission de données et à la synchronisation entre l'application logicielle et matérielle.

La bibliothèque graphique `tkinter` a également été employée pour concevoir l'interface utilisateur. Elle a été choisie pour sa simplicité d'utilisation et sa compatibilité native avec Python, facilitant ainsi la mise en place d'une interface ergonomique et intuitive sans nécessiter l'ajout de dépendances externes.

Les bibliothèques `datetime`, `time`, et `os` ont été incluses dans le projet pour gérer les opérations temporelles, assurer les délais nécessaires aux communications et gérer les opérations liées aux fichiers et répertoires.

Pour la partie Arduino, la bibliothèque standard `Serial` a été utilisée afin de gérer la communication série entre l'Arduino et l'ordinateur. Elle permet la réception et l'envoi efficaces des données nécessaires au fonctionnement de l'automate Fredkin, assurant ainsi une synchronisation parfaite avec l'interface utilisateur développée en Python. De plus, les bibliothèques `MCUFRIEND_kbv`, `Adafruit_GFX`, et `Adafruit_BusIO` ont été utilisées pour gérer l'affichage sur un écran TFT et pour faciliter la communication des données à travers divers protocoles de communication, comme l'I2C ou le SPI. Ces bibliothèques permettent d'afficher visuellement la grille de l'automate cellulaire, facilitant ainsi l'interprétation rapide des états des cellules par l'utilisateur.

Les bibliothèques sélectionnées, tant côté Python que côté Arduino, assurent une interaction fluide, une interface conviviale et une gestion optimale des communications et des données au sein de l'application.

c) Conception de l'interface et architecture logicielle :

Pour assurer une compréhension claire du fonctionnement global du système, nous avons élaboré plusieurs schémas et diagrammes explicatifs – organigrammes, flux de données et diagrammes de séquence – qui illustrent les différentes étapes du processus, depuis l'envoi des données par la carte Arduino jusqu'à leur affichage sur l'interface graphique, en passant par la synchronisation et la gestion du fichier log.

1. Schémas et diagrammes explicatifs

- *Organigramme général du système :*

  Ce diagramme présente les grandes étapes du fonctionnement, depuis la collecte des données sur Arduino, le traitement des états de l'automate, l'envoi des mises à jour via le port série, jusqu'à la réception des données par l'interface graphique sur PC. Il met en évidence les différents modules logiciels (gestion de l'affichage, communication série, et enregistrement des logs) et leur interaction.

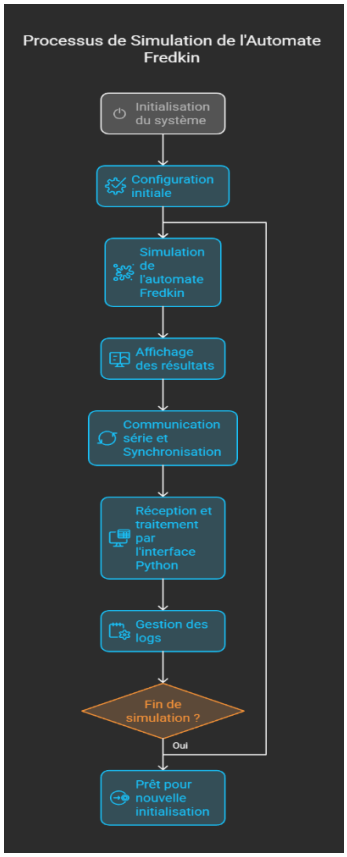

Figure 14 processus de simulation de l'automate de fredkin

- *Flux de données et diagramme de séquence :*

  Un diagramme de séquence détaille le processus d'échange des informations. Au démarrage, l'Arduino initialise la grille et envoie une trame de données décrivant l'état initial. Ensuite, pour chaque génération, l'Arduino calcule le nouvel état de la grille selon les règles de transition (notamment la règle de parité pour Fredkin) et

envoie les données mises à jour via le port série. L'interface graphique, recevant ces données, les interprète et rafraîchit l'affichage en conséquence, tout en enregistrant chaque trame dans un fichier log. Ce flux synchronisé est crucial pour garantir que l'évolution de la simulation se déroule de manière fluide et sans décalage perceptible.

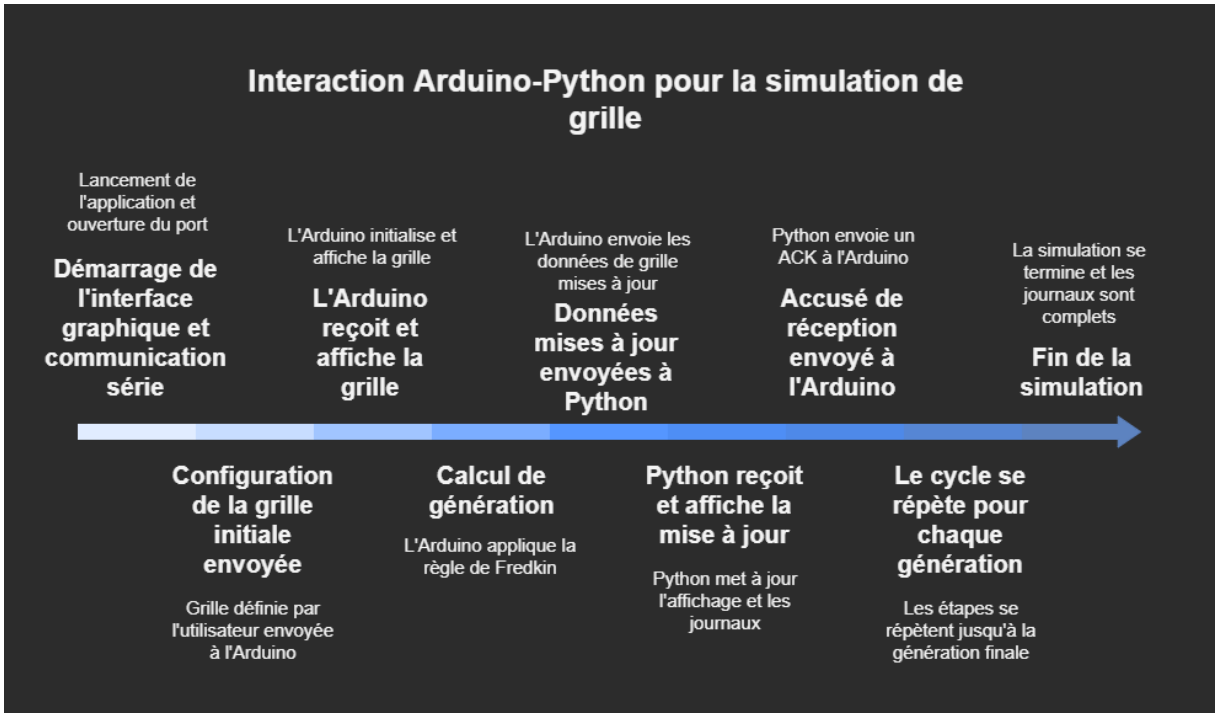

Figure 15 interaction arduino-python

- *Diagramme de synchronisation et gestion du log :*

  Un schéma spécifique illustre le mécanisme de synchronisation entre l'Arduino et l'interface. À chaque envoi, une horodatation est ajoutée aux données par Arduino, et l'interface, après réception, écrit ces informations dans le fichier log. Ce processus permet de conserver une trace détaillée de chaque itération, facilitant ainsi l'analyse post-simulation et le diagnostic de toute anomalie.

