Projet logiciel : développement d'un Solitaire en Java

Dittrick Gabriel et Dubief Louis-Alexis

March 24, 2017

Contents

Introduction

Le solitaire est un jeu de plateau qui, comme l'indique son nom, se pratique seul. Le joueur déplace des pions sur un plateau dans le but de n'en avoir plus qu'un seul, de préférence placé au centre. Au départ Toutes les cases sont remplies sauf une, généralement celle du milieu. Il est a noté que le solitaire utilisé pour développer le logiciel est un solitaire de type anglais. Il existe d'autres formes

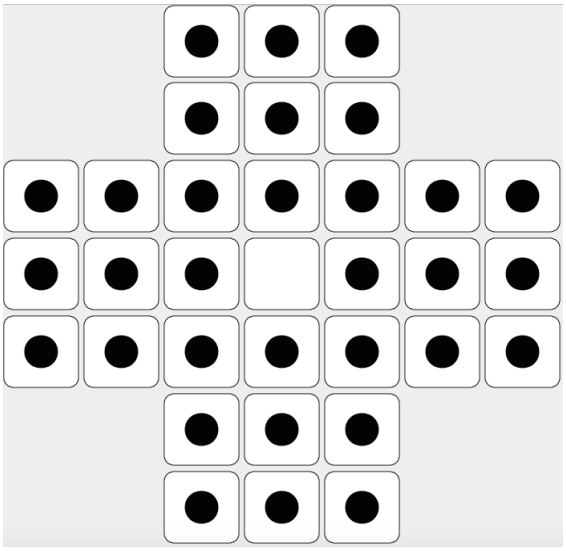

Figure 1: Plateau Solitaire

de plateau. Pour supprimer des pions, il faut que deux pions soient adjacents et suivis d'une case vide. Le premier pion saute par-dessus le deuxième et rejoint la case vide. Le deuxième pion est alors retiré du plateau. Un pion ne peut sauter qu'horizontalement ou verticalement, et un seul pion à la fois.

1 Conception

Il s'agit dans cette partie de détailler une des phases les plus importante dans le développement d'un logiciel, la phase de conception. Cette étape permet de répertorier les étapes de fonctionnement et de calcul du logiciel et de détacher plusieurs scénarios notamment deux types de scénarios :

- Le scénario d'utilisation normal
- Les scénarios pouvant conduire à des erreurs

1.1 Description du problème

Afin de concevoir ce jeu nous devons répertorier tous les éléments qui pourront nous aider derrière à coder le logiciel

1.1.1 Scénario Normal

Le joueur arrive sur le plateau qui est dans le même état que sur la figure 1. C'est à dire avec que des cases contenant un pion sauf celle au centre qui est vide. A ce moment la le joueur est en mesure de cliquer sur le pion qu'il veut

déplacer. Quand ceci est fait, le logiciel regarde si à partir de cette case choisie, un coup est possible. Dans ce cas la case est donc dans un état actif et devient blanche. Le joueur peut alors cliquer sur la case vide sur laquelle le joueur veut déplacer le pion. De la même manière, le logiciel test si le coup est possible puis effectue le coup.

1.1.2 Scénario "erreur"

On répertorie toutes les erreurs possible afin de concevoir un logiciel le plus performant possible :

- erreur lors de la sélection de la case à déplacer, deux erreurs possible :
  - case vide
  - case qui conduira à une combinaison non fructueuse

    Dans ces deux cas c'est simple puisqu'il suffit que le logiciel fasse comme si de rien n'était et qu'aucune case ne s'active. Dans ce cas, le joueur sait qu'il n'a pas sélectionné une bonne case et donc il réitère son opération

- erreur lors de la sélection de la deuxième case, deux erreurs possible :
  - case pleine
  - case qui ne conduira pas à une combinaison fructueuse en regard de la case qui est active

    Dans ces deux cas 2 solutions sont envisageable, un cahier des charges plus détaillé aurait permis de choisir la méthode la plus adaptée :

    * Le logiciel fait comme si de rien n'était et laisse la case active dans l'état actif, le joueur sait donc qu'il s'est trompé de case et sélectionne à nouveau l'endroit où il veut déplacer le pion actif
    * ou alors si le joueur a sélectionné une case pleine, dans ce cas c'est sûrement qu'il a envie de changer la case à déplacer (plusieurs possibilité s'offre au joueur sur le plateau) et dans ce cas le logiciel désactive l'autre case et active celle la (si cette case est légale bien entendu). Maintenant le joueur peut sélectionner la case dans laquelle il veut déplacer le pion.

      c'est ce dernier choix que nous retiendrons.

