



UNIVERSITÉ  
CAEN  
NORMANDIE

## LICENCE 2 INFORMATIQUE

# Simulateur du jeu de la vie

### *Auteurs:*

Sow MARIAMA  
SAOUDATOU  
Diallo MAMADOU ALPHA  
Kotin MAHUDO  
NARCISSE  
EL FEJER MARWA

### *Chargée du cours:*

BonnetGRÉGORY

### *Chargée du TP:*

AntoninCALLARD

26 Avril 2023

# Table des matières

1	Introduction . . . . .	2
1.1	Contexte et objectifs du projet . . . . .	2
2	Mise en place du projet . . . . .	2
3	Jeu de la vie . . . . .	3
3.1	Principe de fonctionnement . . . . .	3
3.2	Architecture du programme . . . . .	3
4	Fonctionnalités implémentées . . . . .	6
4.1	Type de voisinage . . . . .	6
4.2	Type de règles . . . . .	7
4.3	Patterns . . . . .	7
4.4	Hashlife . . . . .	9
4.5	Simulateur . . . . .	13
4.6	Les autres fonctionnalités . . . . .	14
5	Expérimentations et Usages . . . . .	14
5.1	Lancement de l'application . . . . .	14
5.2	Implémentation de l'Interface graphique . . . . .	14
5.3	Fonctionnement de l'interface graphique . . . . .	15
5.4	Test du Logiciel . . . . .	18
6	Conclusions . . . . .	19
6.1	difficultés rencontrés . . . . .	19
6.2	Bilan du projet . . . . .	20
6.3	Améliorations Possibles . . . . .	20

# 1 Introduction

Dans le cadre de notre formation en développement logiciel, nous avons été confrontés à un choix crucial : choisir un sujet de projet à développer dans le langage de programmation orienté objet Java. Parmi les sujets proposés, nous avons été particulièrement attirés par le jeu de la vie. En effet, ce dernier offre de nombreuses possibilités de personnalisation et d'optimisation, ce qui nous a permis de développer notre capacité à concevoir des solutions efficaces et élégantes pour des problèmes complexes. Nous avons également été motivés par le défi intellectuel que représente la conception et l'implémentation d'un projet de cet envergure, ainsi que par la perspective de pouvoir expérimenter et manipuler le modèle pour découvrir de nouveaux comportements .

Dans cette première partie de notre rapport, nous allons présenter le contexte et les objectifs de notre projet, ainsi que les principes de base du jeu de la vie. Nous décrirons ensuite l'organisation de notre projet, en expliquant l'architecture du programme et les éléments techniques utilisés. Enfin, nous présenterons les expérimentations et les usages que nous avons réalisés avec notre implémentation du jeu de la vie, et nous conclurons en dressant un bilan de nos travaux et en proposant des améliorations possibles pour notre projet

## 1.1 Contexte et objectifs du projet

Le projet que nous avons entrepris consiste à implémenter un automate cellulaire basé sur le célèbre "jeu de la vie" de John Horton Conway, qui est un exemple classique d'automate cellulaire introduit en 1970. Ce jeu se déroule sur une grille bidimensionnelle de cellules qui peuvent être dans l'état "vivant" ou "mort". Les cellules interagissent les unes avec les autres en fonction des règles prédéfinies qui déterminent leur état à l'itération suivante en fonction de l'état de leurs voisins.

Notre objectif principal était de concevoir et de développer une version du jeu de la vie qui soit personnalisable par l'utilisateur. Pour cela, nous avons créé une interface conviviale qui permet à l'utilisateur de modifier différents paramètres du jeu, tels que le type de voisinage et les règles (classiques ou personnalisées). Nous avons également implémenté l'algorithme HashLife pour accélérer le calcul de la grille.

Enfin, nous avons cherché à explorer les possibilités offertes par le jeu de la vie en termes de modélisation et d'expérimentation. À cet effet, nous avons développé des outils de manipulation permettant de visualiser des patterns.

## 2 Mise en place du projet

Pour débiter notre projet, nous avons d'abord cherché à comprendre le jeu de la vie ainsi que ses règles. Nous nous sommes appuyés sur différents supports tels que les documents fournis par le professeur, le simulateur de jeu de la vie Golly, ainsi que d'autres ressources disponibles sur internet.

Une fois que nous avons acquis une idée générale du fonctionnement du jeu, nous avons commencé à élaborer le diagramme de nos différentes classes, réparti les tâches et décidé d'organiser notre projet en différents packages pour une meilleure organisation et maintenabilité du code.

Après avoir développé une première version exécutable du jeu sur la console, nous avons ajouté des fonctionnalités permettant de personnaliser les règles et le type de voisinage. Nous avons également jugé nécessaire d'ajouter un package de tests pour tester les méthodes les plus importantes.

Finalement, nous avons implémenté l'algorithme HashLife pour accélérer le calcul de la grille et optimiser les performances de notre jeu de la vie.

## 3 Jeu de la vie

### 3.1 Principe de fonctionnement

Le jeu de la vie est un automate cellulaire bidimensionnel dans lequel chaque cellule peut être dans l'un des deux états possibles : "vivant" ou "mort". Les cellules interagissent les unes avec les autres selon des règles prédéfinies, qui déterminent l'état de la cellule à l'itération suivante en fonction de l'état de ses voisins.

Le principe de fonctionnement du jeu de la vie est basé sur quatre règles simples :

- Si une cellule morte a exactement trois voisines vivantes, elle devient vivante à la prochaine itération.
- Si une cellule vivante a deux ou trois voisines vivantes, elle reste vivante à la prochaine itération.
- Si une cellule vivante a moins de deux voisines vivantes, elle meurt à la prochaine itération .
- Si une cellule vivante a plus de trois voisines vivantes, elle meurt à la prochaine itération .

Ces règles simples peuvent donner lieu à des structures émergentes complexes et fascinantes, comme des oscillateurs, des vaisseaux, etc

### 3.2 Architecture du programme

Pour notre application, nous avons adopté l'architecture **MVC** (Modèle Vue Contrôleur). Ainsi, nous avons dédié quelques packages au modèle qui gère toute la logique métier, un package contenant la vue (interface graphique), et des événements capturés sur la vue pour agir sur le modèle (contrôleur). Le modèle est complètement indépendant de la vue, et la vue représente les données du modèle. Nous avons appliqué au MVC un observer pattern. Les classes du modèle (celles pour lesquelles c'est utile) implémentent donc une interface observable et celles de la vue une interface observer pour écouter sur la vue.

