Domaine Sciences et Technologies
Master Bio-informatique : DLAD

Aix*Marseille université
Socialement engagée

Année 2023-24

Programmation avancée : TP 4
Code UE : SBICU05

Jeu de la vie

Jeu de la vie

Introduction

Le jeu de la vie n'est pas vraiment un jeu, puisqu'il ne nécessite aucun joueur.

Le jeu se déroule sur une grille à deux dimensions dont les cases qu'on appelle des cellules peuvent prendre deux états distincts : vivantes ou mortes.

À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante :

— Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).
— Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

Le but de ce TP est de compléter le code fourni par le dépôt afin d'obtenir un simulateur de jeu de la vie.

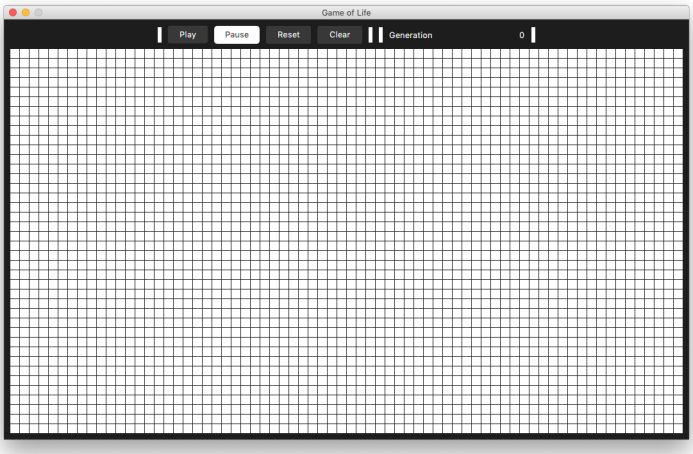
Consignes

Comme pour le TP précédent et celui d'avant, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du TP 2.

Lien vers le projet gitlab à forker pour le TP : lien.

Projet

Une fois le dépôt téléchargé, vous pouvez compiler et exécuter le code cliquer deux fois sur `game-of-life` -> `application` -> `run`. Vous devriez obtenir l'affichage suivant.



Pour exécuter les tests, il faut passer par l'onglet gradle à droite et cliquer deux fois sur 'game-of-life -> Tasks -> verification -> test. Pour le moment, les tests ne passeront pas car certaines classes sont incomplètes.

Malheureusement, l'application ne fonctionne pas correctement. Vous pouvez changer l'état de chaque cellule en cliquant dessus (Une cellule morte est blanche alors qu'une cellule vivante est rouge). Par contre le lancement de la simulation via le bouton *play* ne crée aucun changement dans la grille.

Pour que l'application fonctionne, vous devez compléter le code de la classe `model.Grid`. Cette classe sert à modéliser la grille des cellules. Elle est composé d'une matrice de cellules (objet de type `model.Cell`). Chaque cellule contient elle-même un état qui est représenté par une énumération de type `CellState` pouvant être égal à `CellState.DEAD` (état d'une cellule morte) ou `CellState.ALIVE` (état d'une cellule vivante).

Pour compléter la classe `Grid`, vous aurez besoin d'utiliser les trois méthodes suivantes de la classe `Cell` :

- `public boolean isAlive()` : retourne `true` si la cellule est vivante et `false` sinon.
- `public setState(CellState cellState)` : fixe l'état de la cellule.
- `public CellState getState()` : retourne l'état de la cellule.

La classe `Grid` contient déjà les méthodes suivantes (que vous n'avez pas à modifier) :

- `public Iterator<Cell> iterator()` : permet de parcourir les cellules d'un grille `grid` à l'aide d'une boucle `for(Cell cell : grid)`.
- `public Cell getCell(int rowIndex, int columnIndex)` : retourne la cellule à la position (`rowIndex`, `columnIndex`). Afin de vous simplifier le travail, la méthode `public Cell getCell(int rowIndex, int columnIndex)` autorise des indices en dehors des bornes habituelles (avec des valeurs négative ou supérieur au nombre de ligne/colonnes) en considérant que la grille est enroulée sur elle-même. Elle renvoie donc la cellule en position (0,0) si on lui donne en paramètre (`numberOfRows`,`numberOfColumns`) et la cellule en position (`numberOfRows-1`,`numberOfColumns-1`) si on lui donne en paramètre (-1,-1).
- `public int getNumberOfRows()` : retourne le nombre de lignes de la grille.
- `public getNumberOfColumns()` : retourne le nombre de colonnes de la grille.