2. Description du processus d'envoi et de réception des données

- *Envoi des données :*

  L'Arduino Mega 2560, programmé pour exécuter les règles de l'automate cellulaire de Fredkin, calcule la nouvelle configuration de la grille à chaque génération. Une fois le calcul terminé, il encapsule les données (par exemple, le numéro de génération, le nombre de cellules actives et inactives, et la configuration de la grille) dans une trame formatée et l'envoie via le port série en utilisant la bibliothèque Arduino appropriée.
- *Réception et affichage sur l'interface :*

  L'interface graphique, développée à l'aide de bibliothèques comme Tkinter en combinaison avec PySerial, se connecte au port série de l'ordinateur. Dès qu'une

trame est reçue, l'interface interprète les données, met à jour l'affichage de la grille sur l'écran et rafraîchit l'état de la simulation. Ce rafraîchissement en temps réel permet à l'utilisateur de visualiser l'évolution des états de chaque cellule avec fluidité.

- *Synchronisation et gestion des logs :*

  Pour assurer une cohérence entre la simulation et son affichage, un mécanisme de synchronisation est mis en place. Chaque trame de données envoyée par l'Arduino est accompagnée d'un horodatage, ce qui permet à l'interface de s'assurer que les informations sont traitées dans l'ordre chronologique. Par ailleurs, chaque trame reçue est immédiatement enregistrée dans un fichier log, ce qui permet d'avoir une archive complète des différentes générations. Ce fichier log est structuré pour faciliter l'analyse ultérieure, notamment en cas de besoin de débogage ou d'optimisation des performances.

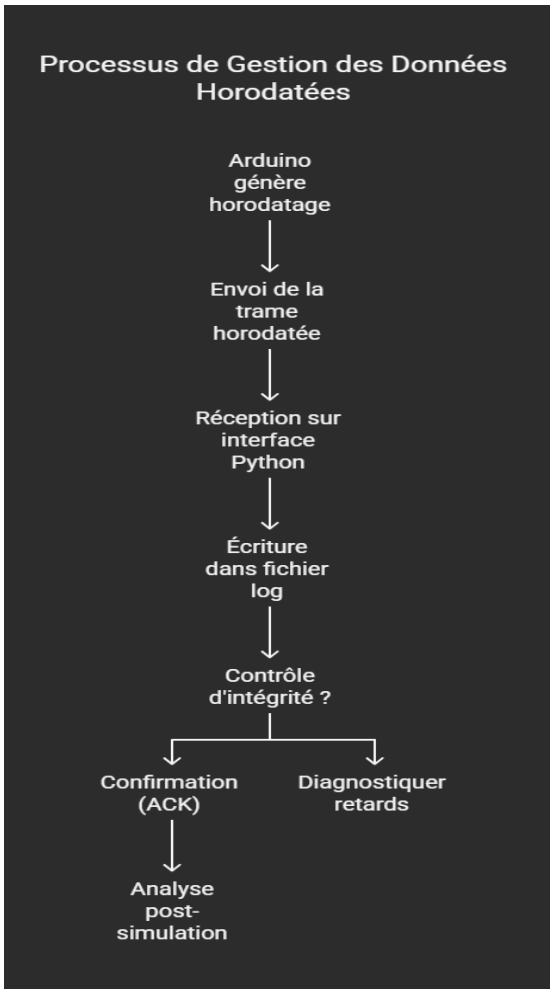

Figure 16 processus de gestion de données horodatées

d) Problématiques techniques et solutions :

Au cours de la mise en œuvre du projet, plusieurs défis techniques se sont présentés, nécessitant l'adoption de solutions adaptées pour assurer la robustesse et la fluidité de la simulation.

1. Synchronisation des données

Difficulté :

- La transmission des mises à jour de la grille depuis l'Arduino vers l'interface graphique devait être réalisée en temps réel, sans introduire de décalage perceptible pour l'utilisateur.
- La synchronisation entre l'envoi des trames de données et leur réception s'est avérée délicate, notamment en raison des délais inhérents à la communication série.

Solutions adoptées :

- Délais de stabilisation **:** Un délai de quelques secondes a été intégré lors de l'établissement de la connexion série pour s'assurer que la communication est stable avant de démarrer la simulation.
- Horodatage et contrôle d'ordre : Chaque trame envoyée par l'Arduino est accompagnée d'un horodatage, ce qui permet à l'interface de traiter les données dans l'ordre chronologique et de détecter d'éventuelles anomalies dans le flux.
- Optimisation du flux de données : Des algorithmes légers et efficaces ont été mis en place pour réduire la taille des trames, minimisant ainsi le risque de saturation du port série.

2. Limitations du matériel

Difficulté :

- La carte Arduino Mega 2560, bien que puissante, dispose d'une mémoire SRAM limitée (8 Ko) et d'un processeur fonctionnant à 16 MHz, imposant des contraintes sur la complexité des algorithmes et la gestion de la grille.
- L'écran TFT de 3,5 pouces, avec une résolution de 320 x 480 pixels, requiert une gestion fine de l'affichage pour éviter les ralentissements lors du rafraîchissement de la grille.

Solutions adoptées :

- **Optimisation de la mémoire :** Le code a été écrit de manière à minimiser l'utilisation de la mémoire, en adoptant des structures de données légères pour représenter la grille et en limitant le stockage temporaire des informations.
- **Algorithmes efficaces :** Les algorithmes de mise à jour de la grille ont été optimisés pour réduire le temps de calcul, en limitant le nombre d'opérations par cellule et en utilisant des boucles imbriquées de manière judicieuse.

