Vie Artificielle
Automates Cellulaires 1D

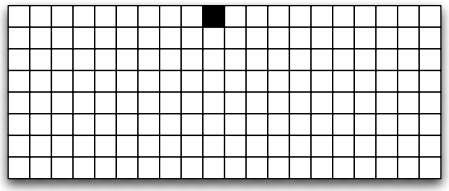
Mise à jour : Janvier 2019

Mots clés: *automates cellulaires, 1 dimension, règles de Wolfram, embouteillages.*

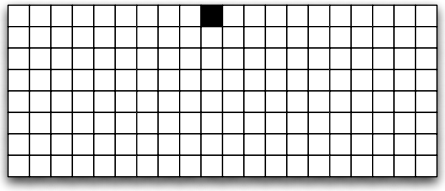
Une page est disponible sur le site http://pages.isir.upmc.fr/~bredeche/ - Vous y trouverez des informations diverses (lien vers les sources, les sujets, etc.). En particulier vous trouverez un lien vers les codes sources donnés en annexe.

Automates cellulaires 1D [sur feuille]

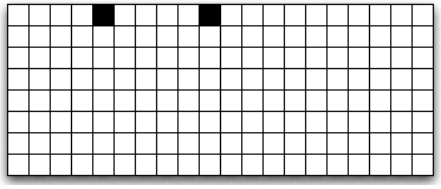
QUESTION 1. décrivez la règle 2, puis exécutez là sur 8 ligne avec la première ligne initialisée comme ci-dessous:



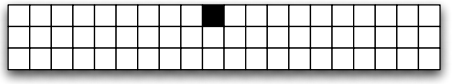
QUESTION 2. décrivez la règle 90, puis exécutez là sur 8 ligne avec la première ligne initialisée comme ci-dessous:



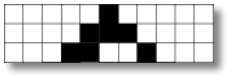
- Exécutez la règle 90 sur 8 lignes, avec la première ligne initialisée comme ci-dessous.



QUESTION 3. Décrivez trois règles permettant de produire le résultat ci-dessous, étant donné la première ligne initialisée comme ci-dessous. Donnez leurs numéros.



QUESTION 4. Donnez la (les) règle(s) qui peut (peuvent) donner le résultat ci-dessous?



Visualiser le déroulement d'automates cellulaires 1D [sur ordinateur]

On se propose maintenant d'écrire un programme (en JAVA) capable d'afficher le déroulement d'un automate cellulaire sur l'écran de votre ordinateur.

Vous récupérerez à partir du site web le code source *CAtoolbox.java* qui contient deux fonctions utiles:

- ***print(chaine-de-caractere)*** : affiche une chaîne de caractère à l'écran.
- ***intialiseRegle(numéro-de-la-règle)*** : renvoi un tableau contenant les conditions et résultats d'une règle

Cette seconde méthode vous permet de récupérer sous forme de tableau directement exploitable la règle de mise à jour de vos automates. Par exemple: ***intialiseRegle(90)*** renvoi un tableau permettant d'utiliser la règle 90. Il s'agit d'un tableau à double entrée, de dimensions 8x4 (les 8 sous-règles et, pour chaque sous règles, 3 cases pour stocker l'états possibles de parents, et 1 case pour stocker l'état de l'enfant).

Vous pouvez vous reporter à la fin du sujet pour un squelette de programme.

QUESTION 1. le générateur de règles

Récupérez la règle 90 à l'aide de la fonction précédemment citée. Pour ce faire, vous définirez un tableau à double entrée contenant des booléens comme suite:

boolean[][] regles = initialiseRegle(90);

Par exemple, l'affichage de la sous-regle 0 se fait en parcourant le tableau comme suit:

```
boolean[][] regles = initialiseRegles(90);
for ( int i = 0; i != 3 ; i++ )
     print(regles[0][i] + " ");
print(" -> " + regles[0][3]);
```

Les 3 premières cases contiennent l'état attendu des cellules juste au dessus, la 4ème l'état de la nouvelle cellule si la condition est satisfaite.