1.2 modélisation

Afin de faciliter le développement du logiciel, nous avons choisi de nous baser sur la simplification du modèle MVC (Model View Controler) avec seulement une partie modèle et une partie Vue. Le principe et le suivant : Nous avons une classe Case et une classe Plateau qui va être en fait un tableau de plusieurs cases. La case Plateau contient toutes les méthodes nécessaires à la mise à jour du logiciel ainsi que toutes les méthodes de calcul (que nous verrons plus tard pour ce qui concerne l'intelligence artificielle). Nous avons aussi une classe Vue et une classe GameWindow. La classe GameWindow prend en paramètre une instance de la classe Vue qui elle même prend en paramètre une instance de la classe Plateau. Cette classe Vue va représenter de manière Graphique, l'état du Plateau (l'état du modèle) et se mettre à jour régulièrement. La classe GameWindow est une simple fenêtre (implémentant l'interface JFrame) dont le contentPane est une instance de la classe Vue. Grâce à cette modélisation il est facile ensuite de développer dans le modèle toutes les méthodes nécessaires au fonctionnement de l'intelligence artificielle.

2 Développement du logiciel

Cette partie concerne la réalisation du logiciel en lui même privé de son intelligence artificielle. Le but est donc de développer à ce stade un logiciel permettant à un utilisateur de jouer au solitaire avec la souris.

2.1 Le modèle

Comme dit précédemment, le modèle contient deux classes, une classe Case et une classe Plateau.

- La classe Case est très simple puisqu'elle contient seulement 3 attributs qui sont passé en argument du constructeur. Ces attributs privé sont : value, i, j qui sont respectivement la valeur, la ligne et la colonne de la case correspondante. Elle contient bien évidemment les méthodes basiques que sont les assesseurs (get) et les modificateurs d'attributs (set).

- La classe Plateau elle ne prend pas d'argument dans son constructeur. Le constructeur se contente de créer un tableau à deux dimensions (7*7) composer de 49 instances de "Case". Il contient donc deux attributs : le tableau de case et un attribut de type "Case" appelé caseActive lui permettant de savoir si oui (et donc laquelle) ou non il possède une case active. La convention suivante a été prise. Chaque case contient une valeur qui est fortement susceptible de changer plusieurs fois au cours d'une partie. voici la convention :

- 0 : case vide
- 1 : case pleine
- 2 : case active
- 3 : case interdite

La classe Plateau contient plusieurs méthodes permettant de réaliser divers calculs comme :

- isCase1Legal(int,int) permettant de savoir si en la case passée en argument permettra d'effectuer une combinaison (elle renvoi true ou false)
- isCase2Legal(int,int,int,int) souvent appelle quand isCase1Legal a renvoyé true, elle permet de savoir si le coup passé en argument est possible (elle renvoi true or false). Un coup est composé de 4 entiers les deux premiers représentent respectivement la ligne et la colonne de la première case (celle sélectionnée par l'utilisateur) et les deux derniers entiers correspondent respectivement à la ligne et la colonne de la deuxième case ou case d'arrivée. (ce sera souvent un tableau de 4 entiers et donc un tableau de coups sera une matrice de taille nbCoup*4)
- si isCase2Legal(int,int,int,int) renvoie true dans ce cas on peut appeler la méthode playMove(int, int, int ,int) qui va jouer le coup ou la combinaison en modifiant l'état des cases impliquées
- getCoup() permettant de savoir les coups possible selon la configuration du plateau (elle utilise les deux méthodes isCase1Legal(int,int) et isCase2Legal(int,int,int,int)
- et bien d'autres encore

2.2 La vue

La vue est représentée par deux classes, une classe GameWindow et une classe Vue.

- La classe GameWindow hérite de la classe JFrame, ce sera la fenetre de notre jeu
- La classe Vue elle hérite de la classe JPanel. Ce sera un panneau sur lequel on va peindre notre plateau. Ce sera la contenPane de notre classe GameWindow.

