Projet de programmation orientée objet : Automates Cellulaires

Dans ce projet, vous mettrez en place un programme permettant d'implémenter un automate cellulaire. Pour faire simple, un automate cellulaire est un système composé d'une grille régulière composée de cases (appelées cellules), qui peuvent être dans plusieurs états (en général, deux états : allumé ou éteint). Une cellule peut changer d'état selon certaines règles prédéfinies, et on regarde avec le temps comment les cases changent d'état à chaque tour. Certaines règles peuvent donner lieu à des comportements étranges, comme le "jeu de la vie" de John Horton Conway (je vous invite fortement à regarder `https://www.youtube.com/watch?v=S-W0NX97DB0`).

1 Les automates cellulaires

Un *automate cellulaire* est un système dynamique évoluant en fonction d'un temps discret il est composé:

- d'une grille régulière de cellules
- d'un ensemble d'état
- une affectation d'états au cellules de la grilles appelé *état initial*
- d'une règle permettant à chaque cellule de changer d'état en fonction de l'état de ses voisins à chaque pas de temps.

L'ensemble des cellules d'un automate cellulaire évolue simultanément. La règle permet de faire évoluer le système à partir de modifications locales. Nous allons voir dans la prochaine section deux exemples d'automates cellulaires très simples dont le premier a un comportement prévisible, alors que le second a un comportement "bizarre".

1.1 Les automates cellulaires de dimension 1

L'exemple le plus simple d'automate cellulaire est composé d'une grille de dimension 1 contenant des cellules appartenant à deux états disons bleu ou blanc, et une règle prenant en compte l'état du voisin de gauche de la case courante, de la case courante et du voisin de droite de la case courante. La règle doit donc

associer à chaque configuration possible de ces trois cases, le nouvel état de la case courante. Pour définir une règle de ce type, il faut donc associer aux 2^3 possibilité pour le voisinage, un des deux états.

Prenons l'exemple de la Figure 2 qui juxtapose les états successifs obtenus en parant d'un état initial composé d'une seule case bleu et pour lequel, à chaque pas de temps t_i, la règle décrite en Figure 1 est appliquée. L'état du voisinage est à gauche de la flèche et l'évolution de l'état la case courante est à sa droite. En encodant les états par des bits, disons 1 pour bleu et 0 pour blanc, on pourrait encoder la règle de la Figure 1 dans la formule suivante:

$$c_i^{t+1} = \max(c_{i-1}^t, c_i^t, c_{i+1}^t)$$

où c_j^k est la valeur du bit associé à l'état de la case d'indice j à l'état k.

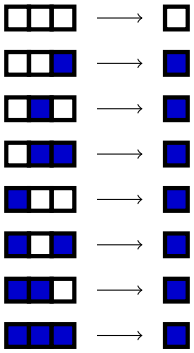

Figure 1: La règle d'un premier automate cellulaire très simple, où une case "s'allume" (devient bleue) si une de ses deux voisines ou elle-même est allumée.

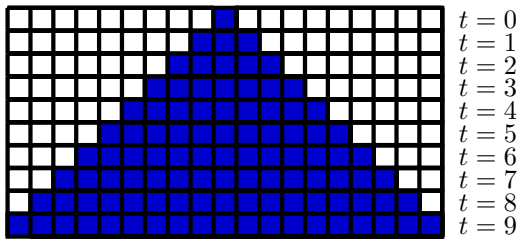

Figure 2: Juxtaposition des états d'un premier automate cellulaire

La juxtaposition des états de notre premier automate cellulaire donne une figure extrêmement simple (qu'il aurait été possible de la prévoir juste en regardant la règle d'évolution du système). Cependant, ce n'est pas toujours le cas comme le montre la Figure 3 (trouvée sur le web) qui est obtenue avec la règle décrite en Figure 4 en partant du même état initial. La figure fractale obtenue

en juxtaposant les états est appelée le triangle de Sierpiński. Notons que cette règle peut être encodée dans la formule

$$c_i^{t+1} = (\max(c_{i-1}^t, c_{i+1}^t) - \min(c_{i-1}^t, c_{i+1}^t)) * (1 - c_i^t)$$

.

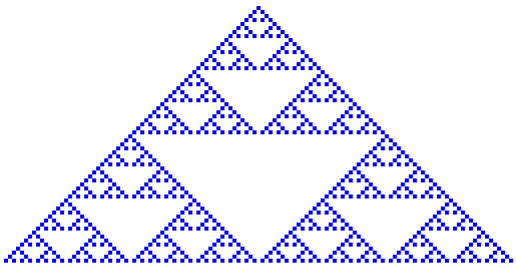

Figure 3: Le triangle de Sierpiński, obtenu en juxtaposant verticalement les différents état d'un système 1D évoluant selon la formule donnée précédemment.

