

# Synthèse des travaux effectués dans le cadre de maths-en-jean

## Les Automates Cellulaires : Le jeu de la vie

Problématique : Comment connaître les situations  
antécédentes ( $t_{-1}$ ) d'une situation ( $t_0$ ) ?

# Sommaire

1. Préambule
2. Recherches
  - a. Antécédents d'une seule cellule (schémas)
  - b. Arbres: un antécédent d'une grille 3x3
  - c. Tous les antécédents d'une grille 3x3
  - d. Multiprocessing: Tous les antécédents de toutes les grilles 3x3
  - e. Projet de la salle informatique
  - f. Optimisation: JIT (Pypy) et symétries/rotations
  - g. Distributed computing: tous les antécédents de toutes les grilles 4x4
  - h. Premiers résultats
3. Ouvertures
  - a. Etude (estimation) des antécédents sur de plus grandes configurations
  - b. Ordinateur quantique
  - c. Chiffrement
  - d. Jardin d'Eden
4. Conclusion
5. Lexique

## Préambule

Le jeu de la vie est un automate cellulaire bidimensionnel, c'est à dire une grille régulière en deux dimensions de "**cellules**" caractérisées chacune par un **état** : vivante ou morte, l'état de chaque cellule à l'étape suivante dépendant de l'état de ses voisines directes. Il a été inventé par John Conway en 1970 et est régit par les règles locales suivantes :

- Une cellule morte possédant exactement 3 voisines vivantes devient vivante (elle naît).
- Une cellule vivante possédant 2 ou 3 voisines vivantes reste vivante, sinon elle meurt.

L'intérêt de cet automate cellulaire est qu'il met particulièrement bien en avant la façon dont des comportements complexes peuvent émerger de règles simples. Le jeu de la vie est notamment dit "Turing-complet", c'est à dire qu'il est possible d'y programmer une machine de Turing.

Les automates cellulaires sont sujets à de nombreuses recherches, très variées. Il s'agit en effet d'un domaine vaste qui présente diverses possibilités d'études. Nous avons, quant à nous, décidé de chercher à faire fonctionner l'automate cellulaire en marche arrière.

## Recherches

### Antécédents d'une seule cellule (schémas)

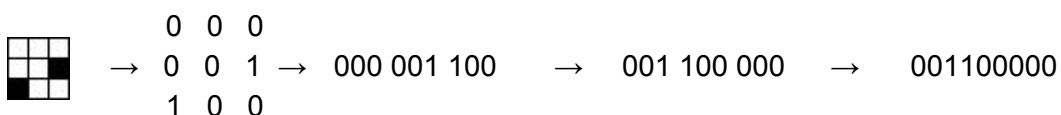
La première étape a été de lister quelles **situations** amenaient une cellule morte et quelles situations amenaient une cellule vivante à la **génération** suivante.

L'état d'une cellule à la génération suivante dépendant de l'état de ses 8 voisines lors de la génération présente, nous nous sommes donc intéressés à une grille de 3 sur 3 (en rouge dans l'illustration) dont la cellule centrale est la cellule dont on cherche à connaître l'état à la génération suivante.



Chaque cellule de cette grille pouvant être soit morte soit vivante, il existe au total  $2^9$ , soit 512 **configurations** possibles. En représentant sous forme binaire de neufs **bits** cette grille (avec l'état vivant correspondant à 1 et l'état mort à 0), la totalité de ces configurations est donc représentée par les 512 premiers chiffres binaires.

*Exemple:*



Cette configuration correspond donc au binaire : 001100000 soit 96 en décimal

Attention! Pour des raisons pratiques, nous avons "inversé" la place des bits sur la grille afin que la configuration 1 (en décimal) corresponde à celle-ci:



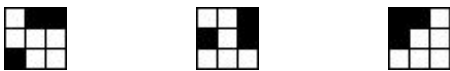
Le bit tout à droite de la représentation binaire correspond ainsi à la cellule en haut à gauche sur la grille. Voici donc les correspondances suivantes:

On appelle ces 512 configurations élémentaires les **schémas**.

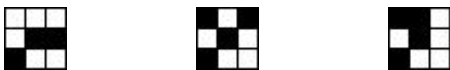
On les classe alors en 4 catégories:

- Les schémas transformant la cellule centrale morte en cellule vivante.
- Les schémas conservant la cellule centrale vivante en cellule vivante.
- Les schémas transformant la cellule centrale vivante en cellule morte.
- Les schémas conservant la cellule centrale morte en cellule morte.