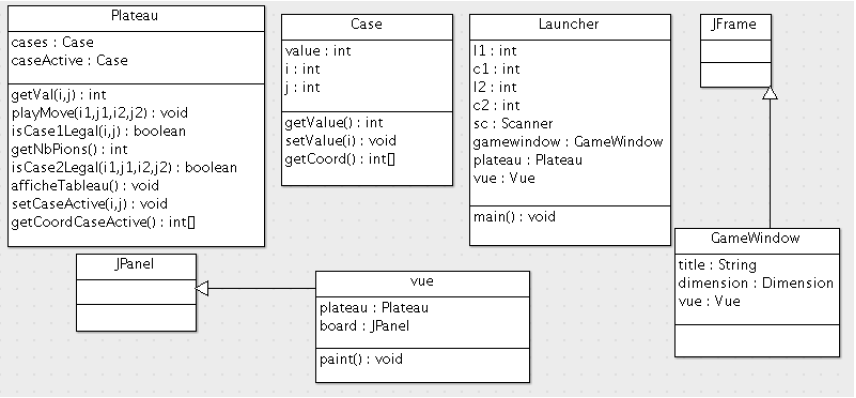

Figure 2: Diagramme UML

Chacune des ces classes comporte la méthode paint(Graphics g) qui va permette lorsqu'il y a changement d'état de la fenetre ou lorsqu'il y a appel de la méthode repaint, de se modifier. Ainsi La méthode paint() de la classe Vue actualise l'affichage du plateau en dessinant les bonnes images au bon endroit sur le JPanel selon les valeurs des cases dans l'objet plateau du modèle. La méthode paint de la classe GameWindow se contente d'appeler la méthode paint() de la classe Vue grâce à la méthode repaint(). Nous avons pu gérer le redimensionnement de fenêtre ainsi il est possible de redimensionner la fenêtre à l'aide de la souris sans générer d'erreurs d'affichage

3 Intelligence artificielle

Dans toute cette partie nous allons présenter nos travaux sur la réalisation d'une intelligence artificielle pour le solitaire. nous avons choisi de nous baser sur l'algorithme de MonteCarlo qui peut s'implémenter assez facilement dans un problème et qui donne de très bon résultat.

3.1 Principe de l'algorithme de MonteCarlo

3.1.1 Présentation

L'algorithme de MonteCarlo se base sur des calculs statistiques afin de trouver la solution qui semble être la meilleur. Dans notre cas, l'algorithme de MonteCarlo par du premier plateau, le plateau avec des pions partout et un trou au centre. A partir de la il y a 4 coups possible. L'algorithme va donc lancer à partir de ces 4 configurations de plateaux possible plein de partie aléatoire (un nombre que l'on pourra modifier pour voir l'influence du nombre de parties aléatoires jouées sur les résultats de l'algorithme) et faire une moyenne des pions restant à la fin de chaque partie. Ainsi il est facile de déduire que la configuration de

plateau donnant la plus petite moyenne est probablement la configuration de plateau qui donnera le meilleur résultat possible. Donc l'algorithme joue le coup qui donne cette configuration de plateau et ainsi de suite jusqu'à que le jeu soit bloqué.