- **Adaptation de la résolution :** La taille des cellules affichées a été choisie pour équilibrer la visibilité et la charge de traitement, en s'assurant de ne pas dépasser la capacité d'affichage de l'écran tout en gardant une bonne lisibilité.

3. Gestion du fichier log

Difficulté :

- La conservation d'une trace détaillée de chaque génération de la simulation était essentielle pour l'analyse ultérieure et le débogage, mais l'enregistrement fréquent dans un fichier log pouvait impacter la performance globale de la simulation.

Solutions adoptées :

- **Bufférisassions des logs :** Plutôt que d'écrire immédiatement chaque trame dans le fichier log, un système de bufférisassions a été mis en place pour regrouper plusieurs entrées avant de les enregistrer, réduisant ainsi les opérations d'écriture fréquentes.
- **Formatage efficace :** Le format des logs a été standardisé avec un horodatage et des informations clés (numéro de génération, nombre de cellules actives/inactives, etc.), facilitant ainsi leur lecture et leur analyse sans alourdir inutilement le fichier.
- **Gestion asynchrone :** La gestion des logs a été séparée du processus de mise à jour de l'affichage pour éviter tout blocage de l'interface, permettant ainsi d'assurer une simulation fluide même lors de l'enregistrement des données.

VII. Difficultés Rencontrées et Leçons Tirées

a) Retour d'expérience :

Au cours de ce projet, nous avons été confrontées à un certain nombre de défis tant sur le plan technique qu'organisationnel.

Problèmes techniques :

- **Synchronisation des données et latence :**
  Nous avons constaté des retards dans la transmission des informations entre l'Arduino et l'interface graphique. Ces délais pouvaient entraîner des décalages dans l'affichage de l'évolution de la grille, affectant ainsi la fluidité de la simulation.
- **Contraintes de mémoire et de performance :**
  La mémoire limitée de l'Arduino Mega (8 Ko de SRAM) a représenté un obstacle majeur, notamment lors du stockage des états de la grille et de l'enregistrement en temps réel des logs. Par ailleurs, le processeur fonctionnant à 16 MHz nécessitait l'optimisation des algorithmes pour garantir des mises à jour rapides et cohérentes.

- **Gestion du fichier log :**
  L'enregistrement fréquent des données dans le log, indispensable pour le suivi et l'analyse post-simulation, ralentissait parfois le système et perturbait la synchronisation globale de la simulation.

Problèmes organisationnels :

- **Planification et coordination de l'équipe :**
  La répartition des tâches et la gestion du temps se sont avérées complexes, en particulier lorsqu'il s'agissait de synchroniser les travaux de développement de l'interface graphique avec ceux liés au traitement des données sur Arduino.
- **Communication et intégration des retours :**
  Le besoin d'un retour d'expérience régulier a mis en lumière des ajustements à apporter au processus de développement, ce qui a parfois nécessité de revoir rapidement nos priorités et notre méthode de travail.

Méthodes d'identification et résolution :

- **Tests itératifs et débogage :**
  Chaque module a été soumis à des tests approfondis dès sa conception, permettant d'identifier rapidement les problèmes de synchronisation, de performance ou de gestion de la mémoire. L'analyse des logs et l'observation directe du comportement de la simulation ont joué un rôle crucial dans la détection des anomalies.
- **Optimisation des algorithmes et bufferisation :**
  Pour pallier les limitations de la mémoire et réduire la latence, nous avons réécrit certains algorithmes afin de les rendre plus efficaces et avons introduit une bufferisation des données avant leur enregistrement dans le log.
- **Révision de l'organisation interne :**
  Pour améliorer la coordination, nous avons adopté une approche agile, organisant des réunions régulières pour échanger sur l'état d'avancement, redistribuer les tâches en fonction des urgences et intégrer les retours des membres de l'équipe de manière constructive.

b) Analyse des choix techniques :

Les solutions que nous avons mises en œuvre ont eu un impact significatif sur les performances et la robustesse de l'interface. Par exemple, l'optimisation des algorithmes de mise à jour de la grille et la bufferisation des données pour la gestion des logs ont permis de réduire les délais de transmission et d'éviter des saturations mémoire. Grâce à ces optimisations, l'interface reste réactive même lors de mises à jour fréquentes, assurant une visualisation en temps réel fluide de l'évolution de l'automate. La synchronisation via l'horodatage des trames envoyées par l'Arduino a également renforcé la cohérence du système, permettant de traiter les informations dans l'ordre correct et de minimiser les risques de décalage.

Cependant, certaines limites subsistent. Par exemple, la contrainte de mémoire sur l'Arduino Mega 2560 reste un défi pour des simulations de plus grande envergure ou pour des configurations plus complexes. De même, l'utilisation de Tkinter, bien qu'accessible et simple à implémenter, peut s'avérer moins performante pour des interfaces nécessitant une gestion graphique très intensive.

Pistes d'amélioration et alternatives pour des projets futurs :

- *Optimisation matérielle :*
  - Passer à des microcontrôleurs avec une mémoire plus conséquente et une vitesse de traitement supérieure pourrait permettre de réaliser des simulations plus complexes sans compromettre la fluidité.
  - Envisager l'utilisation d'écrans à résolution plus élevée pour offrir une visualisation encore plus détaillée de la grille.
- *Amélioration de l'interface graphique :*
  - Migrer vers des bibliothèques graphiques plus avancées ou des Framework modernes (comme PyQt ou Kivy) pourrait améliorer l'esthétique et la réactivité de l'interface.
  - Intégrer des animations plus sophistiquées et des options de visualisation interactives (zoom, panoramique, etc.) permettrait une exploration plus approfondie des dynamiques de l'automate.
- *Gestion de la communication série :*
  - Explorer des protocoles de communication plus rapides ou optimisés pour les environnements embarqués pourrait réduire davantage la latence.
  - Utiliser des microcontrôleurs dotés de capacités de communication sans fil (comme le Wi-Fi ou le Bluetooth) pourrait offrir plus de flexibilité pour des applications distribuées.
- *Approche logicielle :*
  - Développer des modules logiciels modulaires et évolutifs faciliterait la maintenance et l'extension des fonctionnalités de l'interface.
  - L'intégration d'outils d'analyse de données en temps réel, tels que des graphiques interactifs ou des tableaux de bord, pourrait enrichir l'expérience utilisateur et offrir des pistes supplémentaires pour l'analyse post-simulation.

VIII. Exemple de mise en marche

a) Organisation des fichier:

Créez un dossier (par exemple, "MonProjet") et placez-y les fichiers : interface.py, plot.py, logs_fredkin.txt (pour les journaux) et votre code Arduino fredkin.ino dans un repertoire du meme nom.