- **app** : ce package contient toutes les classes liées à la logique du jeu, notamment les classes **Game** et **Generator**, ainsi que la classe **GeneratorThread** pour la gestion des thread
- **constants** : ce package contient toutes les constantes utilisées dans notre programme, telles que les constantes associées au type de voisinage du jeu et les constantes associées à la règle classique du jeu de la vie. regroupe les classes "NeighborsType" et "Rules"
- **modele** : ce package contient toutes les classes liées à la gestion des données. En plus de contenir les classes "Cellule", "Grid", "Position", le package modele contient deux sous-packages :
  - **hashlife** : ce sous-package contient toutes les parties liées à l'implémentation de Hashlife, une technique avancée pour accélérer les calculs du jeu de la vie, il contient la classe "Hashlife" et "QuadNode"
  - **rule**: ce sous-package contient toutes les parties liées à l'implémentation des règles classiques du jeu de la vie ainsi que des règles personnalisables par l'utilisateur. Il contient des classes pour la définition et l'application des règles, ainsi que pour la gestion des règles personnalisées définies par l'utilisateur. il contient les classes : "Rule", "RuleMulttF", "RuleRange", "RuleFormat"
- **tests** : ce package contient toutes les classes de tests nécessaires pour notre projet, pour assurer le bon fonctionnement de toutes les fonctionnalités du jeu.
- **image** : ce package contient toutes les images utilisées dans notre interface homme-machine, telles que les icônes de boutons et les images de fond d'écran.
- **patterns** : ce package contient tous les fichiers utilisés pour générer les patterns, qui sont des motifs prédéfinis de cellules utilisés dans le jeu.
- **Util** : Le package Util contient toutes les classes liées aux événements, comme les écouteurs d'événements. Ces classes sont utilisées pour écouter les événements déclenchés par l'utilisateur, comme le clic de la souris. Les écouteurs sont ensuite utilisés pour déclencher des actions spécifiques dans le jeu, elle contient les classes "AbstractListenableModel", "ListenableModel" et "ListeningModel"
- **views** : Le package Views contient toutes les classes liées à l'interface utilisateur. C'est là que nous avons stocké toutes les classes qui gèrent l'affichage des éléments graphiques, comme les fenêtres de dialogue et la grille du jeu. Les classes de ce package utilisent les classes du package modele pour récupérer les données liées au jeu et les afficher à l'utilisateur

En organisant notre code de cette manière, nous avons pu séparer efficacement les différentes parties du programme et travailler de manière structurée.