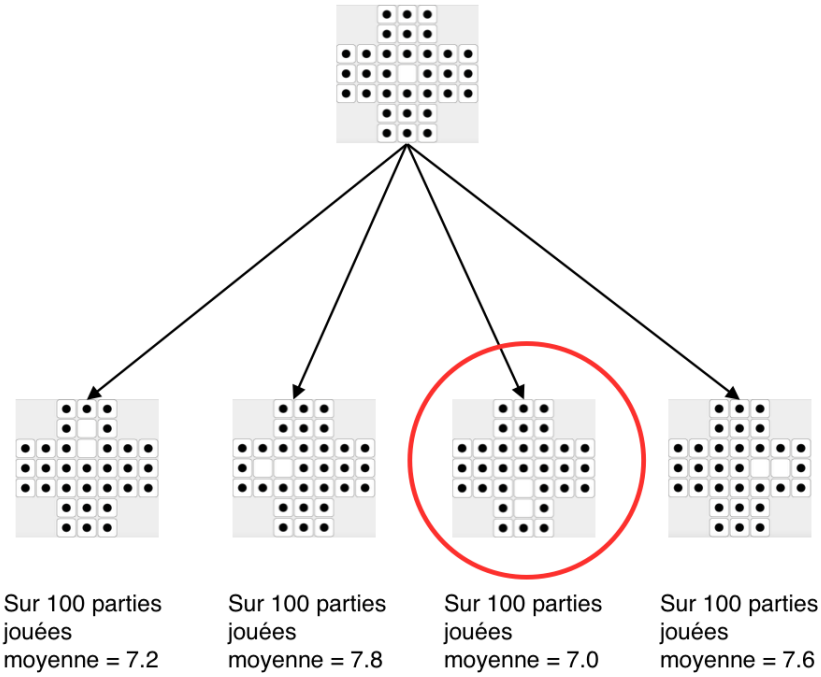

Figure 3: Principe algorithme de MonteCarlo

On voit bien sur la figure 3 que c'est le plateau 3 du niveau 1 qui semble donner les meilleur résultats ainsi à partir du plateau de niveau 0 c'est le coup 3 qu'il faut jouer (il faut se "diriger" vers ce plateau).

3.1.2 Résultats

Algorithme basé sur la moyenne

Afin de mesurer les performances de l'algorithme de Monte Carlo que nous avons développé, nous avons décider de le comparer à un algorithme purement aléatoire. Le principe est que l'on réalise 100 parties de MonteCarlo, on récupère les résultats et on fait une moyenne du nombre de pions restant, nous avons aussi fait de même sur 100 parties aléatoires. A chaque fois les résultats sont donnés

avec un intervalle de confiance permettant d'estimer la validité de la solution. On obtient le résultat figure 4.

```
moyenne pions restants partie aleatoire : 7.54
Intervalle de confiance : [4.094190951314916 ; 10.985809048685084]

moyenne pions restants partie Monte Carlo basé sur la moyenne : 2.88
Intervalle de confiance : [1.9731372760996295 ; 3.78686272390037]
```

Figure 4: Résultats Monte Carlo basé sur la moyenne

L'algorithme de MonteCarlo utilisé se base sur une moyenne de 100 parties aléatoires réalisées sur chaque coups possible afin de récupérer le meilleur coup. Le paramètre valant 100 ici sera appelé nbPartieAleatoire.

100 parties aléatoires donnent en moyenne 7.54 pions restant à la fin d'une partie avec un intervalle de confiance à 95% de [4.09 ; 10.98] soit une amplitude de 6.89

100 parties de MonteCarlo donnent en moyenne 2.88 pions restants à la fin d'une partie avec un intervalle de confiance à 95% de [1.97 ; 3.78] soit une amplitude de 1.81.

On observe que ces deux intervalles de confiances ne se chevauchent pas, on est donc sur à 95% que la moyenne obtenue par MonteCarlo est inférieure à celle obtenue par un algorithme aléatoire.

Ce qui prouve au vu des résultats l'efficacité de MonteCarlo. De plus sont amplitude de confiance est plutôt réduite. On se proposera par la suite d'étudier l'influence du paramètre nbPartieAleatoire sur cet intervalle de confiance.

algorithme basé sur le minimum

Il a été dit que l'algorithme de MonteCarlo que l'on utilise base son choix de coup sur une moyenne de coups aléatoire. Il joue le coup qui en moyenne donne les meilleurs résultats aléatoire. Il est possible de baser l'algorithme sur le principe du minimum. C'est à dire que l'algorithme pourrait jouer le coup qui de manière aléatoire a donné le minimum de pions restant. On obtient les résultats figure 5.

On peut voir que l'on obtient des résultats très proches, une moyenne de pions restant à 2.64 et une amplitude de confiance de 2.29.

```
moyenne pions restants partie aleatoire : 7.63
Intervalle de confiance : [4.778263686804126 ; 10.481736313195874]

moyenne pions restants partie Monte Carlo basé sur le minimum : 2.64
Intervalle de confiance : [1.4903913709440069 ; 3.7896086290559934]
```

Figure 5: Résultats Monte Carlo basé sur le minimum

3.1.3 Complexité

Le but de cette partie est d'étudier la complexité de l'algorithme de MonteCarlo basé sur la moyenne des parties aléatoires. Pour ce faire nous créons un programme qui réalise une boucle à l'intérieur de laquelle on joue une partie de MonteCarlo en changeant à chaque fois le paramètre nbPartieAleatoire. (paramètre expliqué précédemment qui représente le nombre de partie aléatoire utilisée pour calculer le meilleur coup. Il est par défaut à 100). A chaque fois on calcul le temps qu'il faut à l'algorithme pour effectuer une partie de MonteCarlo. ainsi il est possible de tracer le temps de calcul en fonction du paramètre nbPartieAleatoire (voir figure 6)

