Le jeu de la vie

Table of Contents

1. Énoncé du problème

Le jeu de la vie (http://fr.wikipedia.org/wiki/Jeu_de_la_vie) est un automate imaginé en 1970 par John Horton Conway (http://fr.wikipedia.org/wiki/John_Horton_Conway).

On peut se le représenter comme un damier dont certaines cases contiennent ou non des « cellules ».

Le principe consiste à étudier l'évolution de la vie sur ce damier au cours d'un certain nombre de générations et selon les règles décrites dans la page Wikipédia mentionnée au début du sujet.

Initialement, le jeu contient un certain nombre de cellules et l'on étudie l'évolution du jeu au cours des générations. Cette page (https://cypris.fr/loisirs/le_jeu_de_la_vie.pdf) montre quelques combinaisons initiales et leur évolution au cours du temps.

2. Analyse du problème

Intuitivement, on a plusieurs possibilités pour représenter le plateau du jeu en Java :

- Utiliser un *tableau de tableaux*, comme on le ferait en C : un plateau de 10 par 10 serait représenté par un tableau de 10 tableaux, chacun ayant 10 éléments « vivant » ou « vide ». Le plateau serait donc stocké ligne par ligne. On pourrait aussi remplacer les tableaux par des ArrayList (https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html), mais le principe resterait le même.

  L'inconvénient de cette approche est sa lenteur (il faut parcourir tout le premier tableau pour accéder à la dernière ligne du damier, par exemple). Par ailleurs, le plateau de jeu contenant à priori peu de cellules, on se retrouve avec la même problématique que celle des « matrices creuses ».
- Utiliser un *hachage* : outre l'accès direct à une case, l'avantage est que l'on ne stocke *que* les cellules. Les clés sont les paires (lig, col) de la cellule. Le problème est qu'un hachage associe des clés à des valeurs or, ici, on n'a pas besoin de valeur : on veut uniquement savoir si la clé est présente. On a vu que la bibliothèque Java fournit plusieurs implémentations de hachages, notamment HashMap (https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html).
- Utiliser un *ensemble*, c'est-à-dire une collection d'éléments non dupliqués. Java fournit un type HashSet (https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html) qui est implémenté comme un hachage : on garde donc l'efficacité des hachages sans devoir se soucier d'attribuer une valeur. C'est donc la meilleure solution car, finalement, la seule chose qui compte est bien de stocker les coordonnées des cellules vivantes.

3. La classe Game

Pour définir la classe Game qui représente le jeu, il faut d'abord définir la classe Coord pour représenter une paire de coordonnées.

On pourra alors utiliser un HashSet<Coord> pour stocker les coordonnées des cellules.

Étudiez bien toutes les méthodes qui vous sont déjà offertes par HashSet (https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html)...

Comme on l'a expliqué en cours pour les clés de hachage (TP sur la matrice creuse), la classe des éléments d'un **HashSet** doit diposer d'une méthode **equals** permettant de tester l'égalité des éléments et d'une méthode **hashCode** produisant un identifiant identique pour deux éléments égaux (au sens de **equals**)

Vous devez donc redéfinir ces deux méthodes de **Object** dans la classe **Coord**...

Outre l'ensemble des coordonnées des cellules, un jeu doit également mémoriser le nombre de lignes et de colonnes de son damier :

JAVA

```
class Coord {
    int lig, col;
    public Coord(int lig, int col) { ... }
    ...
}

class Game {
    private HashSet<Coord> damier;
    private int nbLigs, nbCols;
    ...
}
```

Pour initialiser le jeu, on a besoin d'un constructeur qui remplira le damier initial à l'aide d'un tableau de Coord et qui indiquera les dimensions du damier :

JAVA

```
public Game(Coord[] listeInitiale, int nbLigs, int nbCols) {...}
```

On peut aussi créer un jeu initialement vide :

JAVA

```
public Game(int nbLigs, int nbCols) { ... }
```

On peut en profiter pour définir une fonction permettant de tester si une coordonnée représente une cellule (c'est-à-dire si elle appartient à l'ensemble des coordonnées du jeu) :

