PuissanceHask

Adrien DUDOUIT-EXPOSITO
Alexandre LEGOUPIL

2012-01-23

Résumé

PuissanceHask est une implémentation en Haskell du jeu de Puissance 4.

Introduction

Nous avons créer une version jouable et parfaitement fonctionnelle du jeu puissance 4 en haskell celui ci dispose d'une interface textuelle. Permettant de jouer agréablement et avec une certaine protection contre les erreurs de la part du joueur.

1 Structure du programme

Pour le programme il a été choisi une architecture de module où chaque fonction telle que l'IA, les structure, les règles de jeux, les mécanismes de jeu, et l'interface son chacun dans un module et le programme principale n'appel que le main défini dans l'interface, l'interface appel les structures et les mécanismes de jeu qui eux appel leqs règles. Ainsi il est possible de changer de type d'IA ou d'interface en une unique ligne.

2 Intelligence Artificielle

Pour l'intelligence artificiel un algorithme MiniMax a été utilisé. D'autres méthode de choix de coups on été implémenté moins couteuse on été implémenté pour pouvoir tester plus facilement comme un algorithme qui remplit toujours la première colonne disponible ou encore un algorithme qui remplit toujours la ligne la moins remplie.

2.1 Algorithme MiniMax

Wikipédia nous donne la définition suivante :

> L'algorithme minimax est un algorithme qui s'applique à la théorie des jeux pour les jeux à deux joueurs à somme nulle. Pour une vaste famille de jeux, le théorème du minimax de von Neumann assure l'existence d'un tel algorithme, même si dans la pratique il n'est souvent guère aisé de le trouver. Le jeu de hex est un exemple où l'existence d'un tel algorithme est établie et montre que le premier joueur peut toujours gagner, sans pour autant que cette stratégie soit connue.
>
> Il amène l'ordinateur à passer en revue toutes les possibilités pour un nombre limité de coups et à leur assigner une valeur qui prend en compte les bénéfices pour le joueur et pour son adversaire. Le meilleur choix est alors celui qui minimise les pertes du joueur tout en supposant que l'adversaire cherche au contraire à les maximiser (le jeu est à somme nulle).

Ainsi on construit d'abord un arbre de possibilité, où un nœud ou une feuille représente un coups et est valué. De plus chaque nœud possède un nombre de fils inférieur ou égal au nombre de colonnes dans le jeu.

Pour généré un arbre de possible on envisage chaque placement sur chaque colonne et on évalue chaque coup selon la méthode suivante :

```
si le coup n'est pas gagnant
vaut -10
si le coup est gagnant pour le joueur
vaut 100
si le coup est perdant pour le joueur
vaut -100
```

Lors de l'implémentation l'évaluation est simultané avec la construction de l'arbre :

```
nextMMTree :: Int -> Table -> Token -> (Int,Token) -> MMTree
nextMMTree diff table token (p,t)
| not (isFreeCol table p) = MMLeaf (p,t) 0
| sameToken win ( Just token ) = MMLeaf (p,t) winVal
| isJust win = MMLeaf (p,t) looVal
| diff == 0 = MMLeaf (p,t) blankVal
| otherwise = MMNode (p,t) blankVal ( constructMMTree (diff-1)
next token )
where
next = placeToken table p t
win = gameWinner next
winVal = 100
looVal = (-100)
blankVal = (-10)
```

Un fois l'arbre construit on applique a chaque nœud le traitement suivant :

```
si p est une feuille
minimax(p) = evaluer(p)
si p est un noeud Joueur avec fils O1, ..., On
minimax(p) = evaluer(p) + MAX(minimax(O1), ..., minimax(On))
si p est un noeud Opposant avec fils O1, ..., On
minimax(p) = evaluer(p) + MIN(minimax(O1), ..., minimax(On))
```

