UPMC/Licence/Info/2I013
Tendroid
Modèle du jeu

Janvier 2019

Le jeu *get10* est un jeu de *puzzle gravitationnel*. Il se joue sur une grille carrée de 5 sur 5 dont chaque case contient un nombre.

Pour jouer, on sélectionne une case d'un groupe de cases adjacentes portant le même nombre pour la remplacer par une seule case contenant le successeur du nombre éliminé. Toutes les autres cases du bloc sont vidées et le plateau est compacté par gravité. Les cases vidées par le compactage sont remplies avec de nouveaux nombres.

La partie est gagnée lorsque l'on a formé un groupe de 9 : son élimination amènera à obtenir la valeur 10.

On peut décomposer une *action de jeu* en 4 phases :

1. sélection d'un groupe par désignation d'une case du groupe
2. «collapse» du groupe sélectionné sur une case : la valeur de la case sélectionnée est augmentée de 1 et les autres cases du groupe sont vidées
3. compactage de la grille par gravité
4. remplisage des cases vides par de nouvelles valeurs

Exemple d'action de jeu soit la grille initiale

2	4	3	1	1
1	3	2	1	1
3	3	1	1	1
3	2	2	3	1
1	3	1	2	3

1. On choisit la case (4,0) (en gras sur la figure ci-dessous), ce qui sélectionne le grand groupe de 1 (en italiques)
2. On choisit ensuite la case (3,1) de ce groupe : sa valeur devient 2 (en gras sur la figure) et toutes les autres cases du groupe sont vidée
3. On compacte la grille *par gravité* (sur la figure, les valeurs qui sont «tombées» sont en italique)
4. On remplit les cases vides avec de nouvelles valeurs aléatoires (en italique)

1.

2	4	3	*1*	**1**
1	3	2	*1*	*1*
3	3	*1*	*1*	*1*
3	2	2	3	*1*
1	3	1	2	3

2.

2	4	3		
1	3	2	**2**	
3	3			
3	2	2	3	
1	3	1	2	3

3.

2	4			
1	3	*3*		
3	3	*2*	*2*	
3	2	2	3	
1	3	1	2	3

4.

2	4	*1*	*2*	*3*
1	3	3	*2*	*1*
3	3	2	2	*2*
3	2	2	3	*2*
1	3	1	2	3

Précisions *au sens du jeu get10* :

- une case peut contenir une valeur ou être temporairement «vide».
- on repère les cases du plateau par un couple de coordonnées qui constitue leur «position». La première coordonnée indique le numéro de colonne (abscisse) et la seconde le numéro de ligne (ordonnée). La case en haut à gauche a pour position (0,0). La case en bas à droite a pour position (4,4).
  Une case est «en dessous» d'une autre lorsque le numéro de ligne de la première est supérieur au numéro de ligne de la seconde ;
- les cases sont dites «adjacentes» par contact horizontal ou vertical uniquement (pas de diagonale) ;
- un «groupe» est constitué d'au moins deux cases adjacentes contenant une même valeur ;
- un case dont les cases adjacente ne portent pas la même valeur qu'elle est dite «isolée» ;
- on «collapse» un groupe à une position donnée en augmentant de 1 la valeur contenue à la position donnée et en vidant les autres cases du groupe.
- on «compacte» une grille en permutant, colonne par colonne ses cases vides et non vides de manière à ce que, dans la colonne, aucune case vide ne se trouve en dessous d'une case non vide. L'ordre, par numéro de ligne, des cases non vides est préservé.

Le modèle du jeu se compose d'un ensemble de classes et de leurs méthodes qui permettent de représenter un *état* du jeu (valeurs contenue dans la grille) et de réaliser les principales *actions* de jeu (sélection d'un groupe, «collapse» d'un groupe, compactage de la grille, remplissage des cases vidées) qui réalisent autant de *transitions* de l'état du jeu.

1 Classes utilitaires

1.1 Position

Une *position* sur une grille est un couple formé d'un numéro de colonne et d'un numéro de ligne. La classe `Position` doit fournir :

- un *constructeur* de signature `public Position(int col, int lig)`.
- les *méthodes d'accés* de signatures `public int getCol()` et `public in getLig()` qui donnent respectivement la valeur du numéro de colonne de la position et la valeur de son numéro de ligne.
- la méthode `public boolean equals(Position p)` qui donne la valeur `true` si les numéros de colonne et de ligne de la position sont identiques à ceux de `p` ; et `false` sinon.

1.2 Liste de positions

La classe `PositionList` étend la classe `ArrayList<Position>`. Elle fournit les méthodes :

- `public boolean add(int col, int lig)` qui ajoute à la liste la position construite avec les valeurs `col` et `lig`
- `public boolean contains(Position p)` qui donne la valeur `true` s'il existe dans la liste une position égale (au sens de `Position.equals`) à `p` ; et `false` sinon.