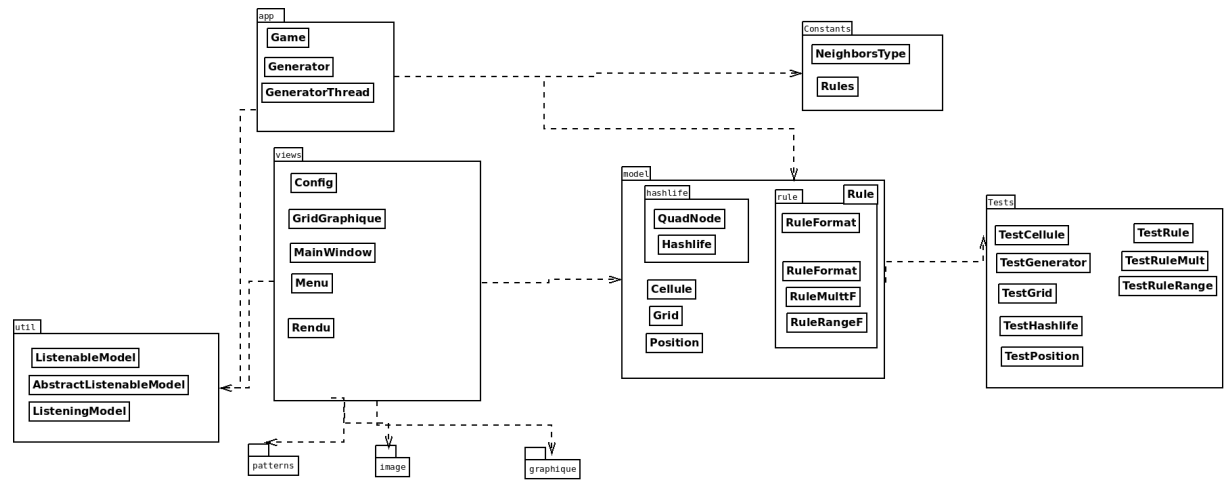


Figure 1: Diagramme des packages

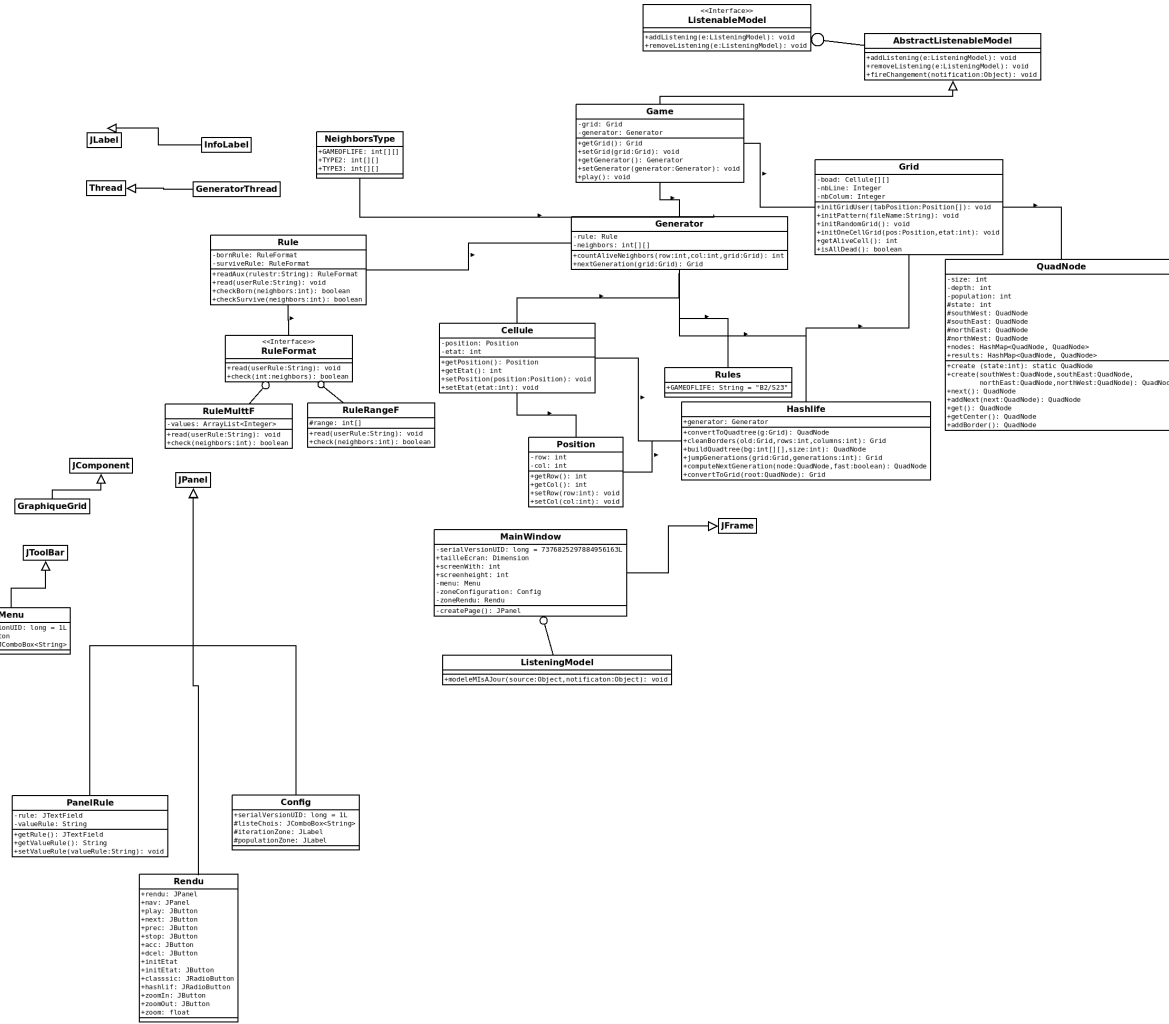


Figure 2: Diagramme des Classes

## 4 Fonctionnalités implémentées

### 4.1 Type de voisinage

Nous avons développé la fonctionnalité **type de voisinage**. Cette fonctionnalité consiste à déterminer quels sont les voisins d'une cellule. Par défaut dans le jeu de la vie, ces voisins sont les huit voisins directs de la cellule. Nous avons fait en sorte que les voisins puissent être configurables. Un voisin pourrait être par exemple **la deuxième cellule à droite** ou encore **la troisième cellule à gauche de deux lignes au-dessus**. Étant donné une cellule de coordonnées

$(x,y)$ , ces voisins correspondraient donc respectivement aux cellules de coordonnées  $(x,y+2)$  et  $(x-2,y-3)$ . Pour implémenter cela, nous avons attribué à notre **générateur** un tableau des voisins d'une cellule correspondant aux positions de ces voisins relativement à la cellule comme décrit précédemment. Ainsi, pour compter le nombre de voisins vivants d'une cellule, nous n'avons qu'à parcourir ce tableau, récupérer les cellules aux positions correspondantes et incrémenter un compteur à chaque fois qu'on tombe sur une cellule vivante.

## 4.2 Type de règles

Nous avons implémenté une fonctionnalité qui permet à l'utilisateur de choisir la règle qu'il souhaite utiliser. Contrairement à la règle classique du jeu de la vie qui impose l'utilisation de la règle **B3/S23**, notre programme permet à **l'utilisateur de spécifier sa propre règle**. Pour ce faire, nous avons créé une interface appelée **"RuleFormat"** qui regroupe deux méthodes : **read** et **check**. La méthode **read** permet de lire une règle tandis que la méthode **check** permet d'appliquer cette règle en fonction du nombre de voisins.

Nous avons également créé deux classes qui implémentent l'interface RuleFormat. La première classe, nommée **"RuleMulttF"**, permet d'implémenter une règle au format **"xy"** tandis que la seconde classe, nommée **"RuleRangeF"**, permet d'implémenter une règle de type **"x-y"**. Nous avons ensuite utilisé une classe appelée **"Rule"** pour traiter une règle spécifique. Cette classe contient une méthode **"read"** qui permet de lire une règle entrée par l'utilisateur et d'instancier soit la classe RuleMulttF soit la classe RuleRangeF en fonction de la forme de la règle.

Par exemple, si l'utilisateur entre la règle **"B23/S2-3"**, l'attribut **"bornRule"** de la classe Rule sera initialisé en instanciant la classe RuleMulttF tandis que l'attribut **"surviveRule"** sera initialisé en instanciant la classe RuleRangeF. De plus, nous avons ajouté une méthode **"checkBorn"** qui appelle la méthode **"check"** sur l'attribut **"bornRule"** et une méthode **"checkSurvive"** qui appelle la méthode **"check"** sur l'attribut **"surviveRule"**. Il est à noter que nous avons utilisé des expressions régulières (regex) pour vérifier la validité de la règle entrée par l'utilisateur. Cette organisation permet une meilleure structure et efficacité de notre programme.

## 4.3 Patterns

Nous avons ajouté une fonctionnalité pour que les joueurs puissent commencer une partie avec une **grille personnalisée**. Pour cela, nous avons implémenté la méthode **"initPattern"** qui permet d'initialiser la grille de jeu en se basant sur les informations contenues dans un fichier texte. Cette méthode prend en paramètre le nom du fichier à lire.

La première étape de la méthode consiste à **créer une matrice appelée "pattern"** qui aura la même taille que la grille de jeu, définie par les attributs **"nbLine"** et **"nbColumn"** de l'objet.



Ensuite, la méthode utilise un objet "Scanner" pour ouvrir le fichier texte et parcourir chaque ligne. Pour chaque ligne lue, la méthode examine chaque caractère et vérifie s'il s'agit d'un 0, d'un 1 ou d'un caractère invalide. Si le caractère est valide, il est stocké dans la matrice "pattern" à l'indice correspondant.

La méthode vérifie également que le nombre de lignes et de colonnes lues ne dépasse pas le nombre de lignes et de colonnes de la grille de jeu. Si cette condition n'est pas remplie, une exception est levée.