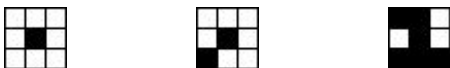
Exemples:



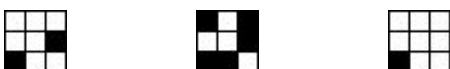
Exemple de schémas générant une cellule vivante à partir d'une cellule morte



Exemple de schémas conservant une cellule vivante à partir d'une cellule vivante



Exemple de schémas générant une cellule morte à partir d'une cellule vivante



Exemple de schémas conservant une cellule morte à partir d'une cellule morte

Afin d'obtenir rapidement une liste de chacune de ces catégories, nous avons programmé un algorithme (en Python) triant chacun des 512 schémas selon 2 critères:

- La valeur du bit central (5e bit), représentant l'état initial de la cellule centrale (à laquelle on s'intéresse)
- **Le poids de Hamming**, représentant le nombre de cellules vivantes sur le schéma étudié.

Concrètement, l'algorithme trie ainsi les schémas:

1. Cellule initiale morte, cellule finale vivante
  - Bit central valant 0
  - Poids de Hamming valant 3
2. Cellule initiale vivante, cellule finale vivante
  - Bit central valant 1
  - Poids de Hamming compris entre 3 et 4
3. Cellule initiale morte, cellule finale morte
  - Bit central valant 0
  - Poids de Hamming différent de 3
4. Cellule initiale vivante, cellule finale morte
  - Bit central valant 1
  - Poids de Hamming non compris entre 3 et 4

Les schémas sont donc triés selon les règles du jeu de la vie.

On peut simplifier cette liste en 2 catégories:

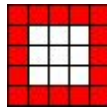
- Les schémas donnant une cellule morte à la génération suivante.
- Les schémas donnant une cellule vivante à la génération suivante.

Cette liste donne donc toutes les situations antécédentes possibles pour une cellule donnée, sans prendre en compte l'état de ses voisines.

### Arbres: un antécédent d'une grille 3x3 (sur grille 5x5)

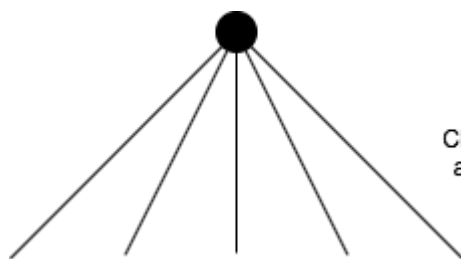
Afin de trouver les possibles configurations de la génération précédente pour la configuration d'une grille donnée, il convient alors de combiner ces schémas. Concrètement, on cherche quelles sont les combinaisons de schémas compatibles. Si on trouve une telle combinaison, on obtient alors un antécédent de la configuration.

Nous avons d'abord décidé de nous attaquer à une petite grille: 3 cellules sur 3 cellules. Cependant, nous avons remarqué en observant le jeu de la vie que certaines configurations précèdent des configurations plus petites : chaque cellule est influencée par ses voisines directes mais si nous avons continué à travailler sur une grille en 3x3, les cellules sur le bord de la grille n'auraient pas eu certaines de leurs voisines. C'est pourquoi nous avons décidé de chercher un antécédent non pas dans une grille de 3 sur 3, mais dans une grille plus grande de 5 sur 5 en ajoutant une "bande vide autour de chaque situation (en rouge sur l'illustration ci dessous).



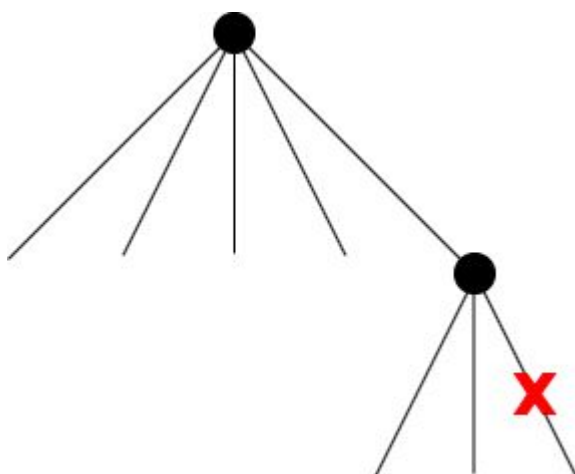
Nous avons donc conçu un algorithme en Python capable de faire cela. On peut visualiser l'action de cet algorithme sous la forme d'un "arbre" ainsi:

Tout d'abord, on cherche l'état de la première cellule de la grille. On en déduit ainsi une liste de schémas possibles dans l'entourage de cette cellule. Ces possibilités forment les premières branches de notre arbre.