JAVA

```
public boolean estVivante(Coord coord) { ... }
```

Pour produire un jeu initial de type « planeur » (voir cette page (http://cypris.fr/loisirs/le_jeu_de_la_vie/jeu_de_la_vie.htm)) sur une grille de 5x5, il suffira donc d'écrire :

JAVA
```
Coord[] coords = {new Coord(0, 3), new Coord(1, 2),
                  new Coord(2, 2), new Coord(2, 3),
                  new Coord(2, 4)};
Game jeu = new Game(coords, 5, 5);
```

Avant d'aller plus loin, intéressons-nous à l'affichage du jeu… Nous voulons que les cellules soient représentées par # et les cases vides par un point :

JAVA
```
private String showCell(Coord coord) { ... }
```

On peut donc maintenant se servir de showCell pour représenter une ligne du jeu sous la forme d'une chaîne de # et de points :

JAVA
```
private String showLig(int numLig) { ... }
```

Le jeu complet sera représenté sous la forme d'une chaîne contenant toutes les représentations des lignes produites par showLig, séparées par un retour à la ligne :

JAVA
```
@Override
public String toString() { ... }
```

On peut déjà tester tout cela en créant un jeu de 5x5 et en l'affichant.

JAVA
```
public static void main(String[] args) {
    Coord[] coords = {new Coord(0, 3), new Coord(1, 2),
                      new Coord(2, 2), new Coord(2, 3),
                      new Coord(2, 4)};
    Game jeu = new Game(coords, 5, 5);

    System.out.println(jeu);
}
```

On doit obtenir :

JAVA
```
. . . # .
. . # . .
. . # # #
. . . . .
. . . . .
```

4. Implémentation du jeu

Il reste maintenant à faire évoluer le jeu. Pour cela on a besoin :

- d'une fonction qui renvoie le nombre de cellules adjacentes à une coordonnée du jeu.
- d'une fonction qui produise un nouveau jeu à partir du jeu courant

4.1. Cellules adjacentes à une coordonnée

Pour calculer le nombre de cellules adjacentes à une coordonnée du jeu, on a besoin d'une méthode examinant ses 8 cellules adjacentes et comptabilisant celles qui contiennent une cellule :

JAVA
```
private int nbVoisinesVivantes(Coord coord) { ... }
```

4.2. Génération suivante

Pour établir la génération suivante, il faut mettre en œuvre les règles de vie et de mort d'une cellule d'une génération à l'autre. On commence par écrire une méthode qui indique si une coordonnée contiendra une cellule à la génération suivante :

JAVA
```
private boolean evolution(Coord coord) { ... }
```

- Si une coordonnée a exactement 2 cellules adjacentes, la génération suivante ne sera pas modifiée pour cette coordonnée (soit survie, soit pas de naissance).
- Si une coordonnée a exactement 3 cellules adjacentes, la génération suivante aura une cellule à cette coordonnée (soit survie, soit naissance).
- Si une coordonnée a moins de 2 cellules ajacentes, ou plus de trois cellules adjacentes, la génération suivante n'aura pas de cellule à cet emplacement (soit mort par isolement, soit mort par étouffement, soit pas de naissance).

Il reste ensuite à appliquer cette méthode à toutes les coordonnées possibles d'un jeu pour obtenir sa génération suivante. La méthode nextGen modifie le jeu courant pour qu'il contienne la génération suivante. Elle renvoie un booléen indiquant le jeu est « stable » :

JAVA
```
public boolean nextGen() { ... }
```

> Un jeu est considéré comme stable s'il n'a pas évolué d'une génération à l'autre : il est donc inutile de poursuivre le programme…

4.3. Affichage des damiers

Le programme principal initialise la première génération, demande le nombre de générations à étudier, puis affiche toutes ses générations. Il faut aussi connaître le nombre de lignes et de colonnes du plateau de jeu (ici, on supposera que ces deux nombres sont fixés à 5) :