Une fois que toutes les lignes ont été lues et que la matrice "pattern" est remplie, la méthode calcule les indices de départ pour placer la matrice "pattern" dans la grille de jeu. Ces indices sont calculés pour centrer la matrice "pattern" dans la grille.

Enfin, la méthode parcourt la matrice "pattern" et place chaque élément dans la grille de jeu à l'indice correspondant. Si une exception est levée pendant la lecture du fichier, un message d'erreur s'affiche sur la console

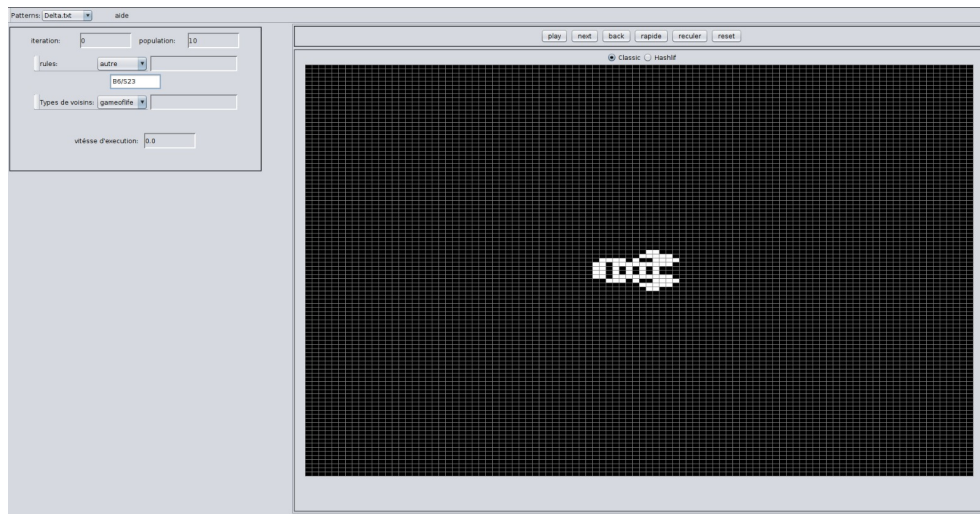


Figure 3: exemple de la grille initialisée avec un pattern

## 4.4 Hashlife

### Hashlife de quoi s'agit t'il ?

HashLife est un algorithme qui améliore considérablement l'efficacité du calcul en **permettant de sauter plusieurs générations et d'évoluer rapidement une grille sur des milliers voire des millions de générations en une seule étape**. En outre, il utilise une représentation mémoire très efficace de la grille, ce qui permet de compresser à la fois l'espace et le temps. En somme, HashLife est un algorithme qui permet d'accélérer considérablement les calculs tout en réduisant l'empreinte mémoire nécessaire pour stocker les données de la grille.

### Implémentation

Nous avons créé une classe **Hashlife** qui prend en attributs un générateur de type "generator", défini dans le package "app". Cette classe contient plusieurs méthodes, notamment **"convertToQuadtree"** qui prend en paramètre un objet de type "Grid" qui représente l'état des cellules du jeu de la vie sous forme de grille. La fonction commence par déterminer la taille maximale de la grille (nombre de lignes ou de colonnes) et trouver la puissance de deux supérieure à cette taille. Ensuite, la fonction crée un tableau de taille size x size appelé "bg" et y copie les valeurs de la grille "g". Les **cellules qui ne sont pas dans la grille sont initialisées avec la valeur -1**. Enfin, la fonction appelle la méthode **"buildQuadtree"** avec le tableau "bg" et la taille "size", pour construire et renvoyer l'arbre quadtree représentant l'état des cellules.

La fonction **"buildQuadtree"** prend en paramètre un tableau d'entiers "bg" représentant une grille de cellules du jeu de la vie et un entier "size" qui indique la taille de la grille. La fonction utilise un algorithme récursif pour construire un arbre quadtree à partir de cette grille. Si la taille de la grille est de 1, la fonction crée un nœud "QuadNode" avec l'état de la cellule unique, en appelant la méthode "create" de la classe "QuadNode". Si la taille de la grille est supérieure à 1, la fonction divise le tableau "bg" en quatre sous-tableaux représentant les quatre quadrants de la grille et appelle récursivement la fonction "buildQuadtree" sur chacun de ces sous-tableaux. Elle crée ensuite un nœud "QuadNode" avec les quatre nœuds retournés par les appels récursifs, en appelant la méthode "create" de la classe "QuadNode".

En résumé, la fonction "buildQuadtree" utilise un algorithme récursif pour diviser une grille de cellules en quatre sous-grilles et créer un arbre quadtree à partir de ces sous-grilles. Elle retourne le nœud racine de l'arbre quadtree, et la fonction "convertToQuadtree" convertit l'état des cellules du jeu de la vie sous forme de grille en un arbre quadtree, qui est utilisé pour stocker et manipuler l'état des cellules dans l'algorithme Hashlife.

Pour stocker les nœuds de l'arbre, nous avons créé une classe "QuadNode" qui possède des attributs tels que la taille, la profondeur, la population et l'état, ainsi que des références aux quatre sous-nœuds. Cette classe a deux constructeurs. Le premier constructeur prend en paramètre un entier "state" qui représente l'état de la cellule du nœud. Ce constructeur initialise les attributs "southWest", "southEast", "northEast" et "northWest" à null, car ce nœud n'a pas de sous-nœuds. Il initialise également l'attribut "state" avec la valeur "state", la profondeur "depth" avec 0 et la taille "size" avec 1. Si l'état de la cellule est -1, cela signifie qu'il n'y a pas de cellule, donc il initialise l'attribut "population" avec 0. Sinon, il initialise "population" avec la valeur "state", qui représente le nombre de cellules vivantes. Le deuxième constructeur prend en paramètre les quatre sous-nœuds "southWest", "southEast", "northEast" et "northWest". Ce constructeur initialise ces sous-nœuds avec les paramètres correspondants et calcule la profondeur "depth" du nœud en ajoutant 1 à la profondeur du sous-nœud "southWest". Il calcule également la taille "size" du nœud en multipliant par 2 la taille du sous-nœud "southWest".

Enfin, il calcule la population "population" du nœud en additionnant les populations des quatre sous-nœuds. En résumé, les deux constructeurs de la classe "QuadNode" permettent d'initialiser les nœuds de l'arbre quadtree en fonction de leur état et de leurs sous-nœuds, et de calculer la profondeur, la taille et la population de ces nœuds.