On vous demande d'afficher tous les cas de la règle 90.

QUESTION 2. mise à jour d'un automate cellulaire

On vous demande maintenant d'afficher le déroulement d'un automate cellulaire. L'état d'un automate cellulaire peut être simplement stocké dans un tableau de taille N (N étant le nombre de cellules). Ceci s'effectue en deux étapes:

1. Affichage de la ligne courante (vous mettrez un "." lorsque la cellule est inactive et un "X" lorsqu'elle est active - pour revenir à la ligne: *print("\n");*);
2. Mise à jour de la ligne pour la prochaine étape. Pour l'instant, vous vous contenterez de recopier à l'identique les états des cellules de la ligne précédente.

Une fois terminée, on recommence pour faire la ligne suivante, et ainsi de suite... (vous limiterez néanmoins le déroulement de l'automate cellulaire à une centaine d'étapes).

Attention, lors de la mise à jour d'une ligne, vous devrez utiliser un second tableau dans lequel vous écrirez le nouvel état de chaque cellule. Une fois l'*ensemble* des cellules mises à jour, vous pourrez écraser le contenu du tableau initial avec les nouveaux états;

QUESTION 3. application des règles

Vous devez maintenant ajouter l'application des règles. Pour cela, lors de la mise à jour de chaque cellule, vous cherchez parmi les sous-règles construites à la question 1 laquelle s'applique à la situation courante, et vous initialiserez la cellule à l'état correspondant pour le prochain pas de temps.

Vous avez maintenant une automate cellulaire fonctionnel.

QUESTION 4. validation

Observez le fonctionnement de votre programme pour les règles 2 et 90 vues dans le premier exercice.

QUESTION 5.

- Tentez de classifier les différents automates que vous pouvez générer (i.e. définissez des classes)
- Modifiez votre programme pour introduire une probabilité de mise à jour pour chaque cellule. C'est à dire que lors de l'écriture d'une nouvelle ligne, vous mettrez à jour une cellule courante avec une probabilité *p* et vous vous contenterez de la recopier avec une probabilité *1-p*. Une probabilité est de *p=1* correspond à une mise à jour sans faille, comme ce que vous avez codé auparavant. Etudiez ce qui se passe si vous modifiez cette probabilité (partant de p=1, vous diminuez peu à peu cette probabilité).

Modélisation d'une file d'automobiles [sur ordinateur]

On souhaite maintenant simuler les déplacements d'automobiles sur une seule file. On considère un tableau à une dimension (la route), sur laquelle on met des voitures (cases actives) ou non (cases inactives). Voici un exemple d'initialisation:

On souhaite simuler le déplacement selon la règle suivante:

une voiture avance avec une probabilité *p* si et seulement si la case à sa droite est libre.

On fait l'hypothèse que les déplacements sont synchronisés (ie. toutes les voitures essayent de se déplacer au même moment).

QUESTION 1. Simuler cet automate en modifiant le fichier TrafficJam1DCellularAutomata.java - On fait l'hypothèse que *p=1.0*, c'est à dire qu'une voiture se déplace toujours si elle le peut, et qu'une voiture disparaissant à droite réapparait à gauche (ie. les voitures se déplacent sur un circuit). Qu'observez-vous?

QUESTION 2. *p=1.0* est une hypothèse irréaliste (il est peut probable que les automobilistes se déplacent de manière synchronisée). Relancer votre simulation avec *p=0.5*, qu'observez vous?

Pour aller plus loin, vous pouvez tester plusieurs valeurs de *p*. Vous pouvez aussi essayer en fixant une valeur de probabilité de déplacement différent selon la voiture.