1.3 Grille

Une grille est représentée par la classe `Grid`. Les valeurs des cases d'une grille sont des instances de la classe `Integer` ou la valeur `null`. Dans ce dernier cas, la case est réputée *vide*. La classe `Grid` fournit

- un constructeur de signature `public Grid(int nbCol, int nbLig)` où `nbCol` est le nombre de colonnes (largeur) de la grille et `nbLig` son nombre de lignes (hauteur). Initialement, toutes les cases de la grille sont *vides*.

— les méthodes d'accés `public int nbCol()` et `public int nbLig()` qui donnent respectivement la largeur et la hauteur de la grille.
— la méthode de signature `public PositionList allPositions()` qui donne la liste de toutes les positions de la grille triées par ordre lexicographique sur les numéro de ligne et numéro de colonne.
— la méthode de signature `public boolean regularPosition(Position p)` qui donne `true` si la position `p` est une position dans la grille ; et `false`, sinon.
— la méthode de signature `public boolean isEmpty(Position p)` qui donne `true` si la position `p` est une position dans la grille et qu'elle est vide.
— la méthode `public Integer get(Position p)` qui donne la valeur en position `p` dans la grille. Le résultat n'est pas spécifié si `p` n'est pas une position dans la grille.
— la méthode `public void set(Position p, Integer v)` qui place la valeur `v` en position `p` dans la grille. Le résultat n'est pas spécifié si `p` n'est pas une position dans la grille.
— la méthode `void unset(Position p)` qui rend *vide* la position `p` de la grille. Le résultat n'est pas spécifié si `p` n'est pas une position dans la grille.
— la méthode `public PositionList adjPositions(Position p)` qui donne la liste des positions adjacentes (au sens du jeu *get10*) à la position `p` dans la grille.

2 La grille du jeu *get10*

La classe `TenGrid` étend la classe `Grid` pour fournir des méthodes spécifiques au jeu *get10*. À savoir :
— `public PositionList getGroup(Position p)` donne la liste des positions constituant le groupe (au sens du jeu *get10*) qui contient `p`. Cette liste contient `p`. Ce groupe est *maximal* au sens suivant : étant donné une position `p1` de la liste, toutes les positions adjacentes à `p1` qui contiennent la même valeur que `p1` (et donc, que `p`) appartiennent à la liste.
Si `p` est isolée (au sens du jeu *get10*), la liste est vide.
— `public void collapseGroup(Position p)` réalise sur la grille le *collapse* (au sens du jeu *get10*) à la position `p` du groupe dont fait partie cette position.
Si la position n'est pas une position dans la grille ou qu'elle est isolée (au sens du jeu *get10*), l'état de la grille est inchangé.
— `public void pack()` compacte (au sens du jeu `get10`) la grille.
— `void refill(int[] ns)` rempli les cases *vides* de la grille avec les valeurs de `ns`.
Le tableau `ns` doit contenir au moins autant de valeurs que de cases vides. Les valeurs successives de `ns` remplissent les valeurs successives de la liste des positions (vides) données par `emptyPositions()`.
— `PositionList emptyPositions()` donne la liste des positions vides de la grille.
La classe `TenGrid` doit également fournir les constructeurs
— `public TenGrid()` qui construit une grille vide de taille 5 sur 5 ;
— `public TenGrid(int[] ns)` qui construit une grille de taille 5 sur 5 et en remplit les cases avec les valeurs du tableau `ns`. Les valeurs sont rangées du tableau dans la grille depuis la case (0,0) avec la première valeur du tableau, puis par ordre croissant de colonne, puis de ligne avec les valeurs successives du tableau. Le comportement du constructeur n'est pas spécifié si le tableau contient moins de 25 valeurs.

3 Les actions de jeu

La classe `TenGame` modélise les actions du joueur. Il y en a essentiellement deux :
— désigner une case pour sélectionner un groupe
— désigner une case du groupe sélectionné pour effectuer le collapse/compactage/remplissage
Une autre action de jeu fort utile est la possibilité de ≪désélectionner≫ un groupe. On associera cette action au fait de désigner une case hors du groupe sélectionné, s'il existe.

La classe `TenGame` étend la classe `TenGrid`. Elle fournit :

- un constructeur de signature `public TenGame()` ;
- un constructeur de signature `public TenGame(int[] ns)` ;
- une méthode `public PositionList getSelectedGroup()` donne la liste des positions du groupe sélectionné, s'il existe, et `null` sinon ;
- la méthode `public void transition(Position p)` qui étant donné une position `p` de la grille effectue l'une des trois actions de jeu possible :
  1. sélectionner un groupe.
  2. collapser un groupe.
  3. changer ou annuler la sélection de groupe.

  Si la position `p` est celle d'une case isolée, la sélection de groupe est `null`.