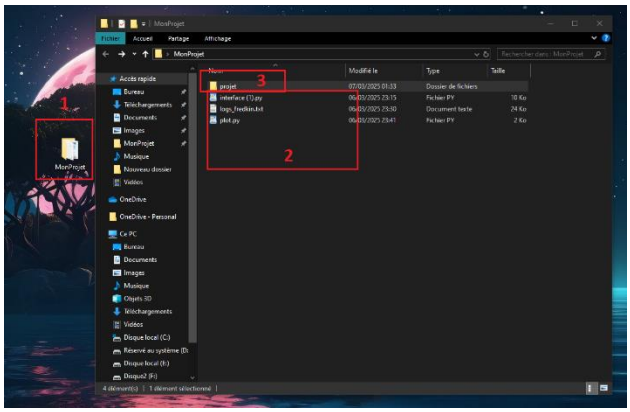

Figure 18 organisation fichier 1

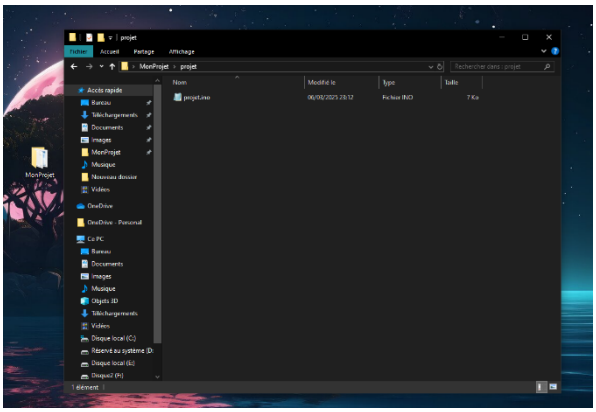

Figure 17 organisation fichier 2

b) Librairies Arduino :

Installez « MCUFRIEND_kbv » (pour l'écran TFT) et « Adafruit_GFX » (pour l'affichage des formes et du texte). Compilez et téléversez afin de vérifier que les bibliothèques sont correctement installées.

c) Dépendances Python :

Dans un terminal ou une invite de commandes, placez-vous dans le dossier « MonProjet » et exécutez :

pip install pyserial

pip install matplotlib

pour installer pyserial (communication série) et matplotlib (tracé de données).

d) Configuration de l'interface :

Dans interface.py, adaptez la ligne SERIAL_PORT à votre port série (par exemple « COM3 » sous Windows ou « /dev/ttyACM0 » sous Linux

```
import serial
import tkinter as tk
from tkinter import messagebox, filedialog
import datetime
import time
import os

# Configuration du port sÃ©rie
SERIAL_PORT = "COM3"  # Ã€ adapter Ã  ta machine
BAUD_RATE = 115200
TIMEOUT = 1  # Timeout en secondes

class FredkinInterface:
    def __init__(self):
        self.WIDTH, self.HEIGHT = 48, 32
        self.CELL_SIZE = 10
        self.serial = None
        self.running = False
        self.log_file = None
```

A ADAPTER SELON LA MACHINE

Figure 19 configuration port serie

Une fenêtre Tkinter s'ouvre, affichant la grille 48×32. Vous pouvez cliquer pour changer l'état de chaque cellule, choisir ou laisser le fichier logs_fredkin.txt, puis cliquer sur « Envoyer Grille » pour transmettre le motif à l'Arduino.

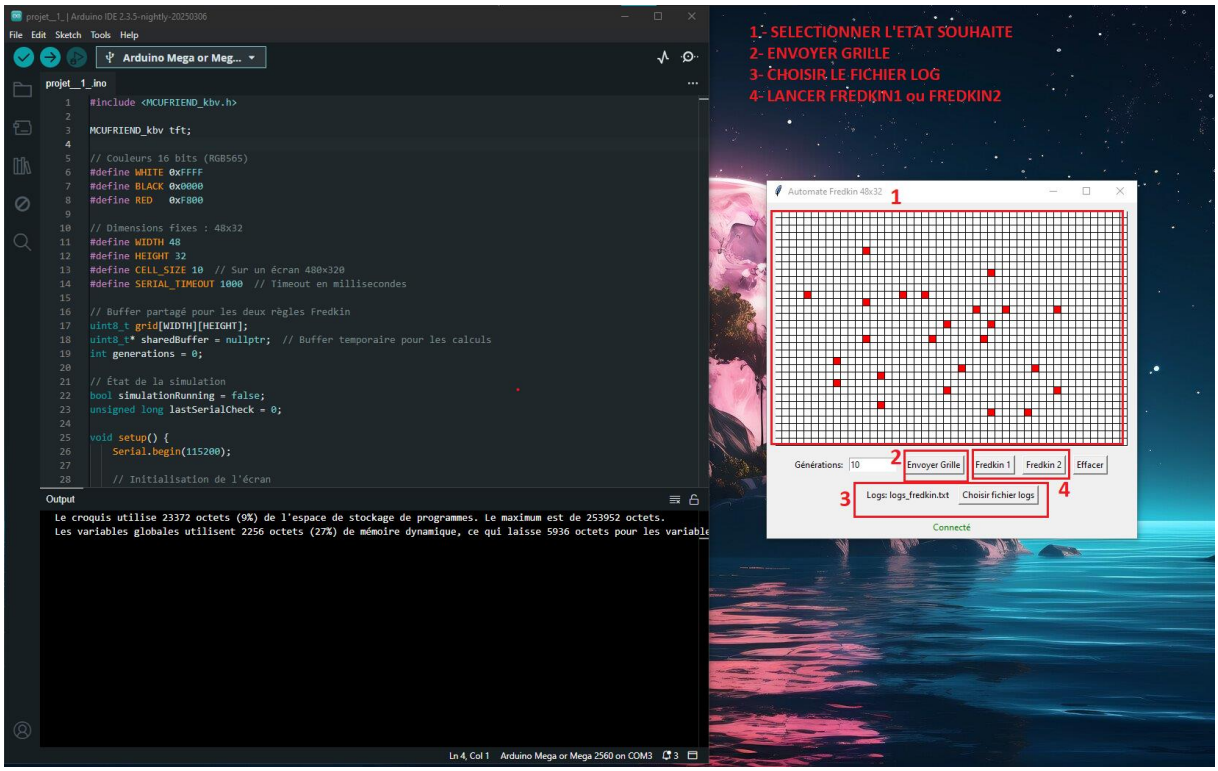

Figure 20 fenetre tkinter

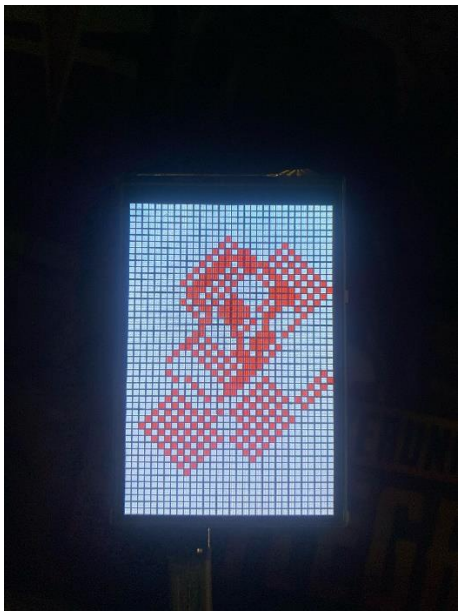

Figure 21 affichage écran

e) Exécution et visualisation :

