Travaux pratiques - TP4
Manipulation de tableaux à deux dimensions - Propagation bidimensionnelle (feu de forêt, épidémie...)

Emmanuelle Becker

1 Rappel de cours

Jusqu'à présent, nos tableaux étaient principalement des tableaux unidimensionnels. Ces tableaux avaient une `longueur`, mais la `largeur` était de 1.

Nous sommes maintenant prêts à aborder les tableaux à deux dimensions, appelés **matrices** (singulier : **matrix**). Une matrice est un tableau composé de lignes (on itère généralement sur les lignes dont l'indice est i) et de colonnes (on itère généralement sur les colonnes dont l'indice est j).
Comme vous l'avez deviné, pour lire une matrice entière, nous aurons besoin de deux boucles imbriquées !

Attention ! Les éléments rangés dans une matrice doivent être de même type.

2 Exemple

Voici un tout petit relevé de notes qui nous serivra d'exemple :

```
1         Physique     Chimie   Algebre      Geometrie    Logique
2 Adama   20           18       20           20           16
3 Basile  18           20       18           20           20
4 Coline  16           18       16           20           20
5 Damien  5            14       12           14           15
```

Attention Bien que les noms des élèves et les matières, ils ne font pas partie des données réelles de la matrice. N'oubliez pas que les données d'un tableau à deux dimensions sont toujours du même type (donc on ne peut pas mélanger les titres de colonnes et lignes qui sont des `String` avec le contenu du tableau qui sont des `int`).

Dans notre exemple, la longueur de la table est 4 (il y a 4 lignes) et la largeur est 5 (il y a 5 colonnes).

Pour définir et utiliser une table à deux dimensions, on doit utiliser 2 séries de crochets. Le premier crochet permet d'aller sélectionnet la ligne voulue, le second crochet permet d'aller sélectionner la colonne voulue.

A partir de notre exemple, nous allons voir comment déclarer une matrice, puis comment remplir et comment acceder aux éléments contenus dans la matrice.

2.1 Déclarer une matrice

Pour déclarer une matrice 4 lignes 5 colonnes de nombres entiers, on utilise la commande suivante :

```
int[][] scores = new int [ 4 ] [ 5 ] ;    // Declares a 2-D array
```

En détails, cette instruction dit explicitement qu'il y aura 4 tables de nombres entiers à stocker dans `scores`, et que chacune de ces tables stocke 5 entiers.

Cette déclaration vous permet donc de stocker une matrice de 20 éléments dont les indices seront les suivants (avec 2 crochets à chaque fois, comme expliqué précédemment).

```
[0][0]   [0][1]   [0][2]   [0][3]   [0][4]
[1][0]   [1][1]   [1][2]   [1][3]   [1][4]
[2][0]   [2][1]   [2][2]   [2][3]   [2][4]
[3][0]   [3][1]   [3][2]   [3][3]   [3][4]
```

Parfois, il peut être utile de se souvenir que votre tableau à deux dimensions est un tableau unidimensionnel de pointeurs vers des tableaux unidimensionnels. Cela signifie que si vous extrayez une ligne (par exemple avec l'instruction `scores[1]`), le résultat est un tableau unidimensionnel (dans notre exemple de type `int[ ]`).

2.2 Remplir une matrice

2.2.1 Remplissage par liste

Tout comme pour un tableau à une dimension, il est possible de remplir un tableau à deux dimensions en utilisant une liste au moment où le tableau est déclaré. Remarquez les accolades à l'intérieur de la liste, ce sont elles qui structurent notre `tableau des tableaux`.

```
int[][] scores = {   { 20, 18, 20, 20, 16 },
                     { 18, 20, 18, 20, 20 },
                     { 16, 18, 16, 20, 20 },
                     { 05, 14, 12, 14, 15 }
                 };
```

2.2.2 Remplissage par défaut

Lorsqu'un tableau est créé, il est automatiquement rempli avec un zéro (pour les valeurs numériques), un `False` (pour les valeurs booléennes) ou un `null` (pour les valeurs de type chaîne de caractères).