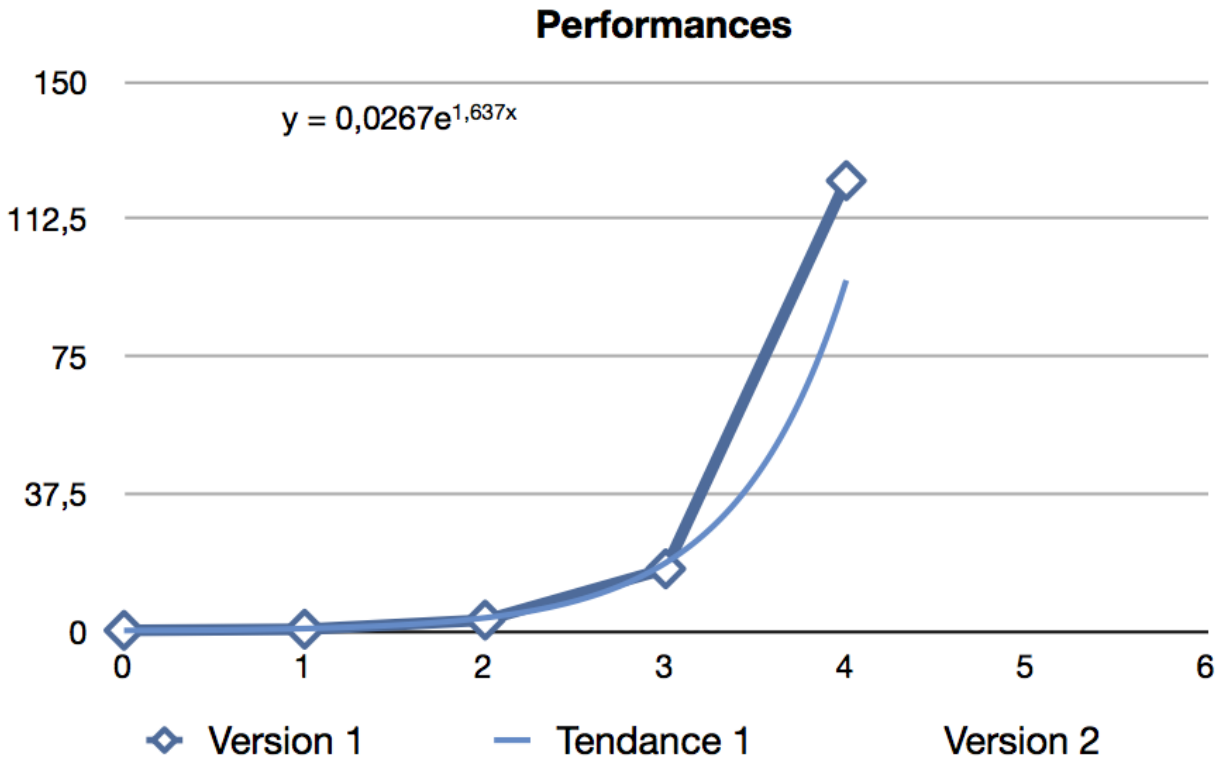
Implémenté ainsi :

```
computeMMTree :: Token -> MMTree -> (Maybe Int, Int)
computeMMTree token (MMNode (a, t) v c)
| token == t = (\(_,x) -> (Just a, v+x)) . bestCoin $
map ( computeMMTree token ) c
| otherwise = (\(_,x) -> (Just a, v+x)) . worstCoin $
map ( computeMMTree token ) c
computeMMTree _ (MMLeaf (a, _) v) = (Just a, v)
```

Où les fonction *bestCoin* et *worstCoin* sélectionnent respectivement le meilleur ou moins bon coup a jouer pour le joueur (*worstCoin* sélectionne le meilleur coup adverse).

2.2 Performances

Comme présenté sur le graphique la complexité exponentielle à un coup en temps assez élever se qui ne permet pas de faire des arbres de profondeur supérieur a 5.