Le code Arduino calcule les générations et inscrit les informations correspondantes dans logs_fredkin.txt.

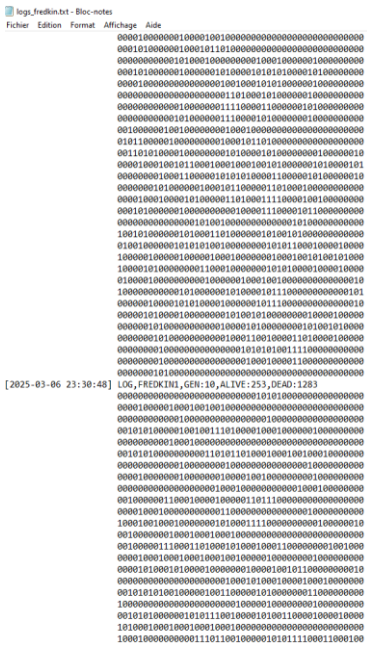

Figure 22 fichier log

Pour visualiser l'évolution du nombre de cellules vivantes et mortes, exécutez plot.py

Le script lit logs_fredkin.txt et trace deux courbes (Alive et Dead) en fonction des générations.

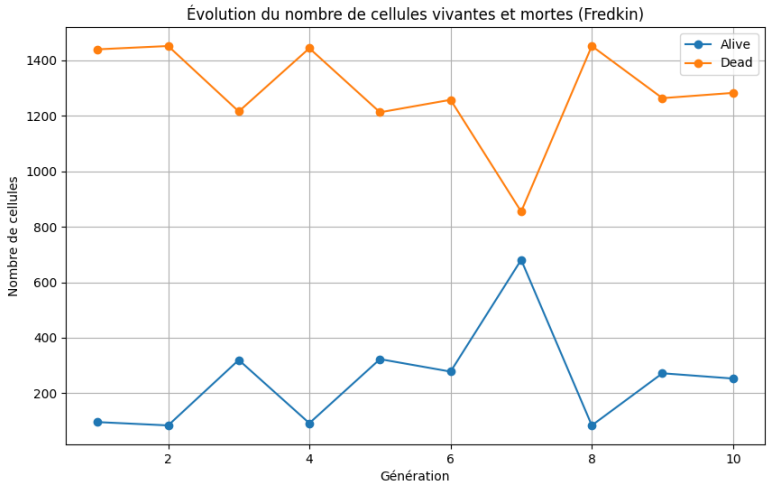

Figure 23 graphique matplotlib de l'évolution de la population

IX. Conclusion et Perspectives

a) Bilan du projet :

Au terme de ce projet, nous pouvons affirmer que nos objectifs ont été largement atteints. Nous avons réussi à concevoir et à développer une interface graphique interactive permettant de simuler en temps réel l'automate cellulaire de Fredkin sur une plateforme Arduino. Grâce à une approche rigoureuse mêlant théorie et expérimentation, nous avons implémenté une solution qui non seulement offre une visualisation fluide de l'évolution de la grille, mais intègre également un système de gestion de la communication série et d'enregistrement des logs performant.

Les résultats obtenus confirment que notre interface permet de saisir efficacement les transitions d'état de l'automate grâce à des mises à jour synchronisées et un affichage optimisé. De plus, le système de logs mis en place facilite l'analyse des différentes générations et offre un outil précieux pour le débogage et l'optimisation des performances.

L'originalité de notre travail réside dans la spécialisation sur l'automate de Fredkin, en se concentrant sur sa règle de transition basée sur le compteur de parité. Cette focalisation nous a permis de développer une solution sur mesure, adaptée aux contraintes d'un microcontrôleur Arduino, et de proposer des optimisations spécifiques pour gérer les limitations en mémoire et en vitesse de traitement. En outre, l'intégration d'éléments interactifs et la possibilité de paramétrer la simulation en fonction des besoins de l'utilisateur témoignent de l'aspect pédagogique et innovant de notre approche.

b) Perspectives d'évolution :

Le projet ouvre la voie à de nombreuses améliorations et extensions qui pourraient enrichir la simulation de l'automate cellulaire de Fredkin et étendre son champ d'application :

1. Optimisation de l'interface et amélioration de l'expérience utilisateur

- *Interface graphique avancée :*

Envisager la migration vers des Framework plus modernes tels que PyQt ou Kivy pourrait offrir une interface plus esthétique et réactive, intégrant des animations plus sophistiquées et des options interactives telles que le zoom, le panoramique ou la modification en temps réel des paramètres de simulation.

- *Intégration de fonctionnalités interactives supplémentaires :*

Proposer des outils permettant à l'utilisateur de personnaliser davantage l'apparence de la grille ou d'ajuster dynamiquement les règles de transition (par exemple, via un éditeur visuel de règles) pourrait enrichir l'expérience pédagogique et pratique.

2. Extension des fonctionnalités

- *Nouveaux modes de simulation :*

Au-delà du mode Fredkin classique, l'ajout de nouveaux modes, tels que des automates hybrides permettrait d'explorer des dynamiques plus complexes et de comparer l'influence de différents types de voisinages ou règles.

- *Analyse et visualisation des données :*

L'intégration d'outils d'analyse en temps réel (tableaux de bord interactifs, graphiques évolutifs, statistiques détaillées) permettrait de mieux comprendre les comportements émergents et de quantifier les performances du système.

3. Perspectives matérielles et applications futures

- *Optimisation matérielle :*

Passer à des microcontrôleurs offrant une mémoire et une vitesse de traitement supérieures pourrait permettre de simuler des grilles plus grandes et d'introduire des calculs plus complexes, tout en maintenant une interface graphique fluide.