2.2.3 Remplissage avec une valeur donnée

Dans ce cas, il est nécessaire d'utiliser des boucles imbriquées. La boucle extérieure contrôle l'indice des lignes et la boucle intérieure contrôle l'indice des colonnes.

```
for (i_row = 0; i_row < 4; i_row++){
     for (j_column = 0; j_column < 5; j_column++){
        scores[i_row][j_column] = 100;
     }
}
```

3 Objectifs de la feuille d'exercice

Le but de cet exercice est de modéliser la propagation d'un incendie dans une forêt, en fonction notamment de la densité des arbres dans la forêt.

La forêt sera modélisée par une matrice `carrée` de dimension `n` (nous imposerons que $n \geq 10$). Chaque cellule de la matrice prendra une valeur entière en fonction de son état :
- valeur de la cellule 0 pour les cellules **vides** (pas d'arbre) ;
- valeur de la cellule 1 pour les cellules **arbre** ;
- valeur de la cellule 5 pour les cellules **arbre en feu** ;
- valeur de la cellule -1 pour les cellules **cendres** (un arbre après avoir brûlé).

Nous créerons des forêts de densité donnée, avec un positionnement aléatoire des arbres sur notre matrice carrée. Ensuite, nous mettrons le feu à un arbre aléatoire, et selon les règles de propagation, nous verrons l'évolution du système global : la forêt va t-elle être entièrement rasée ou non ? Autre question, plus générale : la relation entre densité d'arbre et probabilité de raser la forêt est-elle linéaire ?

Let's go !

4 Classes `Automata` et `TestAutomata`

4.1 Création de la classe `Automata`

Créer une classe `Automata` avec deux champs privés :
- un tableau d'entiers nommé `matrix`, qui stockera notre forêt modélisée ;
- un entier nommé `dimension` qui stockera la dimension de la matrice.

4.2 Les constructeurs

Nous devons maintenant construire une forêt, c'est-à-dire remplir notre matrice avec des 0 et des 1 afin de représenter une forêt spécifique. Ajouter les constructeurs suivants (rappel, nous imposons que $n \geq 10$) :

1. Un premier constructeur `public Automata(int n, double p)`, qui crée une forêt aléatoire de dimension n, et dont la densité des arbres est p (ce qui signifie que chaque cellule a une probabilité p de contenir un arbre) ;
2. un second constructeur `public Automata (double p)`, qui crée une forêt aléatoire de dimension $n = 10$, et dont la densité des arbres est p.

4.3 Affichage de la forêt

Ajoutez une méthode appelée `forestDisplay()`, qui ne renvoie rien mais affiche la matrice actuelle de sorte que :
- les cellules vides sont représentées par `'.'`,
- les arbres sont représentés par `'T'` »,
- les arbres en feu par `'O'`,
- et enfin les cendres par `'_'`.

Attention : On ne vous demande pas de modifier la matrice `matrix`, attribut de la classe `Automata`. On vous demande de lire cette matrice (avec des 0, 1, 5, -1), et d'afficher un `'T'` quand vous lisez un 1, etc...

4.4 Premières méthodes

Ajoutez les méthodes suivantes :

— Une méthode `isRazed()`, qui renvoie le booléen `False` si la forêt a au moins un arbre, et `True` s'il n'y a plus d'arbres dans la forêt.

— Une méthode est `isOnFire()`, qui renvoie le booléen `True` si l'un des arbres de la forêt est en feu, et `False` dans le cas contraire.

4.5 Création de la classe `TestAutomata`

Pour tester votre classe `Automata`, qui contient déjà quelques méthodes intéressantes, créez une autre classe dans le même package appelée `TestAutomata`, qui contient une méthode `public static void main`. Dans cette méthode principale, créez quelques automates de la dimension que vous souhaitez, affichez-les et testez les quelques méthodes écrites. Par exemple :

```
iAC = new Automata(0.3) ;
System.out.println("Is the forest on fire ? " + iAC.isOnFire()) ;
System.out.println("Is the forest completely rased ? "  + iAC.isRazed()) ;
iAC.forestDisplay() ;

jAC = new Automata(25, 0.8) ;
System.out.println("Is the forest on fire ? " + jAC.isOnFire()) ;
System.out.println("Is the forest completely rased ? "  + iAC.isRazed()) ;
jAC.forestDisplay() ;

kAC = new Automata(20, 0.7) ;
System.out.println("Is the forest on fire ? " + iAC.isOnFire()) ;
System.out.println("Is the forest completely rased ? "  + iAC.isRazed()) ;
kAC.forestDisplay() ;
```

Ajoutez quelque chose pour tester que vous avez bien pris en compte l'hypothèse que la dimension de la forêt est au moins égale à 10.