Méthode `getNeighbors`

La première méthode que vous devez coder est `List<Cell> getNeighbors(int rowIndex, int columnIndex )` qui doit retourner la liste des 8 cellules voisines à la cellule (cellules partageant un coin ou un coté avec la cellule) à la position (`rowIndex`, `columnIndex`). Pour que toutes les cellules aient 8 voisines, on considère que les bords en haut et en bas sont voisins et aussi que les bord à droite et à gauche sont aussi voisins.

Vous pouvez tester le code en cliquant deux fois sur `game-of-life` -> `test` -> `run`

Méthode `countAliveNeighbors`

La deuxième méthode que vous devez coder est `public int countAliveNeighbors(int rowIndex, int columnIndex)`. Elle doit retourner le nombre de cellules vivantes parmi les 8 voisines de la cellule. Vous devez appeler la méthode `getNeighbors` pour le code de cette méthode.

Méthode `calculateNextState`

La troisième méthode que vous devez coder est `public CellState calculateNextState(int rowIndex, int columnIndex)` qui renvoie l'état que doit prendre la cellule en position (`rowIndex`, `columnIndex`) à la prochaine étape. On rappelle que cet état est déterminé par les règles suivantes :

— Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).
— Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

Méthode `calculateNextStates`

La quatrième méthode que vous devez coder est `public CellState[][] calculateNextStates()` qui renvoie une matrice correspondant aux états que doivent prendre les cellules de la grille à la prochaine étape.

Méthode `updateStates`

La cinquième méthode que vous devez coder est `public void updateStates(CellState[][] nextState)` qui met à jour tous les états des cellules de la grille en utilisant les valeurs de la matrice donnée en argument.

Méthode `updateToNextGeneration`

La sixième méthode que vous devez coder est `public void updateToNextGeneration()` qui fait avancer la grille d'une étape et donc met à jour tous les états des cellules de la grille en fonction des règles du jeu de la vie.

Méthode `clear`

La septième méthode que vous devez coder est `public void clear()` qui met à jour tous les états des cellules de la grille à `CellState.DEAD`.

Méthode `randomGeneration`

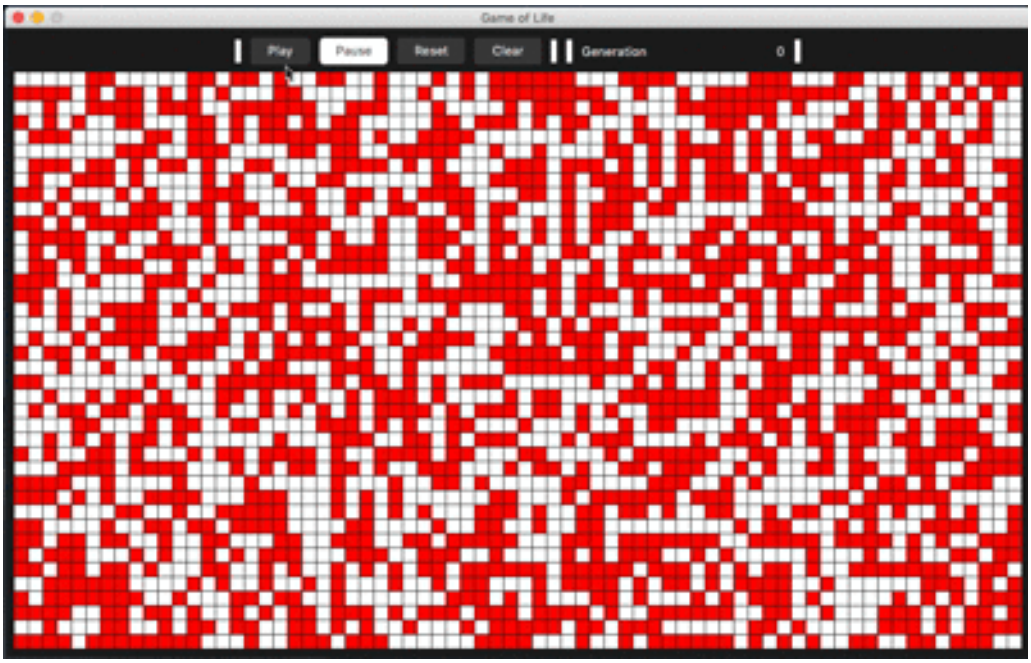
La huitième méthode que vous devez coder est `public void randomGeneration(Random random)` qui met à jour tous les états des cellules de la grille de manière aléatoire à `CellState.DEAD` ou `CellState.ALIVE`. On

utilisera pour cela la méthode `nextBoolean` avec l'objet `random` passé en argument.