Chaque branche correspond à un des schémas amenant la première cellule à son état actuel

On s'enfonce alors d'un niveau en choisissant la première branche de l'arbre. Sur cette branche, on considère le schéma associé pour la première cellule. On cherche alors l'état de la seconde cellule de la grille, et on en déduit alors une nouvelle liste de schémas possibles dans l'entourage de cette cellule. Cependant, certains de ces schémas ne sont pas compatibles avec le premier. On les élimine donc. Les possibilités restantes forment les secondes branches de notre arbre.



Chaque branche correspond à un des schémas amenant la deuxième cellule à son état actuel.

Cependant, certains schémas sont incompatible avec celui de la branche mère. Ils sont donc éliminés.

On continue de s'enfoncer dans l'arbre jusqu'à ce qu'une branche ne donne aucune solution compatible. On supprime alors la branche et on remonte à son noeud de départ, puis on passe à la branche suivante. Une fois que toutes les branches d'un noeud ont été explorées,



des 512 configurations possibles sur une grille de 3x3. Afin d'optimiser le processus, nous avons mis en place un système de "multiprocessing" (**parallélisme informatique**) dans le but d'exploiter la totalité des coeurs de notre ordinateur.

Chaque processus (instance du programme) cherchait les antécédents d'une situation. Un processeur ayant habituellement entre 4 et 8 coeurs, il est capable de faire tourner simultanément entre 4 et 8 processus (autant que son nombre de coeur) à plein régime. Ainsi, nous avons pu traiter simultanément jusqu'à 8 situations en même temps ce qui permet d'augmenter considérablement la vitesse d'exécution.

Au bout de quelques heures, nous avons alors calculé la totalité de ces fichiers.

## Projet de la salle informatique

Afin d'obtenir plus de données, nous avons décidé d'essayer de chercher la totalité des antécédents (sur une grille de 6x6) de la totalité des configurations possibles sur une grille de 4x4 cellules. Cela représente  $2^{16}$  soit 65 536 configurations. Le défi était donc considérable, en comparaison des 512 configurations de 3x3. De plus, le nombre d'antécédent pour chaque configuration est plus élevé qu'en 3x3, et il en va de même pour la complexité de leur recherche.

Afin d'effectuer ces calculs, un simple ordinateur ne suffisait pas. Nous avons donc eu l'idée d'utiliser une salle informatique entière du lycée pendant tout un week-end. Cependant, même les 40 postes de cette salle n'étaient pas suffisamment rapides. Nous avons donc besoin d'optimiser notre programme.

## Optimisation: JIT (Pypy) et symétries/rotations


Nous avons travaillé sur deux caractéristiques du programme: sa vitesse d'exécution et le nombre de configurations à rechercher.


Un premier nettoyage du code nous a permis d'améliorer un peu sa vitesse d'exécution : grâce à un outil de profilage, nous avons pu nous rendre compte que la création de matrices est extrêmement chronophage en Python et que des fonctions créant des matrices étaient appelées plusieurs milliers de fois ce qui ralentissait grandement le programme. En les modifiant afin qu'elles ne génèrent plus de tableaux, nous avons pu largement augmenter la vitesse d'exécution. Mais le gain principal que nous avons obtenu provient de l'utilisation d'une "JIT" ("Just-in-time compilation", "Compilation à la volée" en français): Pypy. Cet outil permet d'exécuter des programmes Python sans utiliser le **compilateur** initial lisant le code ligne par ligne mais plutôt de prendre celui-ci dans son entièreté et d'analyser quelles sont les parties prenant le plus de temps afin de les optimiser. Ainsi, plus un morceau de code est exécuté, plus il va être analysé et donc optimisé. Notre programme étant composé de fonctions répétées plusieurs milliers de fois, l'utilisation de Pypy permît d'accélérer très largement la vitesse d'exécution.

De plus, nous avons remarqué qu'il existait des éléments de symétrie entre certaines situations, symétries qui se retrouvaient dans leurs antécédents. En effet, comme nous travaillons dans un carré, il est possible de faire des rotations ou des symétries d'une situation pour la faire correspondre à une autre dont nous connaissons déjà les antécédents et donc d'en déduire les antécédents sans calcul supplémentaire.



## Exemple

Si nous voulons déterminer les antécédents de la situation 12288 : , il suffit de connaître les antécédents de la situation 3 (qui ont donc été précédemment calculés)

puisque celle-ci est une image par symétrie axiale horizontale de la 12288 : , ce qui nous permet d'en déduire les antécédents par la même transformation.

Cette manipulation nous a permis de diviser le nombre de situation à calculer par environ 8.