JAVA

```
public static void main(String[] args) {
    Coord[] coords = {new Coord(0, 3), new Coord(1, 2),
                      new Coord(2, 2), new Coord(2, 3),
                      new Coord(2, 4)};
    Game jeu = new Game(coords, 5, 5);

    Scanner clavier = new Scanner(System.in);
    int nbGen = 0;
    do {
      System.out.print("Combien de générations voulez-vous étudier ? ");
      try {
        nbGen = clavier.nextInt();
      } catch (InputMismatchException e) {
        clavier.nextLine();
      }
    } while (nbGen <= 0);

    // Boucle de calcul et d'affichage des générations
    boolean stable;
    int genCourante = 1;
    do {
      System.out.println("*** Génération " + genCourante + " ***");
      System.out.println(jeu);
      stable = jeu.nextGen();
      genCourante += 1;
    } while (!stable && genCourante <= nbGen);
}
```

L'exécution de ce programme doit produire l'affichage suivant :

JAVA

```
Combien de générations voulez-vous étudier ? 5
*** Génération 1 ***
. . . # .
. . # . .
. . # # #
. . . . .
. . . . .

*** Génération 2 ***
. . . . .
. . # . #
. . # # .
. . . # .
. . . . .

*** Génération 3 ***
. . . . .
. . # . .
. . # . #
. . # # .
. . . . .

*** Génération 4 ***
. . . . .
. . . # .
. # # . .
. . # # .
. . . . .

*** Génération 5 ***
. . . . .
. . # . .
. # . . .
. # # # .
. . . . .
```

4.4. Remarques

- Il faudrait que le constructeur vérifie la cohérence des coordonnées entrées dans le jeu initial par rapport aux nombre de lignes et de colonnes du plateau de jeu…
- La liste des coordonnées initiales et la taille du damier pourraient être saisies par l'utilisateur dans le programme principal au lieu d'être fixés à la compilation.

5. Ta vie sur le Web…

On voudrait maintenant que le jeu puisse produire une page web comme celle-ci (notez le bon goût des couleurs choisies).

Au lieu de modifier la classe que vous venez d'écrire, créez une nouvelle classe GameHTML dérivant de Game. Cette classe devra également implémenter l'interface Webable :

JAVA

```
public interface Webable {
    String initHtml();       // Un seul appel : pour initialiser la page web
    String toHtml();         // Correspond à un toString() en HTML
    String endHtml();        // Un seul appel : pour terminer la page web
}
```

Ces trois méthodes sont censées produire, respectivement, une entête HTML, un corps HTML et une fin de page HTML. Elles peuvent être utilisées de la façon suivante :

JAVA

```
// Boucle de calcul et d'affichage des générations
boolean stable;

System.out.println(jeu.initHtml());

int genCourante = 1;
do {
  System.out.println("<h2>Génération " + genCourante + "</h2>\n");
  System.out.println(jeu.toHtml());
  stable = jeu.nextGen();
  genCourante += 1;
} while (!stable && genCourante <= nbGen);

System.out.println(jeu.endHtml());
```

Le programme **TestGameHTML** permettant de tester votre classe **GameHTML** peut être appelé :

- sans paramètre, auquel cas le nbre de générations étudiées est fixé à 5 et la sortie s'effectue sur stdout.
- avec un seul paramètre précisant le nombre générations étudiées, avec sortie sur stdout
- avec deux paramètres : le nombre de générations étudiées et le nom du fichier de sortie.

Étudiez la méthode **setOut** de **System**…

Par exemple :

SH

```
$ java TestGameHTML 5 game.html
```

6. Tranches de vie…

On veut maintenant mémoriser les différentes générations par lesquelles passe un jeu.

Pour cela, on va sérialiser chaque génération dans un fichier games.dat en modifiant la boucle principale du programme :

JAVA

```
do {
    System.out.println("*** Génération " + genCourante + " ***");
    System.out.println(jeu);
    // Mettre ici l'écriture de jeu dans games.dat
    stable = jeu.nextGen();
    genCourante += 1;
} while (!stable && genCourante <= nbGen);
```

- Faire les modifications nécessaires pour pouvoir sérialiser les objets Game
- Modifier le code du programme pour que la boucle principale sauvegarde toutes les générations dans games.dat
- Ajouter un code de vérification après cette boucle, consistant à relire le fichier games.dat et à afficher les générations relues à partir de ce dernier.

Last updated 2021-10-04 07:54:50 +0200