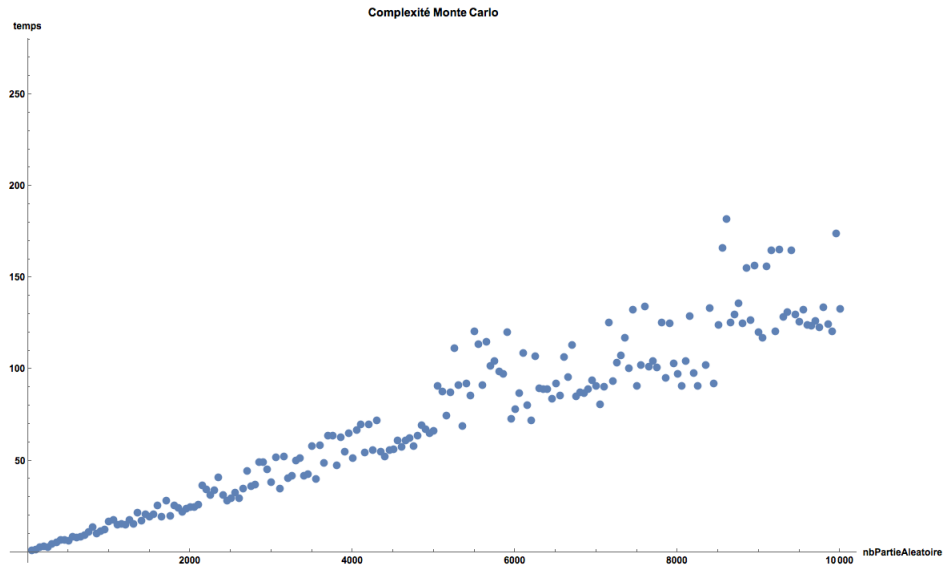

Figure 6: Étude de la complexité de Monte Carlo

On observe que le graphique de complexité de Monte Carlo semble être linéaire avec toute fois des valeurs peut significative lorsque nbPartieAleatoire est très grand. Il semble donc que la complexité de notre algorithme soit en $\alpha.O(n)$.

3.2 Étude de l'influence du paramètre nbPartieAleatoire sur la performance de Monte Carlo

Dans cette partie nous nous proposons d'étudier l'influence du paramètre nbPartieAleatoire que nous pourrons noter N sur la moyenne du nombre de pions restant obtenue par Monte Carlo ainsi que l'intervalle de confiance à 95 %.