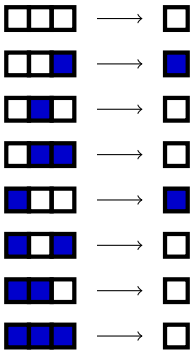

Figure 4: La règle permettant d'obtenir le triangle de Sierpiński

Les automates cellulaires sont des systèmes dynamiques définis à l'aide de règle locales très simples, mais qui peuvent avoir des comportements très surprenants. En fait, les automates cellulaires permettent de simuler des machines de turing (version théorique de l'ordinateur).

1.2 Un automate cellulaire de dimensions 2

Un automate cellulaire de dimension 2 très connu s'appelle le jeu de la vie. Il est composé d'une grille de dimension 2, chaque case de la grille est appelée

une cellule et peut être dans un état parmi deux possibles : vivante (état 1) ou morte (état 0). On peut initialiser la grille avec un ensemble de cellules vivantes aléatoires et on applique la règle d'évolution suivante:

- une cellule morte devient vivante si exactement trois de ses voisines sont vivantes, sinon, elle reste morte.
- une cellule vivante possédant deux ou trois voisines vivantes reste vivante, sinon, elle meurt.

Notons que si l'état initial du jeu de la vie ne comporte aucune cellule vivante, alors la grille reste entièrement morte à chaque pas de temps. Lorsque la grille est identique entre deux pas de temps, on dit que le jeu a convergé. Dans certains cas d'initialisation il lui faut un nombre non nul d'itération avant convergence et parfois même il n'y a pas de convergence mais des motifs périodiques qui se répètent.

Un nombre important de travaux de recherche portent sur les automates cellulaires de dimension 2 mais aussi de dimension quelconque

Pour plus d'information sur le sujet, nous vous conseillons les vidéos de la chaîne youtube scienceEtonnante sur le jeu de la vie (`https://www.youtube.com/watch?v=S-W0NX97DB0`), la fourmi de Langton et d'autres...

2 Le projet

Le but de ce projet est de proposer une architecture et son implémentation en Java afin d'être capable de simuler n'importe quel automate donné en entrée du programme. Pour ce faire il sera nécessaire d'implémenter plusieurs structures de données suffisamment génériques et de les faire interagir. Au moins 3 entités distinctes se retrouvent dans ce projet :

- Un automate cellulaire
- Une grille
- Des règles d'évolution des cellules

Il est clair qu'un automate cellulaire est composé d'une grille d-dimensionnelle et d'une ou d'un ensemble de règles. Vous pouvez aussi proposer d'autres entités si nécessaire à condition de pouvoir justifier de leur importance dans la conception que vous proposerez.

2.1 Implémenter une grille d-dimensionnelle (3 jours)

En java, même si la taille d'un tableau peut être initialisée dynamiquement, sa dimension (int [][] tab, ici tab est de dimension 2) est fixée à l'implémentation. Pour pouvoir générer des grilles de dimension d où d est donnée en entrée du programme, vous devrez implémenter votre propre structure de donnée. Pour cela, il sera nécessaire de proposer une classe **TableauDynamiqueND** (vous

serez libres de choisir le noms qui vous conviendra) dont la définition sera récursive : un tableau dynamique n-dimensionnel de taille $k_1 \times k_2 \times ... \times k_n$ réalisera l'allocation de k_1 tableaux dynamiques $(n-1)$-dimensionnels de tailles $k_2 \times ... \times k_n$ (vous devrez peut-être vous renseigner sur les méthodes à nombre variable de paramètres). Vous pourrez également proposer une classe **Cellule** qui représentera les cellules de la grille.

Les automates cellulaires sont étudiés dans le cadre où la grille est de taille infinie, le support physique qu'est l'ordinateur nous obligera à donner une taille maximale. Toute cellule "en-dehors" de la grille sera considérée comme toujours éteinte si sa valeur doit être prise en compte dans une règle.

2.2 Le voisinage d'une cellule (3 jours)

Dans certaines règles d'évolution des cellules, pour calculer la nouvelle valeur d'une cellule, il est nécessaire d'explorer certaines cellules situées autour de la cellule cible. Ces cellules composent ce que l'on appelle le voisinage de la cellule, et il existe différents voisinages.

