Projet de programmation Java
Puissance 4

Juliusz Chroboczek et Gabriel Scherer

Version du 13 novembre 2014

Le but de ce projet est d'implémenter en Java un jeu de plateau qui s'appelle Puissance 4. Votre programme permettra (1) à deux joueurs humains de jouer entre eux, (2) à l'utilisateur humain de jouer contre l'ordinateur et (3) à l'ordinateur de jouer contre lui-même.

1 Le jeu de *Puissance 4*

1.1 Règles du jeu

Puissance 4 est un jeu de plateau à deux joueurs. Il se joue sur un plateau vertical de 7×6 cases (Figure 1). Chaque joueur dispose de 21 pions, rouges ou jaunes. Rouge joue le premier, et les joueurs jouent tour-à-tour. Le but du jeu est d'aligner quatre pions de sa couleur, verticalement, horizontalement, ou en diagonale.

Lorsque c'est son tour, le joueur p choisit une colonne qui n'est pas encore pleine, et y insère un de ses pions. Le pion tombe jusqu'à ce qu'il soit coincé par un autre pion (Figure 2) ou par le bas de la colonne.

La partie se termine lorsqu'un joueur a gagné (il a aligné quatre pions de sa couleur, Figure 3), ou alors lorsque le plateau est plein (match nul).

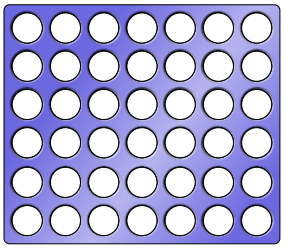

FIGURE 1 – Plateau vide

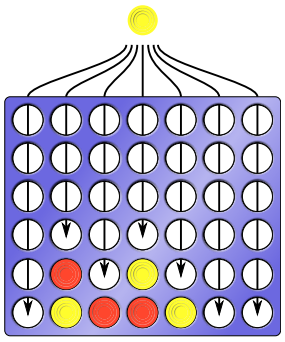

FIGURE 2 – Coups possibles

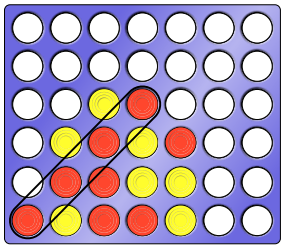

FIGURE 3 – Fin de partie

1.2 Implémentation

Puissance 4 est facile à implémenter : il suffit de modéliser le plateau par un tableau dont les cases peuvent contenir trois valeurs : case vide, pion rouge ou pion jaune.

Lorsque c'est un utilisateur humain qui joue, le programme lui demandera de choisir une colonne et devra déterminer automatiquement (1) si cette colonne n'est pas déjà pleine et (2) à quelle case de la colonne s'arrête le pion. De même, lorsque c'est l'ordinateur qui joue, il choisira une colonne non pleine, et déterminera la case dans laquelle s'arrête le pion.

Après chaque tour, il faudra vérifier la condition de terminaison : si un joueur a gagné, et si le plateau est plein.

2 Fonctionnalités du programme

Votre programme fera s'affronter deux joueurs (Rouge, joueur 1, ou Jaune, joueur 2). Chaque joueur peut être soit un humain, soit un ordinateur. On choisira la nature des joueurs en lançant le programme :

```
$ java Puissance4 PvP
$ java Puissance4 PvC
$ java Puissance4 CvP
$ java Puissance4 CvC
```

Par exemple, `PvC` signifie *Player versus Computer* : le joueur 1 est un humain, le joueur 2 un ordinateur.

Pour chaque joueur qui est un ordinateur, le programme demande de choisir une *stratégie* de jeu. Un des utilisateurs entre le nom de la stratégie dans le terminal.

Enfin, pour commencer la partie, on affiche un plateau vide avec la représentation suivante :

```
1234567
.......
.......
.......
.......
.......
.......
1234567
```

Un point représente une case vide, et on utilisera `R` et `J` pour les cases occupées par un point rouge ou jaune.

C'est ensuite au tour de Rouge de jouer.

Tour de jeu À chaque tour, le programme demande au joueur dont c'est le tour de choisir un coup.