## Distributed computing : Tous les antécédents de toutes les grilles 4x4

Une fois le programme optimisé, calculer l'entièreté des antécédents de toutes les grilles en 4x4 aurait pris un peu plus d'une nuit avec la quarantaine d'ordinateur à notre disposition. Mais la semaine avant la date convenue, le gouvernement ordonna la fermeture des lycées ce qui empêcha de mettre en place notre projet. Afin de finaliser ce qui a été commencé nous avons décidé de distribuer les calculs sur plusieurs ordinateurs personnels, une technique initiée par le projet SETI@home, un projet universitaire américain visant à détecter une vie intelligente extraterrestre en utilisant la puissance de calcul des ordinateurs de particuliers. Ainsi, nous avons créé une page web (encore disponible ici : <https://jupitershost.hopto.org/distributedcomputing/>) permettant de télécharger le code nécessaire aux calculs. Une fois lancé, le programme calculait les antécédents d'une situation demandée par le serveur central (hébergé sur un raspberry pi 3B+) puis lui renvoyait la liste des antécédents. Une fois que le serveur avait obtenu tous les antécédents, la tâche était terminée.

Grâce à la coopération de nos camarades et même de notre professeur de NSI Monsieur Mallet, nous avons pu calculer en quelques heures ce qui nous aurait pris plus d'une semaine de calculs en continu.

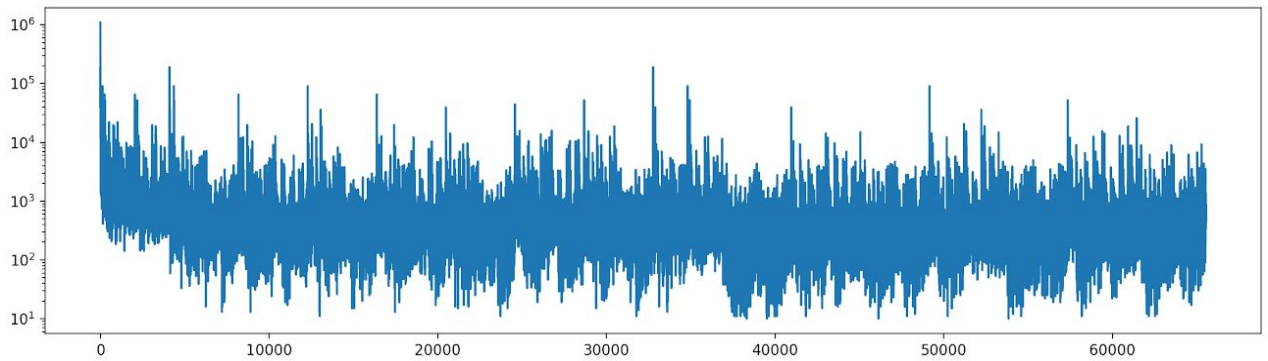
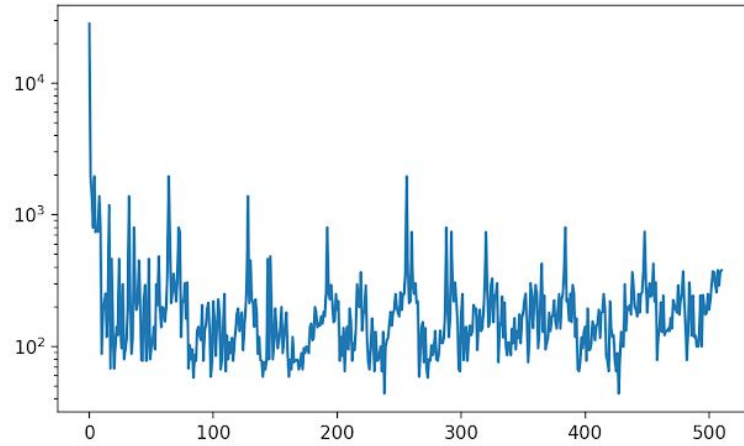
Nous avons alors sur un disque dur l'entièreté des antécédents de l'entièreté des situations possibles dans une grille de 4x4.