- *Applications pratiques :*

Les techniques développées pourraient être appliquées à la modélisation de phénomènes réels dans divers domaines, tels que la biologie (modélisation de la croissance cellulaire ou des dynamiques épidémiques), la physique (simulation de phénomènes de diffusion ou d'auto-organisation) ou même la cryptographie (génération de séquences pseudo-aléatoires robustes).

X. Travaux cités

Bialas, P. (2012). *Comparatif des outils de simulation d'automates cellulaires. Revue d'Informatique Appliquée.*

Conway, J. H. (1970). The Game of Life. *Scientific American*, 120-123.

Cook, M. (2004). *Universality in Elementary Cellular Automata.* Complex Systems.

Droz, B. C. (1998). *Cellular Automata Modeling of Physical Systems.* Cambridge University Press.

Durand, L. (2018). *Limites et potentialités des simulateurs d'automates cellulaires. Journal des Systèmes Dynamiques.*

Elwyn R. Berlekamp, J. H. (1982). *Winning Ways for Your Mathematical Plays, Volume 2.* Academic Press.

Fatès, N. (2010). *FiatLux: Outil de simulation pour automates cellulaires.* Nancy: LORIA Nancy.

Fredkin, E. (1982). "Conservative Logic. *International Journal of Theoretical Physics*, pp. 219-153.

Gardner, M. (1970). Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Scientific American*, p. 223.

Gosper, R. W. (2005). *Golly: An Open-Source Cellular Automata Simulator.* Récupéré sur https://golly.sourceforge.net

Ilachinski, A. (2001). *Cellular Automata: A Discrete Universe.* World Scientific.

Langton, C. G. (1986). Studying Artificial Life with Cellular Automata. *Physica D: Nonlinear Phenomena*, pp. 120-149.

Macia, A. A. (2005). *Unconventional Computing: From Cellular Automata to Wetware.* Springer.

Mans, U. d. (2015). *Simulateur d'automates cellulaires en ligne.* Le Mans.

Margolus, N. (1984). Physics-Like Models of Computation. *Physica D: Nonlinear Phenomena*, p. 210.

Margolus, T. T. (1987). *Cellular Automata Machines: A New Environment for Modeling .* Cambridge, Massachusetts: MIT Press.

Mitchelle. (2009). *Complexity: A Guided Tour.* Oxford University Press.

Morita, Y. (1998). Reversible Computing with Cellular Automata. *Journal of Computer Science and Technology*, p. 45.

Neumann, J. v. (1966). *Theory of Self-Reproducing Automata.* Urbana, Illinois: University of Illinois Press.

Research, W. (2020). *Mathematica Documentation.*

Sipper, M. (1997). *Evolution of Parallel Cellular Machines: The Cellular Programming Approach.*

Stephen Wolfram. (s.d.). *A New Kind of Science.* Wolfram Media.

The MathWorks, I. (2020). *MATLAB Documentation.*

Ulam, S. (1952). *Random Processes and Transformations.* Los Alamos Scientific Laboratory Report.