Si c'est à un humain de joueur, le programme lira le numéro de la colonne entré par l'humain dans le terminal. Si c'est à un ordinateur de jouer, le programme appellera une fonction

correspondant à la stratégie choisie, pour décider quel coup jouer, et affichera le numéro de la colonne choisie dans le terminal.

Ensuite, le programme "joue" le coup : il vérifie que le coup est valide (sinon, on redemande un nouveau coup au joueur ou à l'ordinateur), et modifie le plateau de jeu. Il affiche le nouvel état du plateau, par exemple :

```
1234567
.......
.......
.......
.......
.......
...R...
..RJJR.
1234567
```

Enfin, le programme vérifie les conditions de fin : si l'un des joueurs a quatre pions alignés, il a gagné, et si le plateau est rempli, c'est un match nul.

Si l'une de ces conditions est remplie, le programme le signale et termine la partie. Sinon, c'est au tour de l'autre joueur de jouer.

2.1 Test de fin de partie

Cette section propose des pistes pour implémenter le test de victoire d'un des joueurs, qui est une partie délicate du projet. Il n'est pas obligatoire de respecter ces conseils, vous pouvez faire d'autres choix du moment qu'ils donnent un programme qui fonctionne.

Je vous conseille de représenter le plateau de jeu par un tableau de tableaux d'entiers : `0` pour une case vide, `1` pour une case rouge (joueur 1), `2` pour une case jaune (joueur 2). Une position dans le plateau correspond à un couple d'entiers `i` et `j`.

Il est utile d'avoir une fonction qui vérifie si une position `(i, j)` est à l'intérieur ou en dehors du plateau.

```
public static bool dansPlateau(int[][] plateau, int i, int j)
```

Pour tester la présence de quatre pions alignés, vous devrez suivre les cases de la même couleur dans une certaine direction (vers le haut, vers la droite, selon une diagonale...). Une direction peut être représenté par un couple d'entiers `di` et `dj`, valant -1, `0` ou `1`, qui représentent les petits déplacements à ajouter à la position `(i, j)` pour avoir les coordonnées `(i+di, j+dj)` de la case suivante dans cette direction. Par exemple, `(1, -1)` correspond à une des diagonales.

Pour implémenter la condition de victoire, il est utile de savoir compter le nombre de cases d'une certaine couleur alignées, dans une direction, à partir d'une certaine position :

```
public static int longueurAlignement(int couleur, int i, int j, int di, int dj)
```

Je laisse votre imagination faire le reste.

(Si vous avez du goût et préférez utiliser l'anglais pour éviter les mélanges malheureux, vous pourrez utiliser `insideBoard` et `streakLength`.)

2.2 Stratégies

La fonction qui prend en paramètres un plateau de jeu et le joueur dont c'est le tour et retourne la colonne où joue l'ordinateur s'appelle une *stratégie*. La stratégie la plus simple est la *stratégie aléatoire* — elle choisit au hasard une colonne jouable. Son nom (pour la choisir en début de partie) est `hasard`.

L'autre stratégie que vous devrez implémenter est la *recherche de coup gagnant de profondeur 0*. Dans cette stratégie, l'ordinateur joue une colonne qui lui permet de gagner tout de suite (un *coup gagnant en 0 coups*) s'il en existe une ; sinon, il choisit une colonne non pleine au hasard. Le nom de cette stratégie est `coup gagnant`.

3 Extensions

Toutes les extensions au sujet seront les bienvenues, et seront examinées avec bienveillance par le(s) correcteur(s). Nous espérons que vous implémenterez des extensions auxquelles nous n'avons pas pensé, mais si vous n'avez pas d'idées, voici les nôtres.

Stratégies plus malines Les quelques stratégies indiquées au paragraphe 2.2 ci-dessus sont assez naïves. Nous apprécierons les programmes qui jouent bien, et n'hésitez pas à passer du temps à affiner vos stratégies [1] .

On peut généraliser la stratégie de *recherche de coup gagnant de profondeur 0*, décrite plus haut, à la *recherche de coup gagnant de profondeur 1*. On dit qu'une colonne est *gagnante en un coup* si, lorsque le joueur p joue cette colonne au coup n, quel que soit le coup joué par le joueur q au coup $n + 1$, le joueur a un coup gagnant au coup $n + 2$. Dans la stratégie de recherche de profondeur 1, l'ordinateur joue un coup gagnant en zéro coups s'il en exite un ; sinon, il joue un coup gagnant en 1 coup ; et sinon, il joue au hasard.