## Premiers résultats

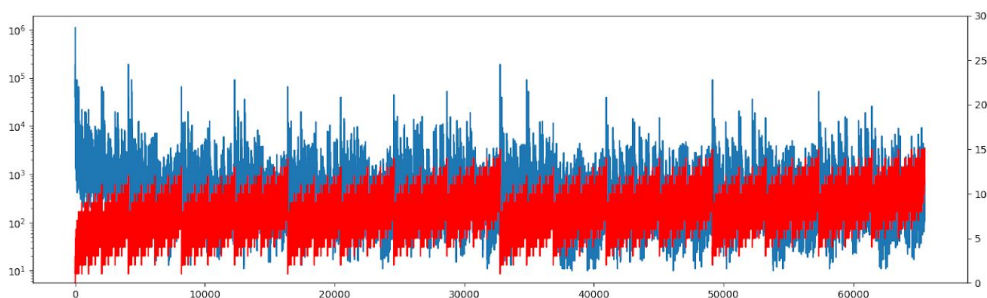
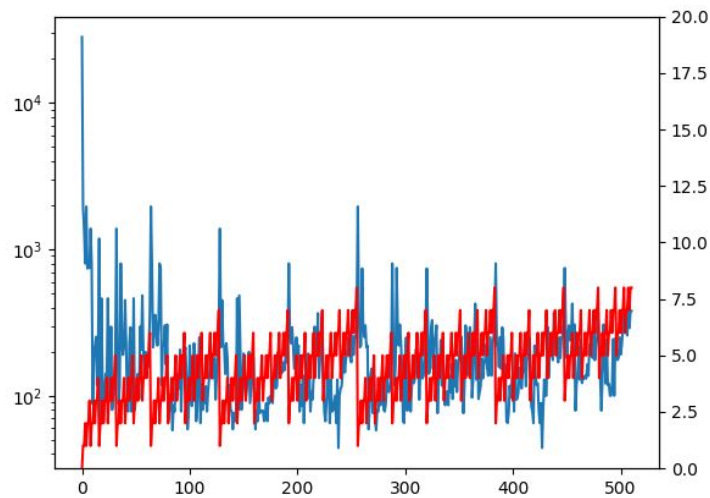
Grâce à ces recherches, nous avons pu tirer plusieurs observations.

La première remarque que nous pouvons faire, c'est que nous avons pu déterminer des antécédents pour chaque situation ce qui signifie qu'aucun jardin d'éden n'existe sur des grilles de 2x2, 3x3 ou 4x4.

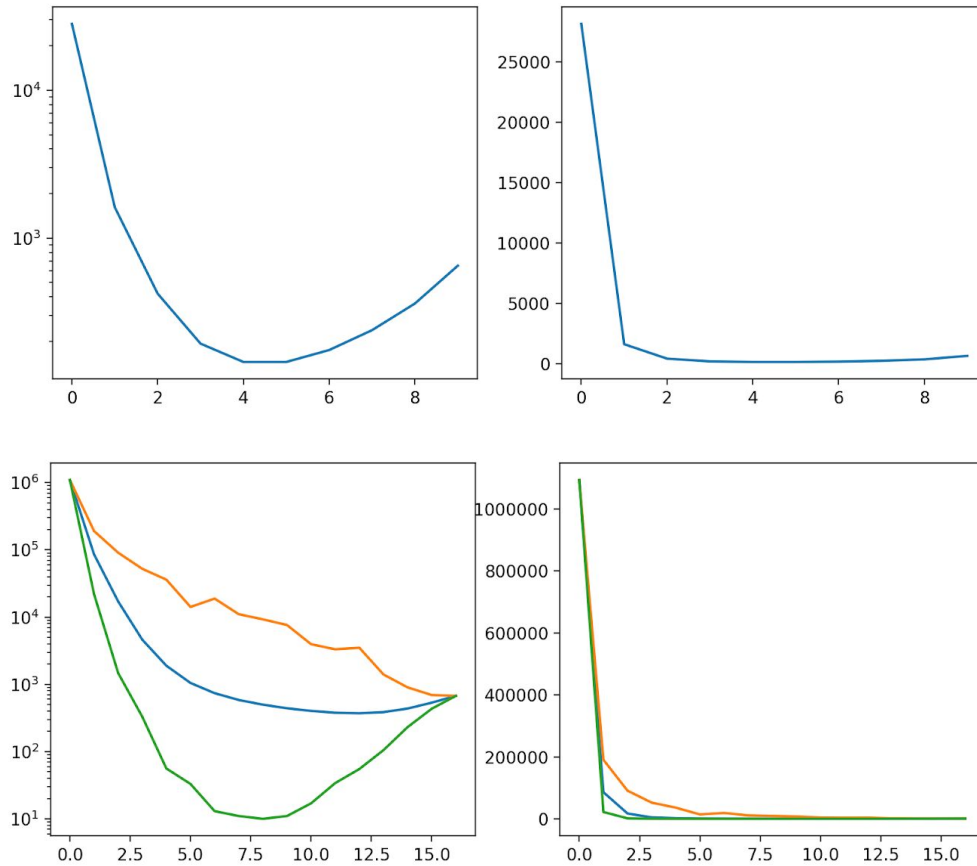
Nous avons ensuite exprimé le nombre d'antécédents en fonction de la situation. On a ainsi obtenu les graphiques suivants, respectivement en 3x3 et 4x4 :