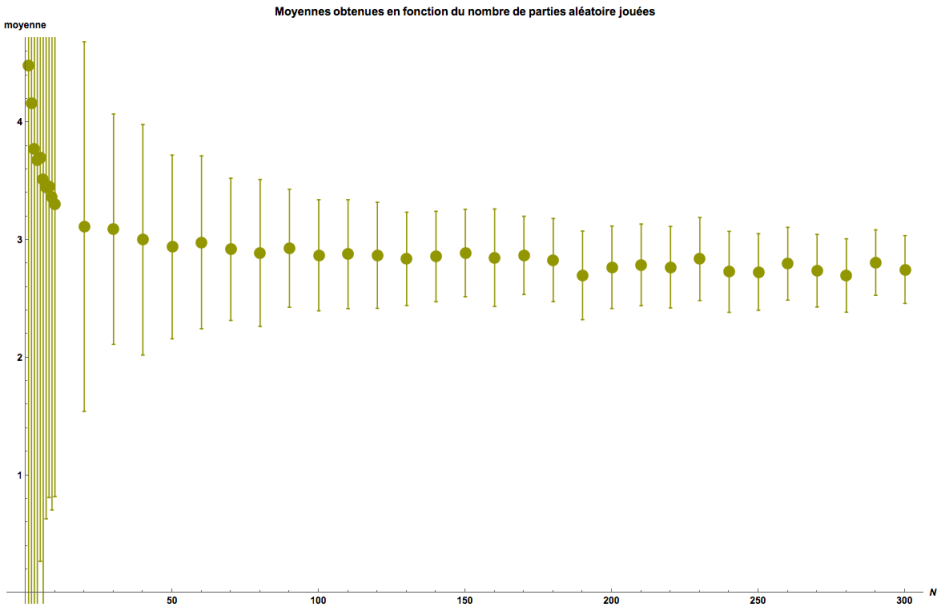

Figure 7: Influence du paramètre N sur la performance de Monte Carlo

On s'aperçoit de manière logique que lorsque l'on augmente le nombre de parties aléatoires (quand on augmente N) jouée sur un coup pour déterminer quel coup semble être le meilleur, l'algorithme mettra plus de temps (cf figure 6) à calculer cependant il gagnera en performance (moins de pions restant à la fin donc meilleur partie) et en répétabilité (écart type ou intervalle de confiance plutôt faible).

L'intervalle de confiance semble évoluer selon une exponentielle décroissante comme la moyenne d'ailleurs ce qui veut dire (est c'est souvent le cas dans ce genre de problème) que l'on peut atteindre une meilleur précision rapidement sans forcement augmenter drastique-ment le nombre N mais que lorsque l'on veut atteindre encore plus de précision, il faut augmenter N très fortement et donc augmenter les temps de calcul.

3.3 Recherche de la meilleur Séquence

L'objectif maintenant est de non pas seulement trouver le meilleur score que

peut obtenir l'algorithme de MonteCarlo mais d'enregistrer la meilleur Séquence qu'il puisse obtenir.

Pour cela, nous avons modifier notre fonction jouerPatieAleatoire qui jusqu'à présent renvoyait seulement le score obtenue à la fin de la partie. Maintenant elle renvoie une séquence aléatoire. Ainsi dans notre nouvel algorithme nous parcourons à partir du plateau d'un certains niveau (de niveau 0 jusqu'à la fin) tous les coups possible, dans lesquels nous effectuons une série de parties aléatoire. Si le score est meilleur que le score actuel dans ce cas on sauvegarde ce score puis on met à jour la meilleur séquence. Ainsi à chaque niveau l'algorithme va tenter de trouver la meilleur séquence possible à partir du niveau considéré et la modifier par la suite à chaque niveau.

Contre toute attente, il s'avère que les résultats ne sont pas forcément meilleur que précédemment cependant, nous avons maintenant non seulement de bons résultats mais aussi la séquence qui permet d'arriver à ce résultat.

Cependant il semble que plus on augmente le paramètre N plus cette algorithme devient performant. Ce qui est très logique mais il cette variation de performance semble plus forte que pour les Algorithmes de monte Carlo précédent (sans sauvegarde de la séquence).

```
Moyenne obtenue sur 100 parties de MonteCarlo jouees en retenant la meilleur séquence : 3.22
Avec un intervalle de confiance à 95 % de  : [2.177694862336369 ; 4.262305137663631]
```

Figure 8: MonteCarlo avec sauvegarde de la meilleur séquence

Conclusion

La réalisation du solitaire en java ce fait plutôt facilement, SWING est une bibliothèque très pratique à utiliser et il s'avère que le rendu est plutôt satisfaisant. Il est fréquent de retrouver dans la littérature des jeux de ce genre (jeu de plateau) comme le jeu de Go, du morpion etc. développer en java. Ce développement permet ainsi d'étudier le jeu en élaborant notamment des intelligences artificielles basé sur plein d'algorithme différents. En ce qui concerne le jeu du solitaire, nous avons choisi de partir sur un algorithme de Monte Carlo, algorithme qui a fait ses preuves que ce soit pour des jeux individuels tel que le solitaire ou des jeux à deux joueurs tel que le jeu de go. D'autres algorithmes sont possible, et avec un projet de plus grande envergure, il aurait été intéressant d'étudier les algorithmes génétiques qui semble petit à petit devenir de plus en plus répandu à en croire la littérature scientifique en informatique. Ce projet nous a permis de consolider nos connaissances du langage Java ainsi que le développement complet d'une interface graphique le tout en utilisant une architecture de type MVC très répandu dans le milieu du software. Nous avons pu concevoir une mini intelligence artificielle capable de donner des résultats bien plus performant qu'un simple jeu aléatoire. nous n'avons pas réussi à élaborer

une IA assez performante pour trouver une solution optimale ou une solution toute simple (reste un seul pion sur le plateau), nous avons réussi cependant à s'en approcher le plus possible avec la récupération d'une séquence amenant à seulement 2 pions sur le plateau.