On peut améliorer les stratégies de recherche bornées en évitant les coups perdants. Un coup est *perdant en 0 coups* pour le joueur p s'il permet au joueur q de jouer un coup gagnant en 0 coups ; de même, un coup est *perdant en 1 coup* pour p s'il permet au joueur q de jouer un coup gagnant en 1 coup. La stratégie de *recherche de profondeur 0* joue un coup gagnant en 0 coups s'il existe ; sinon, elle choisit un coup au hasard mais en évitant les coups perdants en 0 coups.

On peut aller encore plus loin en prévoyant plus de 0 ou 1 coup d'avance.

Chacune des stratégies ci-dessus contient un élément aléatoire ; on peut les améliorer avec des *heuristiques*, c'est à dire des règles qui disent quelle position il vaut mieux jouer. Par exemple, il est avantageux de jouer dans la colonne centrale (tant qu'elle n'est pas beaucoup plus haute que les autres colonnes), et il est bien sûr avantageux de maximiser la longueur de la plus longue ligne de pions.

Comparaison de stratégies La plupart de vos stratégies contiendront un élément aléatoire. Il est donc intéressant de faire des statistiques sur le comportement des stratégies. Votre programme pourrait par exemple avoir une option supplémentaire qui demande à l'utilisateur de choisir une stratégie pour rouge et une stratégie pour jaune, simuler 10000 parties, et indiquer combien de

1. Il a été démontré qu'il existe une stratégie pour le joueur rouge qui lui permet de gagner à chaque fois.

fois chaque joueur gagne. Un autre problème intéressant est de déterminer empiriquement [2] si une stratégie est meilleure pour rouge ou pour jaune.

Généralisations Vous pourriez bien sûr généraliser votre programme à des jeux plus intéressants. Une géneralisation facile mais peu intéressante est d'augmenter le nombre de colonnes et de lignes (les parties prendront alors plus de temps, mais la stratégie reste *grosso modo* la même).

Score Four ou *Sogo* est une généralisation de *Puissance 4* à trois dimensions : le jeu se joue sur un plateau constitué de 4×4 tiges verticales pouvant chacune contenir 4 pions. Il n'est pas forcément facile d'afficher le plateu de Sogo de façon compréhensible.

4 Modalités de soumission

Le projet sera à traiter en groupes de 2 personnes au plus (les projets soumis par des groupes de 3 personnes ou plus ne seront pas acceptés). Votre solution devra consister d'un programme écrit en Java et utilisable sous Linux ou FreeBSD. Elle consistera de :

— un fichier texte nommé README contenant vos noms et indiquant brièvement comment compiler et utiliser votre programme ;

— un petit rapport, en format texte, décrivant ce que vous avez fait, quelles extensions vous avez traitées, et expliquant (ou justifiant) les choix de conception ou d'implémentation que nous pourrions ne pas comprendre du premier coup ;

— votre code source

Votre soumission devra consister d'une seule archive compressée `zip`. L'archive devra *obligatoirement* s'appeler *`nom1-nom2`*`.zip`, et s'extraire dans un répertoire *`nom1-nom2/`*, où *`nom1`* et *`nom2`* sont les noms des deux personnes constituant le groupe. Par exemple, si vous vous appelez Ben Affleck et Rosamund Pike, votre archive devra s'appeler `affleck-pike.zip` et s'extraire dans un répertoire `affleck-pike/`.

La date limite et le mode de soumission seront indiqués sur la page du projet.

Si jamais vous étiez amené-e à utiliser du code qui n'a pas été écrit par vous, il est absolument impératif que vous l'indiquiez explicitement dans votre projet, en entourant le code en question d'un commentaire expliquant sa provenance. Ce morceau de code sera pris en compte différemment dans la notation – mais reprendre du code sans citer explicitement son auteur est un plagiat qui sera considéré comme de la triche.

2. En faisant des expériences.