2.3 Optimisation

Les principale source de calcul sont d'une part la création de l'arbre mais aussi et surtout le choix du meilleur élément qui peut être exécuté jusqu'à 79 000 fois lors d'un un appel (cas d'une recherche jusqu'à 6 coups en avance)

3 Solutions choisies

Le plateau pour des raisons pratique est un carré de 7 par 7 .Cette dimension est modifiable dans le fichier data on peut ainsi jouer sur un plateau de 100 par 100 sans soucis.
L'interface du jeux est la fenêtre shell du compilateur se qui ce révèle être un inconvénient au point de vue esthétique mais permet au final une plus grande portabilité du jeux en n'obligeant pas l'utilisateur à installer une librairie graphique spécifique.
L'interface est relativement bien protégé contre les erreur volontaire ou non de l'utilisateur.

4 Problèmes rencontrés

Au début le programme devait être doté d'une interface graphique grâce à des libraires tel que hOpenGL ou WXhaskell, toutefois nous avons été dans l'incapacité d'installer ces librairies : hOpenGL est rarement mis à jour, son installation est nébuleuse et ses tutoriel sont dépassé.
WXhaskell quand à lui dispose de plus de mise à jour mais malgré différents essais (sur windows et linux) nous avons été incapable de la rendre opérationnel.
Nous nous somme donc rabattu sur une interface shell certes moins esthétique mais totalement fonctionnel.Même si un premier jet de l'interface en wxhaskel à été crée [1].

5 Détails du jeux

- Les colonnes sont numérotés de 1 à 8 dans la configuration de base.
- Le plateau est un carré de 7 par 7 dans la configuration de base.
- Le joueur humain joue avec les croix et l'IA avec les rond.
- Le joueur commencera toujours la partie.

5.1 Régles du jeux

Le gagnant est celui qui le premier réussira a aligner sur une colonne ligne ou diagonal 4 de ses pions, si le plateau est remplis sans qu'il y ait un gagnant la partie se conclue par un match nul.

1. voir Annexe

6 mode d'emploi

Pour utiliser le jeu compilez le fichier puissanceHask et taper la commande main : le jeu est lancé . Les réglés du jeu vous sont expliquées puis une nouvelle partie est lancé avec l'affichage du plateau et l'on vous demande de donner un numéro de colonne ou placer votre jeton (exemple-ci dessous).

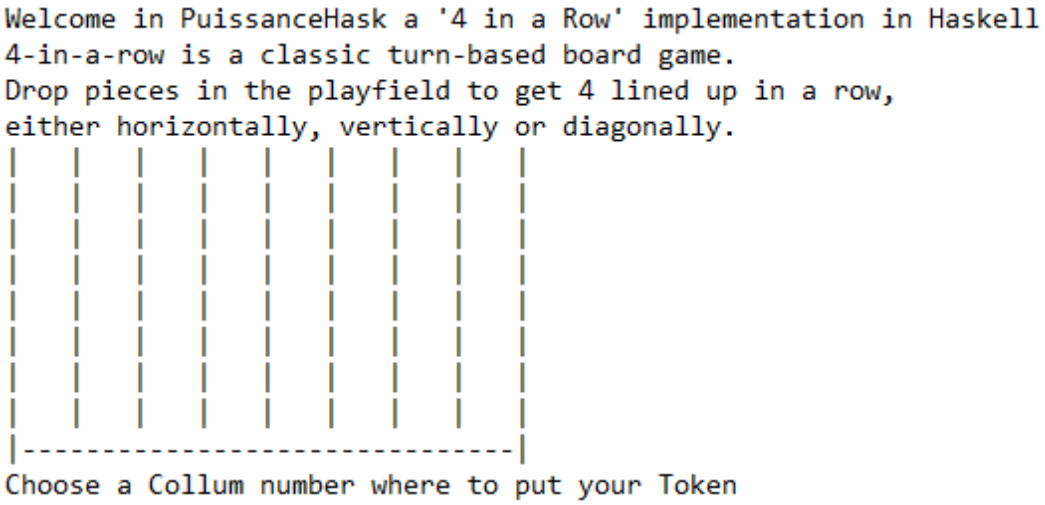


FIGURE 1 – Le jeu au lancement

Ensuite vous devez attendre que le programme affiche votre coup et celui de l'ordinateur sur le plateau,
puis on vous redemandera de donner le numéro d'une colonne(exemple-ci dessous).