Nous avons ainsi pu relever une périodicité dans les courbes, périodicité qui se corrèle avec le poids de hamming (en rouge sur les graphiques) de chaque situation :



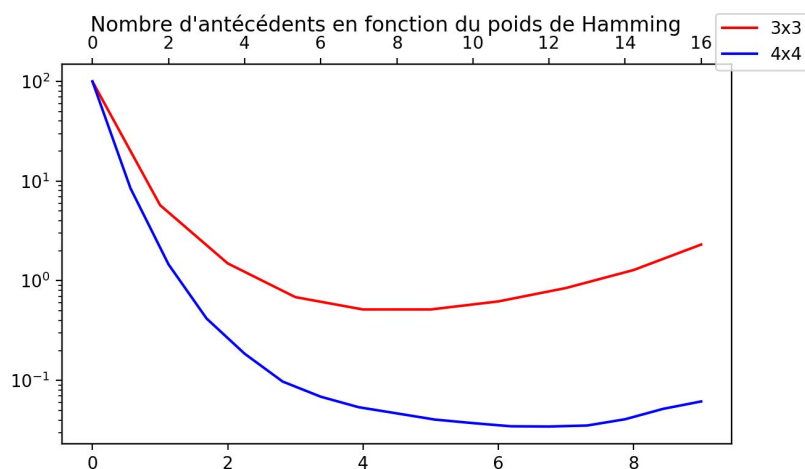
Il nous a donc semblé pertinent d'exprimer le nombre moyen d'antécédents en fonction du poids de hamming, respectivement en 3x3 et 4x4 :



Nous avons par ailleurs fait apparaître la courbe représentant le nombre minimal (en vert) et maximal (en orange) d'antécédents pour chaque poids, ce qui signifie que si nous faisons apparaître toutes les courbes, elles seraient comprises entre l'orange et la verte.

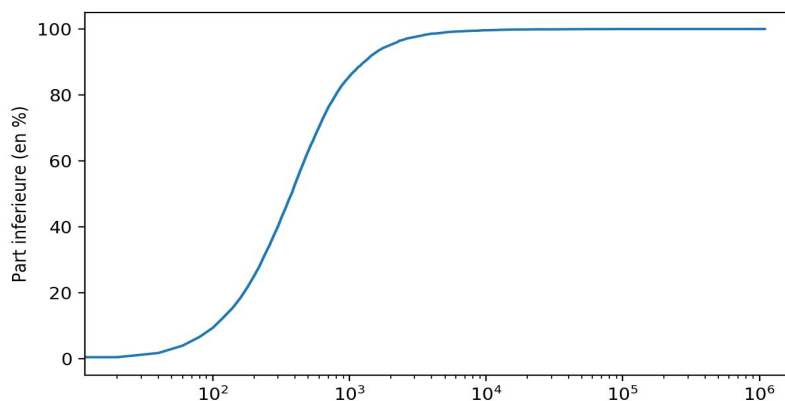
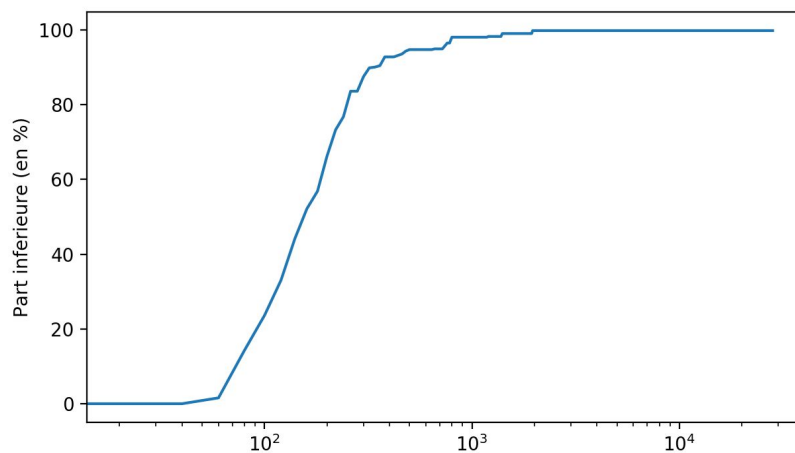
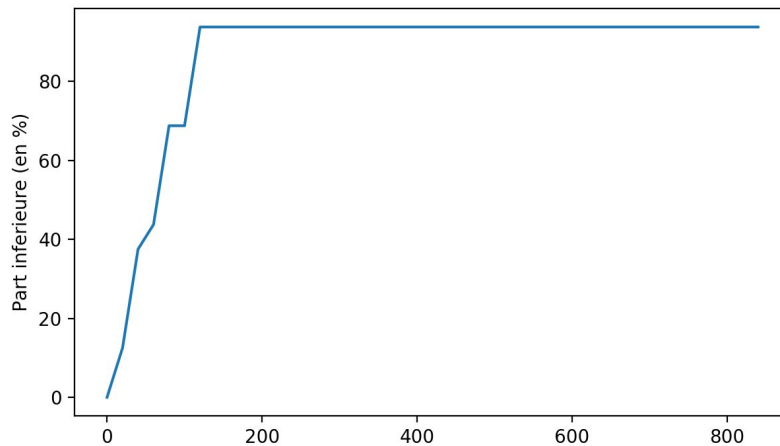
Une autre remarque que nous pouvons faire, c'est que le nombre d'antécédents est minimal lorsque le poids de hamming est égal à la moitié du poids maximal, ce qui signifie concrètement qu'une situation possède en moyenne moins d'antécédents lorsque la moitié de ses cellules environ sont vivantes.