4.6 Feu !

Si nous voulons étudier la propagation d'un incendie, nous devons mettre le feu à une cellule de notre matrice. Créer les deux méthodes suivantes (encore une illustration du polymorphisme) :

1. une méthode publique `putFire(int i, int j)` qui modifie la matrice de manière à ce que la cellule i, j soit en feu (ce qui signifie que $matrix[i][j] = 5$) ;
2. une méthode publique `putFire()` qui modifie la matrice pour mettre une cellule aléatoire en feu (même s'il n'y a pas d'arbre à cet endroit).

Vous pouvez maintenant, à partir de la classe `TestAutomata`, tester simultanément vos méthodes `putFire()` et votre méthode `isOnFire()`... A vous de jouer !

4.7 Propagation du feu

Nous proposons les règles suivantes pour déterminer l'état futur d'une cellule en fonction de son état actuel :

1. si la cellule est « arbre en feu » dans son état actuel, alors son prochain état sera « cendre » ;
2. si la cellule est « arbre » dans son état actuel et si l'un de ses voisins est actuellement dans l'état « arbre en feu », alors son prochain état sera « arbre en feu » ;
3. dans les autres cas, l'état de la cellule ne change pas.

Pour rendre votre code aussi explicite que possible, vous pouvez commencer par coder les deux méthodes privées suivantes :

1. la méthode privée `isTree(int i, int j)` qui renvoie le booléen `true` si et seulement si `matrix[i][j]` contient actuellement un arbre (c'est-à-dire un arbre qui n'est pas en feu), et `false` dans le cas contraire ;
2. la méthode privée `isOnFire(int i, int j)` qui renvoie le booléen `true` si et seulement si `matrix[i][j]` contient actuellement un arbre en feu, et `false` sinon (encore une illustration du polymorphisme ...)

Nous entrons dans le vif du sujet avec les méthodes suivantes :

1. la méthode privée `hasNeighborOnFire(int i, int j)`, qui renvoie le booléen `true` si l'une des cellules autour de la cellule (i, j) est en feu, et `false` dans le cas contraire (veuillez ne pas considérer les diagonales comme des voisins ; ainsi, une cellule a au plus 4 voisins) ;
2. la méthode privée `nextState(int i, int j)`, qui renvoie un code entier pour le prochain état de la cellule (i, j) en fonction de son état actuel et de l'état actuel de ses voisins ;
3. enfin, la méthode privée `propagateFire1()` qui calcule la matrice suivante en fonction de la matrice actuelle (vous aurez besoin de nombreux appels à la méthode `nextState`).
   Avertissement : Vous ne devez pas modifier la `matrix` au fur et à mesure que vous calculez les états suivants (essayez de comprendre pourquoi) ! Vous devrez déclarer une nouvelle matrice, allouer l'espace correctement, et remplir cette nouvelle matrice avec les états suivants. Une fois tous les états suivants calculés, vous pouvez assigner à `this.matrix` la nouvelle matrice.

4.8 Test de ces méthodes

A partir de la classe `TestAutomata`, vous pouvez tester votre méthode `propagateFire1()` (il faudra penser à passer la méthode de `private` à `public` le temps du test), par exemple avec les lignes de code suivantes :

```
iAC=new Automata(15,0.8) ;
iAC.displayForest() ;
iAC.putFire(5,5) ;
iAC.displayForest() ;
iAC.propagateFire1() ;
iAC.displayForest() ;
```

4.9 Propagation du feu sur une longue période

Nous savons comment passer de l'état actuel à l'état suivant. Supposons qu'il s'agit de l'évolution de notre feu après une heure. Nous voulons maintenant calculer l'évolution du feu pendant plusieurs heures, ou l'évolution du feu jusqu'à ce que le feu soit éteint faute de combustible... A vous de jouer pour écrire et tester 2 méthodes de propagation du feu sur une longue période.