Nous avons également créé deux HashMaps statiques, "nodes" et "results", pour stocker les nœuds et les résultats de la simulation. Pour manipuler ces nœuds, nous avons écrit deux méthodes "create". Les méthodes "create" de la classe "QuadNode" sont des méthodes statiques qui permettent de créer des nœuds de l'arbre quadtree à partir de différents paramètres. Elles renvoient le nœud créé après l'avoir ajouté à la HashMap "nodes" pour éviter de créer des nœuds dupliqués.

La première méthode "create" prend en paramètre un entier "state" qui représente l'état de la cellule du nœud. Cette méthode crée un nouveau nœud "QuadNode" avec le constructeur qui prend en paramètre "state", puis appelle la méthode "get" sur ce nœud pour le renvoyer après l'avoir ajouté à la HashMap "nodes".

La deuxième méthode "create" prend en paramètre les quatre sous-nœuds "southWest", "southEast", "northEast" et "northWest". Cette méthode crée un nouveau nœud "QuadNode" avec le constructeur qui prend en paramètre les quatre sous-nœuds, puis appelle la méthode "get" sur ce nœud pour le renvoyer après l'avoir ajouté à la HashMap "nodes".

Dans les deux cas, la méthode "get" vérifie si le nœud créé est déjà présent dans la HashMap "nodes". Si le nœud existe déjà, la méthode renvoie la référence vers ce nœud. Sinon, la méthode ajoute le nœud à la HashMap "nodes" et renvoie sa référence.

Une méthode "next" de la classe "QuadNode" permet de récupérer le nœud

"QuadNode" correspondant à la génération suivante depuis la HashMap "results". La méthode utilise l'objet "this" pour récupérer le nœud courant. Elle appelle ensuite la méthode "get" sur la HashMap "results" avec "this" en paramètre pour récupérer le nœud suivant. Si le nœud suivant n'existe pas dans la HashMap "results", la méthode renvoie null.

Ainsi qu'une méthode "addNext" de la classe "QuadNode" ajoute le nœud "QuadNode" correspondant à la génération suivante à la HashMap "results". La méthode utilise l'objet "this" pour récupérer le nœud courant et utilise l'objet "next" pour récupérer le nœud suivant, correspondant à la génération suivante. Elle ajoute ensuite une entrée à la HashMap "results" avec "this" comme clé et "next" comme valeur. La méthode renvoie le nœud "QuadNode" correspondant à la génération suivante, qui a été ajouté à la HashMap "results".

Nous avons également écrit La méthode "get" de la classe "QuadNode" récupère le nœud correspondant à l'objet courant en utilisant la HashMap "nodes". Si le nœud n'existe pas dans la HashMap, la méthode l'ajoute à la HashMap. La méthode utilise l'objet "this" pour récupérer le nœud courant. Elle appelle ensuite la méthode "get" sur la HashMap "nodes" avec "this" en paramètre pour récupérer le nœud correspondant. Si le nœud existe déjà dans la HashMap, la méthode renvoie la référence vers ce nœud. Si le nœud n'existe pas dans la HashMap, la méthode ajoute une entrée avec "this" comme clé et "this" comme valeur dans la HashMap "nodes". Elle renvoie ensuite la référence vers le nœud courant.

Une méthode "getCenter" de la classe "QuadNode" renvoie le nœud "QuadNode" correspondant au centre du nœud courant. La méthode utilise les sous-nœuds "southWest", "southEast", "northEast" et "northWest" de l'objet courant pour créer un nouveau nœud "QuadNode" avec les sous-nœuds "northEast" de "southWest", "northWest" de "southEast", "southEast" de "northWest" et "southWest" de "northEast". La méthode appelle ensuite la méthode "get" sur le nouveau nœud créé pour récupérer le nœud correspondant à l'objet courant en utilisant la HashMap "nodes". Si le nœud n'existe pas dans la HashMap, la méthode l'ajoute à la HashMap. La méthode renvoie ensuite la référence vers le nœud correspondant au centre du nœud courant.

Une méthode "hashCode" de la classe "QuadNode" calcule un code de hachage unique pour l'objet courant en utilisant les codes de hachage des sous-nœuds "southWest", "southEast", "northEast" et "northWest". Si le nœud courant est une feuille, la méthode renvoie simplement l'état de la cellule représentée par le nœud. Sinon, la méthode calcule un code de hachage en utilisant la formule suivante : le code de hachage du nœud courant est égal à 31 multiplié par le code de hachage de chaque sous-nœud "northWest", "northEast", "southWest" et "southEast", additionné au résultat. La méthode renvoie le code de hachage ainsi calculé. La méthode "equals" permet de vérifier si l'objet courant est égal à un autre objet en comparant la profondeur et les sous-nœuds de l'objet courant avec ceux de l'objet passé en paramètre.

Pour optimiser la simulation, on a ajoutés une méthode "cleanBorders" prend en paramètres une grille "old" représentant l'état actuel du jeu de la vie, ainsi que le nombre de lignes et de colonnes de la grille. Elle crée ensuite une nouvelle grille "grid" de la taille spécifiée. La méthode parcourt ensuite les cellules de la grille "old" et copie leur état dans la grille "grid". Cela permet de créer une copie de la grille "old" sans les bordures supplémentaires qui ont été ajoutées pour faciliter la manipulation de la grille avec l'algorithme Hashlife. Enfin, la méthode renvoie la grille "grid" ainsi créée.

Une méthode "jumpGenerations" prend en paramètres une grille "grid" représentant l'état actuel du jeu de la vie, ainsi qu'un entier "generations" indiquant le nombre de générations à sauter. La méthode commence par convertir la grille en un quadtree en utilisant la méthode "convertToQuadtree". Ensuite, elle ajoute des bordures au quadtree pour sauter plusieurs générations à la fois. Le nombre de bordures ajoutées dépend de la profondeur du quadtree et du nombre de générations à sauter. Ensuite, si des bordures ont été ajoutées, la méthode calcule plusieurs générations d'un coup en utilisant la méthode "computeNextGeneration". Elle supprime ensuite les bordures supplémentaires ajoutées en appelant plusieurs fois la méthode "getCenter" sur le quadtree résultant.