On souhaitera pouvoir définir, dans le fichier de configuration, nos propres voisinages, mais certains voisinages considérés basiques seront définis par avance dans votre code. Dans tous les cas (afin de simplifier votre programme), un voisinage sera toujours nommé par un G majuscule suivi d'un numéro et éventuellement d'une étoile. Voici les voisinages que nous souhaiterons définir de base dans le code :

- G0 : représente la cellule examinée seule.
- G2 : valable en 1D, ce voisinage représente la cellule examinée, ainsi que sa voisine de droite et sa voisine de gauche.
- G4 : valable en 2D, ce voisinage représente la cellule examinée, ainsi que sa voisine du dessus, du dessous, de droite et de gauche.
- G8 : valable en 2D, ce voisinage représente G4, ainsi que la voisine en haut à droite, en haut à gauche, en bas à droite et en bas à gauche.
- G6 : valable en 3D, ce voisinage représente la cellule examinée ainsi que sa voisine du dessus, du dessous, de droite, de gauche, de devant et de derrière.
- G26 : valable en 3D, ce voisinage représente la cellule examinée ainsi que toutes se voisines qui touchent une face, une arête ou un coin de la cellule 3D.

Nous définirons aussi les Gk*, avec $k \in \{2, 4, 8, 6, 26\}$, qui sont égaux au voisinage Gk mais où la cellule examinée n'est pas prise en compte.

Afin de représenter un voisinage, il vous est possible (mais pas obligatoire) de regarder les interface *Iterator* et *Iterable* de Java. Dans votre implémentation

des classes de voisinage, il sera important de montrer qu'un voisinage n'est rien d'autre qu'une règle de parcours de cellules voisines d'une certaine cellule.

Il sera possible, dans le fichier de configuration, de définir ses propres voisinages. Pour ce faire, il suffira de lister les coordonnées des voisins d'une cellule si cette dernière a pour coordonnées l'origine. Ainsi, le voisinage G4* pourrait s'écrire :

$$G4* = \{(-1,0),(1,0),(0,-1),(0,1)\}$$

Si on souhaitait définir un voisinage G3 en 3d, avec la cellule examinée en elle-même, sa voisine en haut à droite et sa voisine du dessous, on écrirait :

$$G3 = \{(0,0,0),(1,1,0),(-1,0,0)\}$$

.

2.3 Implémentation des règles (4 jours)

La règle régissant un automate cellulaire peut se décomposer en opérateurs que vous devrez implémenter. Chaque opérateur renvoie une valeur numérique ; certains opérateurs prendront en paramètre des valeurs, qui pourront être des constantes numériques ou le résultat d'autres opérateurs, et d'autres opérateurs prendront en paramètre un voisinage.

- ET(val1, val2) : renverra 1 si val1 et val2 sont tous deux différents de 0, et 0 sinon.
- OU(val1, val2) : renverra 0 si val1 et val2 sont tous deux égaux à 0, et 1 sinon.
- NON(val) : renverra 1 si val vaut 0, et 0 sinon.
- SUP(val1, val2) : renverra 1 si val1 est strictement supérieure à val2, et 0 sinon.
- SUPEQ(val1, val2) : renverra 1 si val1 est supérieure ou égale à val2, et 0 sinon.
- EQ(val1, val2) : renverra 1 si val1 est égal à val2, et 0 sinon.
- COMPTER(voisinage) : renverra le nombre de cellules à l'état 1 dans le voisinage de la cellule observée.
- ADD(val1, val2) : renverra la somme de val1 et val2
- SUB(val1, val2) : renverra la soustraction de val1 et val2
- MUL(val1, val2) : renverra le produit de val1 et val2
- SI(val1, val2, val3) : si val1 est différent de 0, renverra val2, et sinon, renverra val3

Point important : dans la liste précédente, les val peuvent être eux-mêmes des résultats d'opérateurs, ou directement des valeurs numériques.

Grâce à ces opérateurs et aux voisinages définis précédemment (ou aux voisinages personnels), nous pourrons implémenter différentes règles avancée. Par exemple, imaginons la règle suivante : *Si une cellule est vivante et qu'elle a 4 voisines vivantes ou plus dans son 8-voisinage, alors elle meurt, sinon, elle reste vivante. Si une cellule est morte et qu'elle a exactement deux voisines vivantes dans son 8-voisinage, alors elle devient vivante, sinon, elle reste morte.*