4 Propriétés attendues

4.1 Classe `Position`

Constructeur et méthodes `getCol` et `getLig` Pour tout `int i1, i2` :

```
(new Position(i1,i2)).getCol() == i1
(new Position(i1,i2)).getLig() == i2
```

Méthode `equals` Pour tout `Position p` :

```
p.equals(p) == true
```

si `p.getCol() == i1` et `p.getLig() == i2` :

```
p.equals(new Position(i1,i2)) == true
```

si `p.getCol() != i1` ou `p.getLig() != i2` :

```
p.equals(new Position(i1,i2)) == false
```

4.2 Classe `PositionList`

Méthode `contains` Pour toute `PositionList`, `ps`, et pour toute `Position p`,
si il existe un `int i`, tel que `ps.get(i).equals(p) == true` :

```
ps.contains(p) == true
```

si pour tout `int i`, `ps.get(i).equals(p) == false` :

```
ps.contains(p) == false
```

Méthode `add` Pour toute `PositionList`, `ps`, et pour toute `Position p`,

```
(new PositionList()).contains(p) == false
ps.add(p).contains(p) == true
```

4.3 Classe `Grid`

Constructeur et méthodes `nbCol` et `nbLig` Pour tout `int i1,i2` :

```
(new Grid(i1,i2)).nbCol() == i1
(new Grid(i1,i2)).nbLig() == i2
```

Méthode `regularPos` Pour tout `Position p` et toute `Grid g`,
si `p.getCol()`$\in$[0,g.nbCol()-1] et `p.getLig()`$\in$[0,g.nbLig()-1] :

```
g.regularPos(p) == true
```

si `p.getCol()`$\neg\in$[0,g.nbCol()-1] ou `p.getLig()`$\neg\in$[0,g.nbLig()-1] :

```
g.regularPos(p) == false
```

Méthodes `set`, `get`, `unset` et `isEmpty` Pour tout `Grid` g, pour tout `int i`, pour tout `Position p` tel que `g.regularPos(p) == true`,
après `g.set(p,i)` :

```
g.get(p) == i
```

après `p.unset(p)` :

```
g.isEmpty(p) == true
```

Méthode `adjPositions` Pour tout `int i` tel que `i`$\in$[0,g.adjPositions(p).size()-1]

```
g.regularPos(g.adjPositions(p)) == true

   g.adjPositions(p).get(i).getCol() == p.getCol()+1
|| g.adjPositions(p).get(i).getCol() == p.getCol()-1
|| g.adjPositions(p).get(i).getLig() == p.getLig()+1
|| g.adjPositions(p).get(i).getLig() == p.getLig()-1
```

Méthode `allPositions` Pour tout `Position p` :

```
g.allPositions().contains(p) == g.regularPosition(p)
```

pour tout `int i, j` tels que `i,j`$\in$[0,`g.allPositions().size()`-1], si `i`$\leq$`j` :

```
ps.get(i).nbLig() <= ps.get(j).nbLig()
```

et si `ps.get(i).nbLig() == ps.get(j).nbLig()` :

```
ps.get(i).nbCol() <= ps.get(j).nbCol()
```

4.4 Classe `TenGrid`

Constructeur

```
(new Grid()).nbCol == 5
(new Grid()).nbLig == 5
```

Pour tout `int[] ns` tel que `ns.length`$\geq$25,

```
(new Grid(ns)).nbCol == 5
(new Grid(ns)).nbLig == 5
```

pour tout `int i`$\in$[0,`ns.length-1`] :

```
(new TenGrid(ns)).get(new Position(i%5,i/5)) == ns[i]
```

pour tout `Position p` tel que `(new TenGrid(ns)).regular(p)`

```
(new TenGrid(ns)).get(p) == ns[p.nbLig()*5+p.nbCol()]
```

Méthode `getGroup` Pour tout `TenGrid g`, pour tout `Position p` tel que `g.regularPos(p) == true`, pour tout `int i`$\in$[0,`g.getGroup(p).size()`-1] :

```
g.getGroup(p).get(i) == g.get(p)
```

pour tout `Position p1` tel que `g.getGroup(p).contains(p1) == true`, pour tout `Position p2` tel que `g.adjPositions(p1).contains(p2)` et `g.get(p1) == g.get(p2)` :

```
g.getGroup(p).get(p1).contains(p2) == true
```

Méthode `collapseGroup` Pour tout `TenGrid g`, pour tout `Position p`,
si `g.get(p) == n` et si `g.getGroup(p).size() > 1`,
après `g.collapseGroup(p)` :

```
g.get(p) == n+1
```

pour tout `Position p1`, si `g.getGroup(p).contains(p1) == true`,
après `g.collapseGroup(p)` :

```
g.isEmpty(p1) == true
```

si `g.getGroup(p).contains(p1) == false` et si `g.get(p1) == m`,
après `g.collapseGroup(p)` :

```
g.get(p1) == m
```

Méthode `emptyPositions` Pour tout `TenGrid g`, pour tout `Position p`,
si `g.emptyPositions().contains(p) == true` :

```
g.regularPosition(p) && g.isEmpty(p) == true
```

Méthode `refill` Pour tout `TenGrid g` telle que `ps` = `g.emptyPositions()`, pour tout `int ns[]` tel que `ns.length`≥`ps.size()`,

après `g.refill(ns)` : pour tout `int i`,

```
g.get(ps.get(i)) == ns[i]
```