Enfin, elle réinitialise le cache de résultats. Si aucune bordure supplémentaire n'a été ajoutée, la méthode calcule les générations une par une en utilisant la méthode "computeNextGeneration". Enfin, la méthode renvoie la grille résultante sans les bordures supplémentaires, en utilisant la méthode "cleanBorders" et en convertissant le quadtree résultant en une grille en utilisant la méthode "convertToGrid".

La méthode "computeNextGeneration" calcule la prochaine génération d'un QuadTree donné. Si le nœud en entrée est une feuille (profondeur 2), la fonction convertit le nœud en une grille, calcule la prochaine génération de la grille à l'aide du générateur et convertit la grille en un nouveau QuadTree. Le QuadNode central du nouveau QuadTree est renvoyé en tant que prochaine génération. Sinon, la fonction divise le nœud en neuf sous-nœuds (n1 à n9) et calcule la prochaine génération pour chaque sous-nœud en appelant récursivement la fonction "computeNextGeneration". Les sous-nœuds résultants (r1 à r9) sont ensuite combinés pour former la prochaine génération du nœud. Si le paramètre "fast" est vrai, les sous-nœuds résultants ne sont pas centrés et leur prochaine génération est également calculée de manière "rapide" en appelant la fonction "computeNextGeneration" avec le paramètre "fast" à "true". Sinon, les sous-nœuds résultants sont centrés et la prochaine génération est calculée à l'aide de la fonction "create" pour créer un nouveau QuadTree à partir des sous-nœuds centrés. Le nœud résultant est ensuite enregistré dans la table de hachage "results" pour une utilisation ultérieure et renvoyé en tant que prochaine génération.

Pour afficher l'état des cellules, on a également créé une méthode "convertToGrid" prend en entrée un objet "QuadNode" et renvoie une grille Grid qui représente l'état des cellules. Elle commence par créer une grille de la taille de la racine de l'arbre QuadNode en utilisant la méthode "getSize()" de la classe QuadNode. Ensuite, elle crée un tableau de cellules board qui contient les états des cellules à partir de l'objet Grid créé. La fonction appelle ensuite la fonction récursive convertToGridRecursive qui va remplir le tableau board en parcourant l'arbre QuadNode. Finalement, la fonction renvoie Grid avec les états des cellules de l'arbre QuadNode.

La méthode "convertToGridRecursive" permet de convertir un arbre de quadtree en une grille de cellules. Elle prend en entrée un nœud de l'arbre "node", une grille de cellules "board", les coordonnées "x" et "y" du coin supérieur gauche de la zone de la grille correspondant à ce nœud, et la taille "size" de cette zone. Si le nœud "node" est une feuille, c'est-à-dire s'il ne possède pas de sous-nœuds, la méthode remplit la zone correspondante de la grille "board" avec des cellules ayant l'état du nœud. Sinon, la méthode divise la zone en 4 sous-zones et appelle récursivement la méthode "convertToGridRecursive" pour chaque sous-nœud en passant les coordonnées et la taille de sa sous-zone correspondante. Ainsi, en combinant cette méthode avec la méthode "convertToGrid" qui initialise une grille de la taille de l'arbre et appelle cette méthode pour remplir la grille à partir de la racine, on obtient une grille de cellules représentant l'état de l'automate cellulaire représenté par l'arbre de quadtree.

Enfin, on a ajouté une méthode "addBorder" de la classe "QuadNode" ajoute une couche de bordure de cellules mortes à un nœud de l'arbre. Elle crée un nouveau nœud "nodeBorder" représentant une cellule morte, puis l'ajoute récursivement au nœud "nodeBorder" précédent pour créer une couche de bordure de profondeur "depth-1". Ensuite, elle crée quatre nouveaux nœuds pour chaque sous-nœud du nœud courant, en utilisant les sous-nœuds correspondants et le nœud "nodeBorder" comme paramètres pour créer un nouveau nœud. Finalement, elle retourne le nouveau nœud créé avec les sous-nœuds modifiés.

Cette approche permet une accélération significative de la simulation et permet de simuler des configurations plus grandes et plus complexes.

## 4.5 Simulateur

Pendant la simulation du jeu de la vie, notre générateur génère des générations de grilles à l'infini (en réalité, la simulation s'arrête lorsque toutes les cellules sont mortes, mais selon la complexité de la grille initiale, cette éventualité peut être très rare, et cela peut donc ressembler à une simulation à l'infini). Ainsi, pendant le calcul de ces générations, il nous était impossible de traiter d'autres opérations sur l'application, comme par exemple capturer un événement de clic sur un bouton stop pour arrêter la simulation. Nous avons donc pensé à implémenter le **multi-threading** pour que le calcul des prochaines générations ne bloque

pas notre application. Ainsi, nous avons créé un thread séparé pour le calcul des prochaines générations. Le thread principal peut donc continuer à traiter d'autres opérations simultanément.

## 4.6 Les autres fonctionnalités

Nous avons aussi implémenté des fonctionnalités connexes à la simulation. Ainsi, pour **arrêter la simulation**, il suffit de stopper le thread de calcul des générations. Et pour la **reprendre**, il suffit de le relancer. Ce thread marque une pause de 100 millisecondes par défaut entre chaque génération. Pour **modifier la vitesse de génération**, nous modifions donc la variable qui contient ce temps de pause. Quand elle augmente, la simulation est moins rapide et dans le cas contraire elle est plus rapide. À chaque génération, nous sauvegardons la génération actuelle dans une variable (qui est écrasée à chaque génération). De ce fait, pour revenir à la **génération précédente**, il suffit de lire dans cette variable. Pour **recommencer**, nous réinitialisons juste la grille.

# 5 Expérimentations et Usages

## 5.1 Lancement de l'application

Vous devez ouvrir un terminal à l'emplacement du dossier.

Pour :

- Lancer l'application sur le terminal : exécuter `ant runConsol`
- Lancer l'application avec l'interface graphique: exécuter `ant runInterface`
- Initialiser le projet : exécuter `ant init` . Cette initialisation créera les dossiers de base bin, doc et dist.
- Compiler le projet : exécuter `ant compile`
- Générer la Javadoc : exécuter `ant javadoc`
- Générer le fichier jar : exécuter `ant packaging`
- Nettoyer le projet : exécuter `ant clean`
- Lancer les tests : exécuter `ant test`

## 5.2 Implémentation de l'Interface graphique

Java Swing est l'interface que nous avons choisie pour faire l'implémentation graphique de notre simulateur. Nous y avons défini 5 classes dans le package `views` et une classe `Aide` dans le package `util`.