XI. Annexes

a) Code python :

```
import serial
import tkinter as tk
from tkinter import messagebox, filedialog
import datetime
import time
import os

# Configuration du port serie
SERIAL_PORT = "COM3"  # A adapter a ta machine
BAUD_RATE = 115200
TIMEOUT = 1  # Timeout en secondes


class FredkinInterface:
    def __init__(self):
        self.WIDTH, self.HEIGHT = 48, 32
        self.CELL_SIZE = 10
        self.serial = None
        self.running = False
        self.log_file = None

        # Grille en memoire Python (0 = mort/blanc, 1 = vivant/rouge)
        self.grille = [[0] * self.HEIGHT for _ in range(self.WIDTH)]

        self.setup_gui()
        self.connect_serial()

    def connect_serial(self):
        """Etablit la connexion serie avec gestion d'erreur."""
        try:
            self.serial = serial.Serial(SERIAL_PORT, BAUD_RATE,
timeout=TIMEOUT)
            time.sleep(2)  # Attente de la connexion serie
            self.status_label.config(text="Connecte", fg="green")
        except serial.SerialException as e:
            self.status_label.config(text="Non connecte", fg="red")
            messagebox.showerror(
                "Erreur", f"Impossible d'ouvrir le port {SERIAL_PORT}. {e}"
            )
            self.serial = None

    def setup_gui(self):
        """Initialise l'interface graphique."""
        self.root = tk.Tk()
```

```
        self.root.title("Automate Fredkin 48x32")

        # Frame principal
        main_frame = tk.Frame(self.root)
        main_frame.pack(padx=10, pady=10)

        # Canvas pour dessiner la grille
        self.canvas = tk.Canvas(
            main_frame,
            width=self.WIDTH * self.CELL_SIZE,
            height=self.HEIGHT * self.CELL_SIZE,
            bg="white",
        )
        self.canvas.pack()

        # Frame pour les controles
        controls_frame = tk.Frame(main_frame)
        controls_frame.pack(pady=10)

        # Frame pour le fichier de logs
        log_frame = tk.Frame(main_frame)
        log_frame.pack(pady=5)

        # Label et bouton pour le fichier de logs
        self.log_label = tk.Label(
            log_frame, text="Aucun fichier de logs selectionne",
wraplength=400
        )
        self.log_label.pack(side=tk.LEFT, padx=5)
        tk.Button(
            log_frame, text="Choisir fichier logs",
command=self.select_log_file
        ).pack(side=tk.LEFT, padx=5)

        # Etiquette pour le nombre de generations
        tk.Label(controls_frame, text="Generations:").pack(side=tk.LEFT)

        # Champ pour choisir le nombre de generations
        self.entry_generations = tk.Entry(controls_frame, width=10)
        self.entry_generations.insert(0, "10")
        self.entry_generations.pack(side=tk.LEFT, padx=5)

        # Boutons de controle
        tk.Button(
            controls_frame, text="Envoyer Grille", command=self.envoyer_grille
        ).pack(side=tk.LEFT, padx=5)
        tk.Button(
            controls_frame, text="Fredkin 1", command=lambda:
self.set_mode("FREDKIN1")
```

```
        ).pack(side=tk.LEFT, padx=5)
        tk.Button(
            controls_frame, text="Fredkin 2", command=lambda:
self.set_mode("FREDKIN2")
        ).pack(side=tk.LEFT, padx=5)
        tk.Button(controls_frame, text="Effacer",
command=self.clear_grid).pack(
            side=tk.LEFT, padx=5
        )

        # Status bar
        self.status_label = tk.Label(self.root, text="Non connecte", fg="red")
        self.status_label.pack(pady=5)

        # Binding des evenements
        self.canvas.bind("<Button-1>", self.clic_souris)
        self.root.protocol("WM_DELETE_WINDOW", self.cleanup_and_exit)

        self.dessiner_grille()

    def select_log_file(self):
        """Permet a l'utilisateur de choisir l'emplacement du fichier de
logs."""
        initial_dir = os.path.dirname(self.log_file) if self.log_file else
os.getcwd()
        filename = filedialog.asksaveasfilename(
            initialdir=initial_dir,
            title="Choisir le fichier de logs",
            defaultextension=".txt",
            filetypes=[("Fichiers texte", "*.txt"), ("Tous les fichiers",
"*.*")],
            initialfile="logs_fredkin.txt",
        )
        if filename:
            self.log_file = filename
            self.log_label.config(text=f"Logs: {os.path.basename(filename)}")
            # Creer le fichier s'il n'existe pas
            with open(self.log_file, "a") as f:
                f.write(f"[{datetime.datetime.now()}] Fichier de logs
cree/ouvert\n")

    def write_log(self, message, with_timestamp=True):
        """Ecrit un message dans le fichier de logs."""
        if not self.log_file:
            return

        try:
            with open(self.log_file, "a") as f:
                if with_timestamp:
```

```
                    now = datetime.datetime.now().strftime("%Y-%m-%d
%H:%M:%S")
                    f.write(f"[{now}] {message}\n")
                else:
                    # Pour les lignes de grille, pas d'horodatage
                    prefix_len = len("YYYY-MM-DD HH:MM:SS") + 3
                    spaces = " " * prefix_len
                    f.write(f"{spaces}{message}\n")
        except IOError as e:
            messagebox.showerror(
                "Erreur", f"Impossible d'ecrire dans le fichier de logs: {e}"
            )

    def clear_grid(self):
        """Efface toute la grille."""
        self.grille = [[0] * self.HEIGHT for _ in range(self.WIDTH)]
        self.dessiner_grille()

    def dessiner_grille(self):
        """Dessine la grille sur le Canvas Tkinter."""
        self.canvas.delete("all")
        for y in range(self.HEIGHT):
            for x in range(self.WIDTH):
                color = "red" if self.grille[x][y] else "white"
                self.canvas.create_rectangle(
                    x * self.CELL_SIZE,
                    y * self.CELL_SIZE,
                    (x + 1) * self.CELL_SIZE,
                    (y + 1) * self.CELL_SIZE,
                    fill=color,
                    outline="black",
                )

    def clic_souris(self, event):
        """Gere les clics de souris sur la grille."""
        cx = event.x // self.CELL_SIZE
        cy = event.y // self.CELL_SIZE
        if 0 <= cx < self.WIDTH and 0 <= cy < self.HEIGHT:
            self.grille[cx][cy] = 1 - self.grille[cx][cy]
            self.dessiner_grille()

    def envoyer_grille(self):
        """Envoie la grille a l'Arduino avec gestion d'erreur."""
        if not self.log_file:
            if not messagebox.askyesno(
                "Attention",
                "Aucun fichier de logs selectionne. Voulez-vous en choisir un
maintenant?",
            ):
```

```
                return
            self.select_log_file()
            if not self.log_file:
                return

        if not self.serial or not self.serial.is_open:
            messagebox.showerror("Erreur", "Port serie non disponible")
            return

        try:
            generations = int(self.entry_generations.get())
            if generations <= 0:
                raise ValueError("Le nombre de generations doit etre >= 1")

            # Reinitialisation du fichier de logs
            self.write_log("Debut nouvelle simulation")

            # Conversion de la grille en chaine binaire
            grille_str = "".join(
                str(self.grille[x][y])
                for y in range(self.HEIGHT)
                for x in range(self.WIDTH)
            )

            # Envoi des donnees
            data = f"GEN:{generations}:{grille_str}\n"
            self.serial.write(data.encode())
            self.serial.flush()

        except ValueError as e:
            messagebox.showerror("Erreur", str(e))
        except serial.SerialException as e:
            messagebox.showerror("Erreur", f"Erreur de communication serie:
{e}")
            self.status_label.config(text="Deconnecte", fg="red")

    def set_mode(self, mode):
        """Configure le mode de simulation avec gestion d'erreur."""
        if not self.serial or not self.serial.is_open:
            messagebox.showerror("Erreur", "Port serie non disponible")
            return

        try:
            cmd = f"MODE:{mode}\n"
            self.serial.write(cmd.encode())
            self.serial.flush()
        except serial.SerialException as e:
            messagebox.showerror("Erreur", f"Erreur de communication serie:
{e}")
```

```
            self.status_label.config(text="Deconnecte", fg="red")

    def lire_logs(self):
        """Lit les donnees de l'Arduino avec timeout."""
        if not self.serial or not self.serial.is_open:
            self.root.after(100, self.lire_logs)
            return

        try:
            if self.serial.in_waiting:
                ligne_complete =
self.serial.readline().decode().rstrip("\r\n")
                if ligne_complete:
                    if ligne_complete.startswith("LOG,"):
                        self.write_log(ligne_complete, with_timestamp=True)
                    else:
                        self.write_log(ligne_complete, with_timestamp=False)

                    self.serial.write(b"OK\n")
                    self.serial.flush()

        except (serial.SerialException, IOError) as e:
            print(f"Erreur de communication: {e}")
            self.status_label.config(text="Deconnecte", fg="red")

        self.root.after(100, self.lire_logs)

    def cleanup_and_exit(self):
        """Nettoie les ressources avant de quitter."""
        if self.serial and self.serial.is_open:
            self.serial.close()
        self.root.destroy()

    def run(self):
        """Lance l'interface."""
        self.lire_logs()
        self.root.mainloop()


if __name__ == "__main__":
    app = FredkinInterface()
    app.run()
```