Sur ce graphique, nous avons fait apparaître les deux courbes pour bien mettre en lumière leur similitude :



Nous avons ensuite cherché à connaître la répartition du nombre d'antécédents. Nous avons d'abord eu recours à des histogrammes mais ceux-ci étaient très peu lisibles à cause de la très grande étendue des résultats.

Nous avons donc tracé la fonction de répartition du nombre d'antécédents : chaque point représente la part des situations ayant un nombre d'antécédents inférieur ou égal à l'abscisse de ce point. Ces graphiques sont respectivement les fonctions de répartition du nombre d'antécédents en 2x2, 3x3 et 4x4.



Ce que l'on peut observer, c'est qu'à l'échelle logarithmique lorsque le nombre de cases grandit, la distribution se rapproche d'une distribution normale (courbe de Gauss). En effet, lorsque le nombre de cases est faible (en 2x2), la fonction de répartition est très éloignée de celle de la loi normale mais en 3x3 celle-ci s'en rapproche bien plus et en 4x4 elle sont tout

à fait similaires. On peut donc conjecturer que la fonction de répartition des antécédents d'une grille infinie est exactement identique à celle de la loi normale.

## Ouverture

### Étude (estimation) des antécédents sur de plus grandes configurations

La première question se posant après ces premiers travaux est bien entendue celle des limites de notre algorithme. Jusqu'à quelle taille de grille pourrions-nous aller ainsi ? La complexité augmentant significativement avec l'expansion de la grille, nous serions sans doute vite limités avec cette méthode. En nous adaptant en conséquence, quelles informations pourrions-nous tout de même espérer obtenir ?

Une étude plus poussée des premiers résultats permettrait sans doute d'effectuer des conjectures quant à certaines propriétés du jeu de la vie, et peut-être même faciliter la recherche d'antécédents sur de plus grandes configurations.

En quelque sorte, le problème sur lequel nous travaillons peut se résumer à chercher les possibles "passés" d'une configuration du jeu de la vie. Avec un peu d'imagination, on peut donc également se demander ce que pourrait nous apprendre cette "machine à voyager dans le temps" sur des systèmes bien plus complexes tels que le monde réel.

### Recherche de configurations particulières

L'utilisation de notre algorithme pourrait également servir à rechercher ou identifier des configurations particulières. Par exemple, il pourrait permettre de trouver des structures cycliques de différentes périodes ou encore des situations stables. Il pourrait peut-être aussi permettre de répondre à des questions telles que le "problème du père unique" (à savoir: "Existe-t-il une configuration stable dont l'unique antécédent est elle même ?") formulé par John Conway et encore non résolu à ce jour.

### Ordinateur quantique

Bien que nos connaissances dans le domaine de l'informatique quantique soient bien trop limitées pour l'affirmer avec certitude, on peut supposer que l'utilisation d'un ordinateur quantique pourrait significativement améliorer la vitesse de calcul. En effet, les ordinateurs quantiques sont conçus pour résoudre des problèmes "massivement parallèles", ce qui laisse penser que notre algorithme pourrait être particulièrement adapté pour ce type de machine. Alors que certaines entreprises affirment avoir atteint la suprématie quantique, il serait donc intéressant de se pencher de façon plus approfondie sur cette question afin de déterminer si l'informatique quantique permettrait bel et bien une augmentation considérable de l'efficacité de notre méthode.

### Chiffrement

Il semble possible d'utiliser ces recherches pour chiffrer des communications de manière symétrique. Il faudrait utiliser le protocole suivant :

- Le client qui initie la connexion génère un nombre aléatoire  $N$  qui servira de clé de chiffrement.