`Views` contient les classes suivantes :

1. **Config** : permet de configurer le jeu en définissant les règles et le type de voisinage utilisé, et de voir l'évolution de la simulation (cellule vivante, nombre de générations, ainsi que la vitesse de l'algorithme choisi). Elle contient également un bouton qui permet d'initialiser la grille avec une configuration aléatoire.
2. **GridGraphique** : représente la grille graphique. Elle dispose d'une fonction qui permet de cliquer sur la grille pour changer l'état de la cellule.
3. **Rendu** : cette classe est divisée en 3 parties : une pour contenir les boutons de navigation, une pour la grille, et une dernière permettant à l'utilisateur de choisir l'algorithme utilisé pour la simulation.
4. **Menu** : cette classe permet à l'utilisateur de choisir dans une liste de motifs prédéfinis et un bouton d'aide qui explique l'utilisation correcte de l'application.
5. **MainWindow** : est la classe principale qui compose l'ensemble des composants de l'interface graphique. Elle met en liaison la partie interface graphique et le modèle du simulateur (**game**).

Util contient la classe suivante :

- **Aide** : cette classe importe un fichier texte expliquant le bon fonctionnement du simulateur.

### 5.3 Fonctionnement de l'interface graphique

Au lancement du programme, une fenêtre s'affiche, comme illustré dans la figure ci-dessous. Cette fenêtre permet aux utilisateurs de commencer une nouvelle partie. Les utilisateurs ont le choix entre deux boutons pour choisir entre le jeu classique et le jeu avec Hashlife.

La fenêtre comprend également les éléments suivants :

- Les boutons de contrôle, qui permettent de démarrer/arrêter(play/stop) la simulation, avancer/reculer(next/back) d'une génération, accélérer/ralentir(rapide/reculer) la simulation et restaurer(reset) la grille. Le bouton "Aide" redirige vers une fenêtre d'aide pour une utilisation optimale de l'application
- La liste de patterns disponibles(patterns), qui permet aux utilisateurs de choisir un modèle prédéfini.
- Les boutons de règles(rules), qui permettent aux utilisateurs de choisir la règle utilisée pour la simulation (Game of Life par défaut).
- Les boutons de type de voisins, qui permettent aux utilisateurs de choisir le type de voisinage utilisé (game of life par défaut).
- Le contrôle de la vitesse d'exécution, qui permet aux utilisateurs de visualiser la vitesse à laquelle la simulation avance selon l'algorithme utilisé.



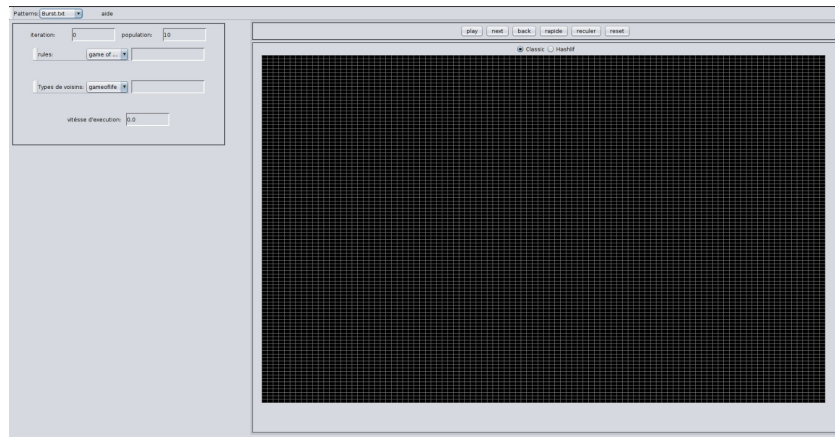


Figure 4: Lancement du jeu

- initialisation de la grille : pour initialiser la grille il est possible de cliquer sur le bouton "Init random" pour initialiser la grille avec des cellules aléatoires, ou sélectionner un pattern existant dans la liste, ou encore cliquer directement sur les cellules de la grille pour les activer/désactiver



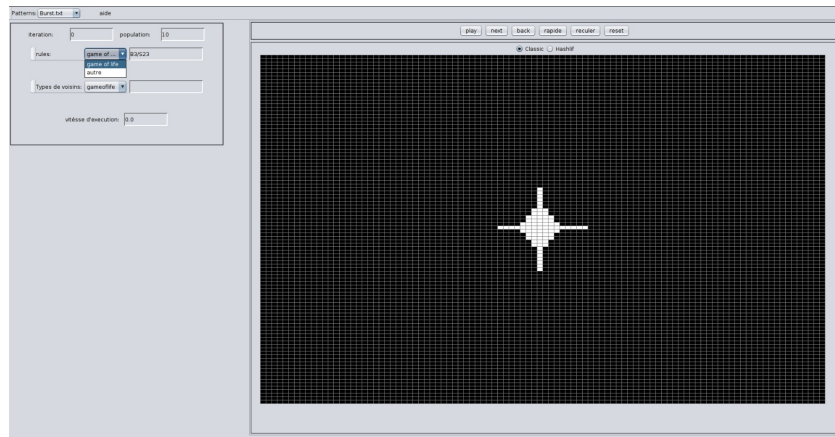


Figure 7: Illustration du bouton "rules"