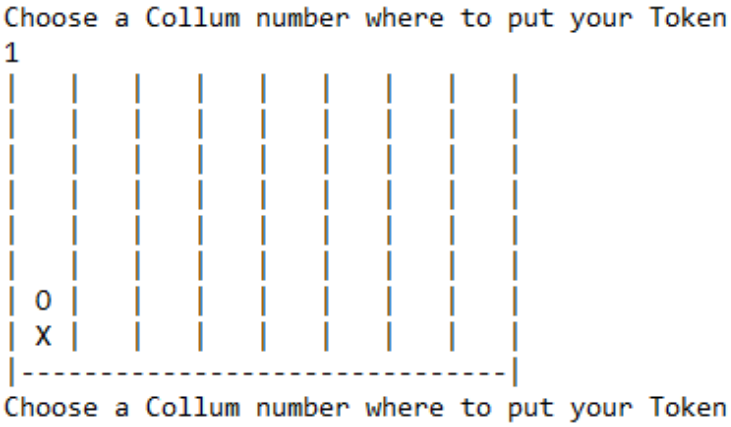


FIGURE 2 – aprés quelques coups

Et ainsi de suite jusqu'à la défaite de l'humain, de l'ordinateur ou du match nul.
Une fois la partie terminé il vous sera demandé si vous voulez rejouer si vous refusez vous quitterez le jeu.

Toute utilisation d'un caractère autre qu'un numéro entrainera la fin de la partie. Les numéros irrecevable ne seront pas pris en compte et une nouvelle chance vous serra donnée.

7 Annexe

Debut d'une interface utilisant wxHaskell

```
module Main where
import Graphics.UI.WX

-- constantes : taille des case du plateau et des pions, positionnement des balles
Tcase, maxX, maxY, maxH, radius :: Int
Tcase = 25
maxY = Tcase * 6
maxX = Tcase * 7
maxH = maxY
radius = 10
--the main function
main = start puissance4

puissance4
  = do
       game <- varCreate []

       -- creer la fenetre
       f <- frameFixed [text := "Puissance 4"]

       -- creer une zone de dessin
       p <- panel f [on paint := jeuEnCours game]

       -- creer un timer pour mettre a jour le plateau
       t <- timer f [interval := 20, on command := miseAJour game p]

       -- detecter le clic d'un joueur
       set p [on click         := jouerCoup game p]

       -- put the panel in the frame, with a minimal size
       set f [layout := minsize (sz maxX maxY) $ widget p]

          bNewGame <- button f [ text := "Nouvelle Partie", on command := newTable]
        where
        -- dessiner les pions
    jeuEnCours :: Var [[table]] -> DC a -> Rect -> IO ()
    jeuEnCours game dc viewArea
      = do pions <- varGet grille -- a modifier pour detecter la couleur des pions
           set dc [brushColor := red, brushKind := BrushSolid]
           mapM_ (drawPion dc) [p | (p:ps) <- grille]

    drawPion dc pt
      = circle dc pt radius []

      -- mettre afficher les nouveaux pions
    miseAJour :: Var [[Point]] -> Panel () -> IO ()
    miseAJour game p
      = do varUpdate game
           repaint p

      -- Recuperer la position du coup
    jouerCoup :: Var [[Point]] -> Panel () -> Point -> IO ()
```

```
jouerCoup game p pt
  = do varUpdate game (placerCoup pt:)
       repaint p

-- jouer le coup
placerCoup (Point x)
  = -- non complete
```

Références