1. La méthode publique `propagateFire(int n)`, qui affiche l'évolution du feu heure par heure pour n heures. Pour faciliter la visualisation, il peut être intéressant de mettre le programme en «pause» entre les différentes heures représentées, ce que vous pouvez faire avec le morceau de code suivant :

```
try {
  Thread.sleep (5000);
} catch (InterruptedException ex) {
  Thread.currentThread().interrupt()
}
```

2. La méthode publique `propagateFire()`, qui affiche l'évolution de l'incendie heure par heure jusqu'à son extinction.
3. Testez ces méthodes à partir de la classe `TestAutomata`, en variant les scénarios.

4.10 L'importance cruciale de la densité d'arbres

La densité des arbres dans la forêt a-t-elle une incidence sur la propagation du feu ? C'est ce que nous allons tester maintenant.

```
for (double d=0.05; d<=1;d=d+0.05){
  int c=0 ;
  for (int j=0; j<100;j++){
    iAC=new Automata(25,d) ;
    iAC.putFire();
    iAC.propagateFeu() ;
    if iAC.isRazed() { c=c + 1 ; }
  }
  System.out.println("With density "+d+" : "+c+" forest completely razed")
}
```

On vous demandera de faire un graphique (sous R, Excel, LibreOffice - tout ce que vous voulez) pour mesurer la dépendance entre la densité d'arbres (axe des x) et la probabilité d'avoir totalement rasée la forêt (axe des y). La dépendance est-elle linéaire ?

4.11 Bonus 1 : changement des règles de propagation

Certaines modifications peuvent être apportées au modèle actuel. Par exemple, réfléchissez à la manière de prendre en compte tous les voisins dans le calcul de l'état suivant (voisins en diagonale inclus), ou à la manière de prendre en compte le vent.

4.12 Bonus 2 : le temps de calcul

Vous avez peut-être constaté que les simulations sont pluis longues lorsque la taille de votre matrice est plus grande. Nous pouvons chercher à caractériser ceci, par exemple avec le code suivant, dans lequel vous devez ajouter des instructions permettant de mesurer combien de secondes sont necessaires pour faire tourner 100 simulations pour des matrices de tailles $n = 10$, puis 100 simulations pour des matrices de taille $n = 15$, puis $n = 20$... Allez chercher sur le net comment mesurer le temps d'execution, puis cherchez dans l'API pour vérifier que les librairies existent encore et comment executer les méthodes...

```
for (double n=10; n<=30;n=n+5){
  for (int j=0; j<100;j++){
    iAC=new Automata(25,0.3) ;
    iAC.putFire();
    iAC.propagateFire() ;
  }
}
```

Avertissement : Ce n'est pas tout a fait propre comme façon de mesurer le temps d'execution, mais c'est néanmoins une bonne première approximation. Par contre, il faut absolument que vous ne lanciez rien d'autre en parralèle pendant l'execution de votre programme... Le but est de laisser le CPU disponible pour votre programme.

Là encore, faites un graphique pour mesurer la dépendance entre taille de la matrice (axe des x) et temps de simulation (axe des y). La relation est-elle linéaire ?

5 Bonus : propagation d'une épidémie (à faire dans un nouveau package !)

Fortement inspiré de l'exercice précédent, imaginez la modélisation simple d'une épidémie très transmissible. Vous placerez les individus sur une matrice, même si ce n'est pas très réaliste. Les cellules seront peuplées :

- d'individus sains non vaccinés ;
- d'individus sains vaccinés ;
- d'individus souffrant de la polio ;
- d'individus guéris ;
- de personnes décédées ;
- personne (cellule vide).

Au départ, vous peuplerez votre matrice d'une densité donnée d d'individus, dont un pourcentage p sera vacciné. Ensuite, vous contaminerez par la polio une cellule au hasard.

Vous calculerez la propagation avec les règles suivantes :

- Les individus sains non vaccinés, dont l'un des voisins est malade dans l'état actuel, deviennent malades au cours de l'étape suivante.
- Un individu malade se rétablit ou meurt dans l'étape suivante, avec une probabilité de décès de m ;
- Les individus rétablis ne peuvent plus être malades.
- Les individus vaccinés ne peuvent pas être malades.
- Les cellules vides restent vides.