Automates Cellulaires 1D : classes, nombre d'états, influence de la mise à jour. [facultatif][sur ordinateur]

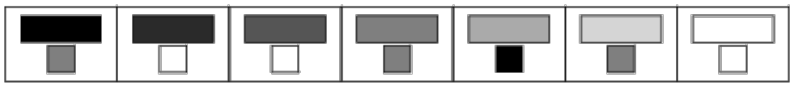
QUESTION 1. On vous demande d'implémenter un générateur de règle utilisant le paramètre λ introduit par C. Langton en 1991. Pour mémoire: λ donne la probabilité que parmi l'ensemble des configurations possible une cellule meurt. Soit, formellement:

λ = 1 - (nombre de configurations ou la cellule meurt / nombre de configurations possibles)

Ainsi, plus λ est proche de 1, plus les cellules ont de chances de rester ou devenir vivantes.

QUESTION 2. Faites plusieurs tests en prenant des valeurs de lambda allant de 0 à 1 en progressant par pas de 0,1. Tentez de vérifiez si la classification proposée par Langton a du sens (cf. cours).

On s'intéresse maintenant à l'automate cellulaire défini selon les règles suivantes:



QUESTION 1. Quelles sont les caractéristiques de cet automate cellulaire? (décrivez le)

QUESTION 2. A partir du programme que vous avez écrit lors du dernier TP, programmez cet automate cellulaire et exécutez le à partir d'un environnement ou seule une cellule est active (couleur noire).

QUESTION 3. Exécutez ce même automate cellulaire à partir d'une initialisation au hasard

QUESTION 4. Exécutez ce même automate cellulaire en ne mettant à jour que 50% des cellules à chaque pas de temps (ie. 50% des cellules sont des simples recopies). Le choix des cellules à mettre à jour est bien sur fait au hasard.

Pour vous aider, vous disposez d'un nouvelle fonction `update` *définie dans le fichier CAImageBuffer.java et qui prend en paramètre un tableau à deux dimensions contenant les valeurs entières des composées de couleurs rouge, vert et bleu -- valeurs entières entre 0 et 255. Par exemple: couleur noire (0,0,0) ; couleur blanche (255,255,255) ; couleur grise (128,128,128).*

Voir aussi: *http://mathworld.wolfram.com/TotalisticCellularAutomaton.html*


```
public class MonAutomateCellulaire extends CAtoolbox {

	public static void main(String[] args) {

		int tailleTableau = 100;

		boolean[] tableauCourant = new boolean[tailleTableau];
		boolean[] nouveauTableau = new boolean[tailleTableau];

		boolean[][] regles = initialiseRegles(90);

		// * initialisation

		// initialise les paramètres de simulation (ex. le nombre de pas de la simulation)

		int nombreDePasMaximum = 80;
		int it = 0;


		// * on fait tourner l'automate

		while ( it != nombreDePasMaximum )
		{
			for ( int i = 0 ; i != tableauCourant.length ; i++ )
			{
				// 1 - affiche etat courant

				//...affichage de la ligne contenu dans tableauCourant[]

				// 2 - met a jour l'automate cellulaire (pour la prochaine etape)

				//...mise à jour des états des cellules dans nouveauTableau[]
			}


			// 3 - passe a la ligne suivante

			//...on recopie le contenu de nouveauTableau[] dans tableauCourant[]

			it++;
		}

	}
}
```

Annexe 2. fichier CAtoolbox.java (méthodes pour construire une règle selon son numéro)

```
public class CAtoolbox {

     static void print (String __s)
     {
          System.out.print(__s);
     }


     static boolean[][] initialiseRegles(int __noCA)
     {
          boolean[][] rules = new boolean[8][4];

          // conditions
          for ( int i = 0 ; i != 8 ; i++ )
          {
               if ( ( i & 1 ) == 1 )
                    rules[i][2] = true;
               if ( ( i & 2 ) == 2 )
                    rules[i][1] = true;
               if ( ( i & 4 ) == 4 )
                    rules[i][0] = true;
          }


          // sorties
          for ( int i = 1 , j = 0 ; i != 256 ; i*=2, j++ )
          {
               if ( ( __noCA & i ) > 0 )
                    rules[j][3] = true;
               else
                    rules[j][3] = false;
          }


          return rules;
     }


}
```