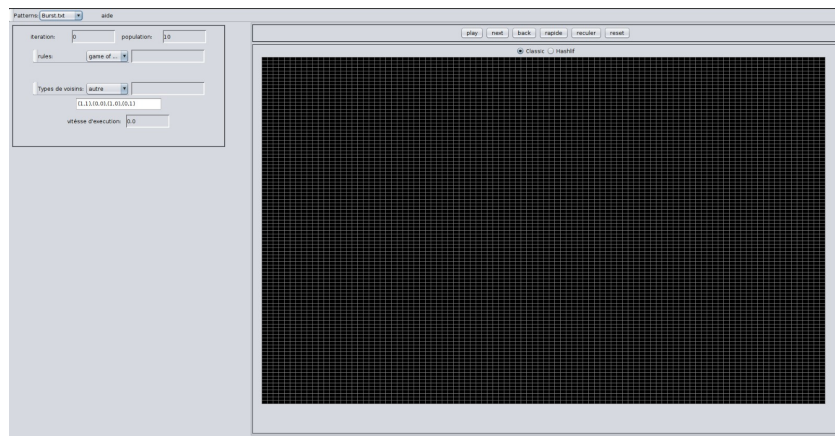
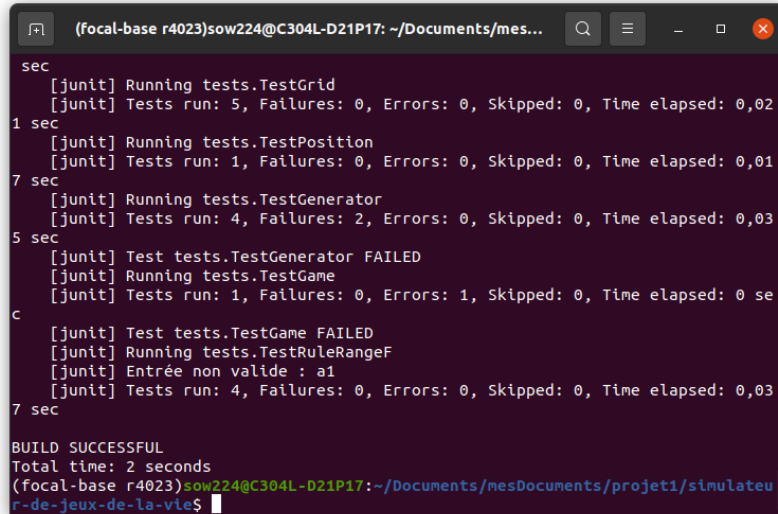


Figure 8: Illustration du bouton "type de voisins"

## 5.4 Test du Logiciel

Nous avons utilisé le framework open source JUnit 4.12 pour réaliser nos tests unitaires. Ces tests portent sur l'ensemble des méthodes essentielles pour le bon fonctionnement de notre projet



```
(focal-base r4023)sow224@C304L-D21P17: ~/Documents/mes...
sec
[junit] Running tests.TestGrid
[junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0,02
1 sec
[junit] Running tests.TestPosition
[junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0,01
7 sec
[junit] Running tests.TestGenerator
[junit] Tests run: 4, Failures: 2, Errors: 0, Skipped: 0, Time elapsed: 0,03
5 sec
[junit] Test tests.TestGenerator FAILED
[junit] Running tests.TestGame
[junit] Tests run: 1, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 0 se
c
[junit] Test tests.TestGame FAILED
[junit] Running tests.TestRuleRangeF
[junit] Entrée non valide : a1
[junit] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0,03
7 sec

BUILD SUCCESSFUL
Total time: 2 seconds
(focal-base r4023)sow224@C304L-D21P17:~/Documents/mesDocuments/projet1/simulateu
r-de-jeux-de-la-vie$
```

Figure 9: test”

## 6 Conclusions

### 6.1 difficultés rencontrés

La plus grande difficulté que nous avons rencontrée dans le développement de notre jeu de la vie en Java était la gestion de l'état des cellules en fonction du nombre de voisins. Pour un jeu classique sans hashlife, nous avons considéré que toutes les cellules sur les bords ne changeraient pas d'état car elles n'ont pas huit voisins. La deuxième difficulté que nous avons rencontrée concernait l'implémentation de hashlife.

Le premier problème que nous avons identifié avec hashlife est que lorsque nous ajoutons des bordures au jeu, il arrive que ces bordures prennent vie en raison des règles du jeu. Pour remédier à cela, nous avons introduit un troisième état possible pour les cellules bordant le jeu : -1. Cet état est réservé aux cellules mortes qui ne devraient jamais prendre vie. De cette manière, nous nous assurons que le calcul des générations suivantes reste correct.

Le deuxième problème que nous avons rencontré avec hashlife est que plus le nombre de cellules dans le jeu est grand, plus le nombre de motifs possibles augmente, ce qui augmente le risque de collision de hachage. Pour détecter ces collisions de hachage, nous utilisons un hashmap compartimenté qui compare les caractéristiques. Cependant, cela prend du temps et ralentit considérablement l'algorithme hashlife.

## 6.2 Bilan du projet

En conclusion, le développement du jeu de la vie en Java a été une expérience très enrichissante pour notre équipe. Nous avons pu mettre en pratique nos connaissances en programmation orientée objet, en architecture MVC, en gestion de projet et en collaboration en équipe. Nous avons également développé notre créativité et notre capacité à résoudre des problèmes complexes de manière efficace.

De plus, ce projet nous a permis de renforcer nos compétences en communication, en coordination et en travail d'équipe. Nous avons appris à travailler ensemble de manière efficace pour atteindre nos objectifs communs, ce qui a été une expérience très formatrice pour nous tous.

## 6.3 Améliorations Possibles

Il existe plusieurs possibilités pour améliorer l'expérience utilisateur de notre application.

Tout d'abord, une amélioration intéressante serait de permettre à l'utilisateur de sélectionner les voisins d'une cellule directement à partir de la grille, plutôt que de devoir choisir un type de voisinage à partir d'un bouton. Cette modification faciliterait la sélection des voisins et améliorerait l'ergonomie de l'application.

De plus, pour améliorer la visualisation de la grille, il serait utile d'ajouter une fonction de zoom/dezoom. Ainsi, l'utilisateur pourrait afficher la grille à différentes échelles en fonction de ses besoins. Il serait également judicieux de donner la possibilité à l'utilisateur de définir une taille personnalisée pour la grille, ce qui permettrait une plus grande flexibilité dans la création de modèles personnalisés.

Il serait aussi intéressant de permettre à l'utilisateur de choisir le nombre de générations à sauter. Pour l'instant, nous avons seulement codé une fonction pour effectuer cette tâche, mais nous n'avons pas eu le temps de l'implémenter sur l'interface graphique.

Ces améliorations permettraient de rendre notre application plus conviviale et plus personnalisable pour l'utilisateur.

# Bibliographie

- [1] /dev/.mind/ hashlife. <https://www.dev-mind.blog/hashlife/>.
- [2] Golly game of life. <https://golly.sourceforge.net/webapp/golly.html>.
- [3] Mathmatique jeu de la vie. <http://pdbzro.com/gags/math/jeudelavie.pdf>.
- [4] WIKIPEDIA hashlife. <https://en.wikipedia.org/wiki/Hashlife>.
- [5] WIKIPEDIA jeu de la vie. [https://fr.wikipedia.org/wiki/Jeu\\_de\\_la\\_vie](https://fr.wikipedia.org/wiki/Jeu_de_la_vie).