Test de vos méthodes

Vous pouvez tester vos méthodes, vous pouvez lancez les tests en cliquant deux fois sur `game-of-life` -> `Tasks` -> `verification` -> `test`. Cela lancera des tests de bases pour la classe `Grid`. Vous pouvez écrire vos propres tests en suivant les consignes du précédent TP.

Si vous avez tout codé correctement, vous allez obtenir un affichage comme suivant :



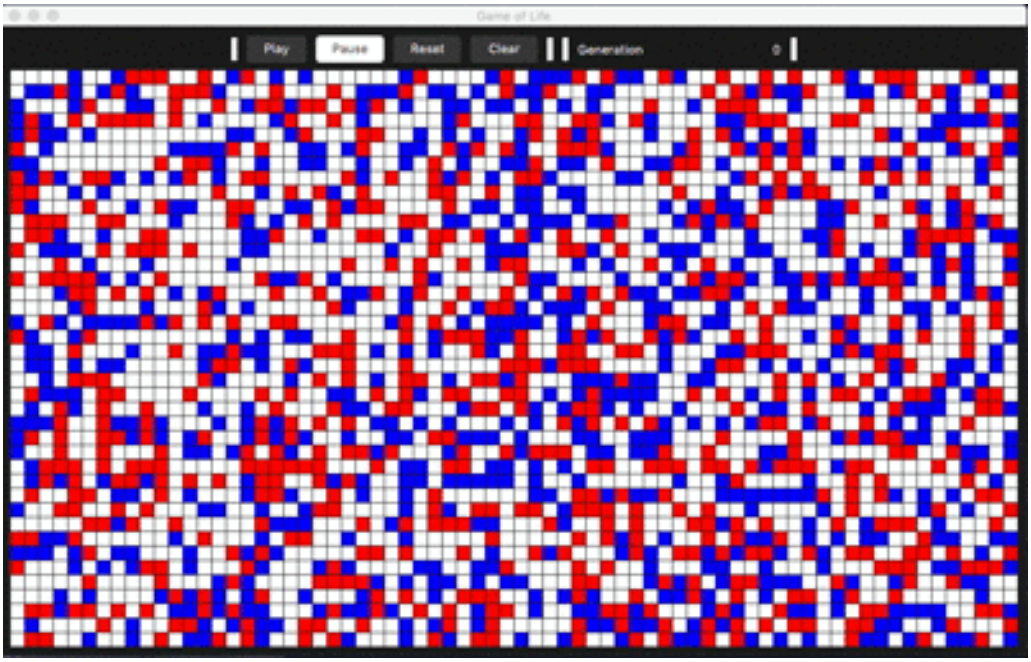
Variante colorée

On souhaite maintenant avoir deux états possibles pour une cellule vivante : `BLUE` ou `RED` en plus de l'état `DEAD` pour une cellule morte. Une cellule `BLUE` devra être affiché en bleue alors qu'une cellule `RED` devra être affichée en rouge.

- Une cellule morte possédant exactement trois voisines vivantes devient vivante et naît avec la couleur majoritaire de ses voisines.
- Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt. Une cellule survivante garde sa couleur.

Pour la génération aléatoire, on gardera la probabilité d'avoir un cellule morte à 1/2. La probabilité d'avoir une cellule `RED` devra être de 1/4 et la probabilité d'avoir une cellule `BLUE` devra aussi être de 1/4.

Si vous avez tout codé correctement, vous allez obtenir un affichage comme suivant :



Tâches optionnelles

— Écrivez des tests pour toutes les méthodes non-testées.
— Augmentez le nombre de couleurs possible des cellules à 3, 4 ou plus.