- Il la partage avec l'autre client de manière sécurisée (protocole RSA par exemple)
- Il récupère le premier octet de la communication puis utilise l'algorithme développé par nos recherches afin de "reculer" de N étapes. Les antécédents sont choisis aléatoirement.
- Il envoie la série de dix bits ainsi obtenue puis répète l'opération sur chaque octet
- Dès qu'il reçoit une série de dix bits, le deuxième client applique les règles du jeu de la vie pour "avancer" de N étapes.
- Une fois que toutes les séries de 10 bits sont déchiffrées, la communication est terminée.

Ce système est sécurisé mais est bien moins optimal que des algorithmes tels que AES car il nécessite une grande puissance de calcul sans pour autant être asymétrique. Il est cependant peut-être possible d'appliquer nos recherches dans un algorithme de chiffrement asymétrique mais nous n'avons pas trouvé de solution viable.

## Jardin d'eden

Nos travaux peuvent avoir des applications dans la recherche de **Jardins d'Eden**. En effet, pour être sûr qu'une situation n'est pas un jardin d'Eden, il faut et suffit qu'elle ait un antécédent. Pour une situation donnée, il suffit donc de chercher un seul antécédent et si l'algorithme n'en trouve pas, celle-ci est un Jardin d'Eden. Concrètement, pour connaître l'ensemble des Jardins d'Eden qui existent dans une grille donnée, il faut donc tester l'une après l'autre chaque situation.

Cependant, comme nous pouvons le voir sur le graphique représentant la répartition du nombre d'antécédent en fonction du poids de Hamming, le nombre d'antécédent faiblit lorsque le poids est d'environ la moitié du nombre de cellules, ce qui nous permet de conjecturer que la majorité des Jardins d'Eden possèdent une densité proche de 0,5. Cette observation se retrouve d'ailleurs dans la liste des 23 Jardins d'Eden du LifeWiki dont la densité moyenne est de 0.56 +/- 0.03.

Il semble donc pertinent de ne pas rechercher les jardins d'Eden dans l'ensemble des configurations sur une grille donnée mais de plutôt se concentrer sur un échantillon dont la densité serait proche de cette valeur.

## Conclusion

L'algorithme est cependant encore largement améliorable puisqu'il est actuellement très gourmand en ressources, ce qui pose problème lorsque l'on s'attaque à des grilles plus imposantes. De plus, il est impossible d'affirmer avec certitude que nous avons trouvé l'entièreté des antécédents : nous sommes sûrs que les antécédents trouvés sont corrects (ils ont été testés) mais rien ne nous permet de conclure qu'il n'en n'existerait pas d'autres, si ce n'est la confiance en notre travail. Les données produites ouvrent de très nombreuses possibilités, tant par leur quantité que leur nouveauté : nous n'avons pas trouvé de traces d'autres études portant précisément sur cette problématique. C'est donc à priori la première fois qu'un programme est capable de remonter par cette méthode dans le jeu de la vie. Nous sommes donc convaincus que leur analyse permettrait d'approfondir encore nos connaissances dans le domaine.

## Lexique

**Antécédent:** Un antécédent d'une situation S est une situation ayant pu amener à cette situation S après une génération.

**Cellule:** Une cellule est une case composant la grille qui forme l'aire de jeu.

**Configuration:** Ensemble des états possibles de chaque cellule composant une grille.

**Compilateur:** Programme permettant de transformer le code d'un algorithme en binaire exécutable par un ordinateur.

**Etat (d'une cellule):** Il existe deux états possibles : mort (représenté par une case blanche ou un bit à 0) et vivant (représenté par une case noire ou un à 1).

**Jardin d'Eden:** Situation orpheline, c'est à dire ne possédant aucune situation antécédente.

**Schéma:** Situation en 3x3 évoluant en une situation dont la cellule centrale est dans un état en particulier. Les schémas représentent donc l'ensemble des cellules influant sur la cellule centrale (qui influe aussi sur elle même).

**Situation:** Ensemble des états de chaque cellule composant une grille.

**Génération:** Ensemble du système une fois que les règles du jeu de la vie ont été appliquées.

**Parallélisme informatique (multiprocessing):** Technique consistant à mettre en œuvre des architectures algorithmes visant à traiter des informations de manière simultanée. Utilise la segmentation en coeurs des processeurs en leur confiant chacun une tâche.

**Poids de Hamming:** Nombre de bits à 1 dans un nombre en binaire.

*Nous dédions cette synthèse, ainsi que nos humbles recherches, à John Horton Conway, mathématicien de génie et père du jeu de la vie, emporté par la COVID-19 le 11 avril 2020, à l'âge de 82 ans.*