b) code arduino :

```
#include <MCUFRIEND_kbv.h>

MCUFRIEND_kbv tft;
```

```
// Couleurs 16 bits (RGB565)
#define WHITE 0xFFFF
#define BLACK 0x0000
#define RED   0xF800

// Dimensions fixes : 48x32
#define WIDTH 48
#define HEIGHT 32
#define CELL_SIZE 10  // Sur un écran 480×320
#define SERIAL_TIMEOUT 1000  // Timeout en millisecondes

// Buffer partagé pour les deux règles Fredkin
uint8_t grid[WIDTH][HEIGHT];
uint8_t* sharedBuffer = nullptr;  // Buffer temporaire pour les calculs
int generations = 0;

// État de la simulation
bool simulationRunning = false;
unsigned long lastSerialCheck = 0;

void setup() {
    Serial.begin(115200);

    // Initialisation de l'écran
    uint16_t ID = tft.readID();
    if (ID == 0xD3D3) ID = 0x9486; // Ajustement pour certains écrans
    tft.begin(ID);
    tft.setRotation(1);   // orientation paysage
    tft.fillScreen(WHITE);

    // Allocation du buffer partagé
    sharedBuffer = new uint8_t[WIDTH * HEIGHT];

    // Message de démarrage
    Serial.println("LOG,System started");
}

void loop() {
    // Vérifie les commandes série avec timeout
    if (Serial.available()) {
        String command = Serial.readStringUntil('\n');
        processCommand(command);
    }
}

void processCommand(const String &command) {
    if (command.startsWith("GEN:")) {
        parseGenCommand(command);
```

```
    }
    else if (command == "MODE:FREDKIN1") {
        if (!simulationRunning) {
            runSimulationFredkin1();
        }
    }
    else if (command == "MODE:FREDKIN2") {
        if (!simulationRunning) {
            runSimulationFredkin2();
        }
    }
}

bool waitForAck(unsigned long timeout = SERIAL_TIMEOUT) {
    unsigned long startTime = millis();
    while (!Serial.available()) {
        if (millis() - startTime > timeout) {
            return false;
        }
    }
    String ack = Serial.readStringUntil('\n');
    return ack == "OK";
}

void parseGenCommand(const String &cmd) {
    int pos1 = cmd.indexOf(':');
    int pos2 = cmd.indexOf(':', pos1 + 1);

    if (pos1 < 0 || pos2 < 0) {
        Serial.println("LOG,Error: Invalid command format");
        return;
    }

    String genStr = cmd.substring(pos1 + 1, pos2);
    generations = genStr.toInt();

    if (generations <= 0) {
        Serial.println("LOG,Error: Invalid generation count");
        return;
    }

    String bits = cmd.substring(pos2 + 1);
    if (bits.length() != WIDTH * HEIGHT) {
        Serial.println("LOG,Error: Invalid grid data length");
        return;
    }

    // Remplir la grille
    int idx = 0;
```

```
    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
            grid[x][y] = (bits.charAt(idx++) == '1') ? 1 : 0;
        }
    }

    drawGrid();
    Serial.println("LOG,Grid initialized");
}

void drawGrid() {
    tft.fillScreen(WHITE);

    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
            uint16_t color = (grid[x][y] == 1) ? RED : WHITE;
            tft.fillRect(x * CELL_SIZE, y * CELL_SIZE, CELL_SIZE, CELL_SIZE,
color);
            tft.drawRect(x * CELL_SIZE, y * CELL_SIZE, CELL_SIZE, CELL_SIZE,
BLACK);
        }
    }
}

void runSimulationFredkin1() {
    simulationRunning = true;

    for (int g = 1; g <= generations; g++) {
        if (!evolveFredkin1()) {
            Serial.println("LOG,Error: Simulation interrupted");
            break;
        }

        drawGrid();

        // Stats
        int aliveCount = countAliveCells();
        int deadCount = (WIDTH * HEIGHT) - aliveCount;

        // Log principal
        Serial.print("LOG,FREDKIN1,GEN:");
        Serial.print(g);
        Serial.print(",ALIVE:");
        Serial.print(aliveCount);
        Serial.print(",DEAD:");
        Serial.println(deadCount);

        // Grille
        sendGridState();
```

```
        // Attendre acknowledgement avec timeout
        if (!waitForAck()) {
            Serial.println("LOG,Error: Ack timeout");
            break;
        }
    }

    simulationRunning = false;
}

bool evolveFredkin1() {
    if (!sharedBuffer) return false;

    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
            int sumNeighbors = 0;

            // Voisins orthogonaux
            if (y > 0) sumNeighbors += grid[x][y - 1];
            if (y < HEIGHT - 1) sumNeighbors += grid[x][y + 1];
            if (x > 0) sumNeighbors += grid[x - 1][y];
            if (x < WIDTH - 1) sumNeighbors += grid[x + 1][y];

            sharedBuffer[y * WIDTH + x] = (sumNeighbors % 2);
        }
    }

    // Copie du buffer vers la grille
    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
            grid[x][y] = sharedBuffer[y * WIDTH + x];
        }
    }

    return true;
}

void runSimulationFredkin2() {
    simulationRunning = true;

    for (int g = 1; g <= generations; g++) {
        if (!evolveFredkin2()) {
            Serial.println("LOG,Error: Simulation interrupted");
            break;
        }

        drawGrid();
```

```
        // Stats
        int aliveCount = countAliveCells();
        int deadCount = (WIDTH * HEIGHT) - aliveCount;

        // Log principal
        Serial.print("LOG,FREDKIN2,GEN:");
        Serial.print(g);
        Serial.print(",ALIVE:");
        Serial.print(aliveCount);
        Serial.print(",DEAD:");
        Serial.println(deadCount);

        // Grille
        sendGridState();

        // Attendre acknowledgement avec timeout
        if (!waitForAck()) {
            Serial.println("LOG,Error: Ack timeout");
            break;
        }
    }

    simulationRunning = false;
}

bool evolveFredkin2() {
    if (!sharedBuffer) return false;

    const int8_t offsets[8][2] = {
        {-1,-1}, {0,-1}, {1,-1},
        {-1, 0},         {1, 0},
        {-1, 1}, {0, 1}, {1, 1}
    };

    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
            int sum = 0;
            // Vérification des 8 voisins
            for (int i = 0; i < 8; i++) {
                int nx = x + offsets[i][0];
                int ny = y + offsets[i][1];
                if (nx >= 0 && nx < WIDTH && ny >= 0 && ny < HEIGHT) {
                    sum += grid[nx][ny];
                }
            }
            sharedBuffer[y * WIDTH + x] = sum % 2;
        }
    }
```

```
    // Copie du buffer vers la grille
    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
            grid[x][y] = sharedBuffer[y * WIDTH + x];
        }
    }

    return true;
}

int countAliveCells() {
    int count = 0;
    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
            if (grid[x][y] == 1) count++;
        }
    }
    return count;
}

void sendGridState() {
    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
            Serial.print(grid[x][y]);
        }
        Serial.println();
    }
}

void cleanup() {
    if (sharedBuffer) {
        delete[] sharedBuffer;
        sharedBuffer = nullptr;
    }
}
```