Décomposons cette règle en sous-morceaux afin de l'exprimer avec nos opérateurs. Tout d'abord, pour savoir si une cellule est vivante, on comptera simplement le nombre de cellules vivantes dans son voisinage G0 :

$$EQ(COMPTER(G0), 1)$$

Ensuite, pour dire qu'une cellule qui a 4 voisines vivantes ou plus dans son 8-voisinage meurt, et sinon, elle reste vivante, on pourra écrire :

$$SI(SUPEQ(COMPTER(G8), 4), 0, 1)$$

Enfin, pour dire qu'une cellule qui a deux voisines vivantes dans son 8-voisinage devient vivante, on pourra écrire :

$$SI(EQ(COMPTER(G8), 2), 1, 0)$$

La formule complète sera :

$$\begin{aligned} &SI(EQ(COMPTER(G0), 1),\\ &\quad SI(SUPEQ(COMPTER(G8), 4), 0, 1),\\ &\quad SI(EQ(COMPTER(G8), 2), 1, 0)) \end{aligned}$$

Il est fortement conseillé de bien réfléchir à la hiérarchie des classes qui sera utilisée afin de représenter les opérateurs en Java.

2.4 Interface graphique (2 jours)

Un exemple de code source joint au projet vous propose de dessiner une grille bidimensionnelle de cases, et de colorier certaines des cases en bleu. Vous pourrez librement vous inspirer et réutiliser ce code afin de proposer une visualisation de votre automate cellulaire (et choisir vous même quelle couleur correspond à quel état)

Dans le cas d'un automate 1d ou 2d, l'automate devra être affiché à l'écran. Dans le cas d'un automate 3d ou plus, on souhaitera afficher une coupe 2d de cet automate, en spécifiant les coordonnées de la coupe que l'on souhaitera observer (voir plus de détails dans la section du fichier d'entrée).

2.5 Fichier d'entrée (1 semaine)

Un automate cellulaire sera décrit par un fichier d'entrée, dont le format précis est laissé à votre choix (le format XML est un choix à considérer, de nombreuses bibliothèques de lecture des fichiers XML existant en Java). Le fichier d'entrée devra permettre de saisir les informations suivantes de la scène :

- La dimensionnalité de la grille considérée (1d, 2d, 3d, etc).
- La taille de la grille considérée (k valeurs, où k sera égal à la dimensionnalité définie précédemment).
- Si la dimensionnalité de la grille est supérieure ou égal à 3, le plan que l'on souhaitera afficher sur l'interface graphique. Par exemple, imaginons que l'on ait une grille 3d, et on souhaite observer le plan de coordonnées $x = 37$. On pourra alors écrire ce plan comme $(37, :, :)$ (cela devrait vous rappeler la syntaxe de Matlab).
- Eventuellement, des voisinages supplémentaires utilisés dans la règle d'évolution.
- La règle qui régit l'évolution de la grille, suivant la syntaxe donnée précédemment.
- Les coordonnées des cellules initialement à l'état 1. Si l'utilisateur saisit ici le mot clef $RANDOM(k)$ (avec k un entier entre 0 et 100), alors la grille sera initialisée aléatoirement, chaque cellule ayant $k\%$ de chances d'être dans l'état 1 au début.

Votre programme devra être capable de lire et décrypter un fichier d'entrée, afin de proposer une simulation correspondant aux données décrites dans le fichier.

2.6 Consignes

Votre programme principal devra proposer un menu affichant tous les fichiers d'entrée disponibles dans le répertoire actuel, et proposer à l'utilisateur de lancer la simulation correspondant à l'un des fichiers. Vous devrez proposer au moins un fichier de configuration correspondant au jeu de la vie (en initialisant les cases initiales à RANDOM(15)), ainsi qu'un fichier de configuration correspondant au triangle de Sierpiński.

Il vous faudra rendre, à l'issue de ce projet, un fichier zip contenant votre code source et vos fichiers d'entrée, ainsi qu'un rapport en format PDF. Dans ce rapport, vous ferez apparaître le diagramme de classe correspondant à votre projet, et expliquerez vos choix d'implémentation, les extraits de code vous paraissant intéressants, et tout autre détail vous semblant important. N'oubliez pas de consulter le guide de rédaction des rapports.

2.6.1 Question bonus

Aidez vous de votre implémentation pour trouver un automate cellulaire de dimension 1 qui après juxtaposition des états successifs donne une figure fractale ou chaotique. La règle que vous utiliserez pour le trouver considérera le voisinage défini par les deux cases à gauche de la case courante et les deux cases de droite. La figure obtenue ne doit pas être obtenu avec un voisinage défini seulement par la case de gauche, la case courante et celle de droite. Attention, cette fois-ci l'énumération peut être couteuse, le nombre de telles règles est de l'ordre de 2^{2^5}. Vous pouvez proposer un état de départ différent de seulement une case bleue.