PuissanceHask

Adrien DUDOUIT-EXPOSITO
Alexandre LEGOUPIL

2011-12-07

Résumé

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

1 Introduction

2 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.

3 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.

3.1 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)
```

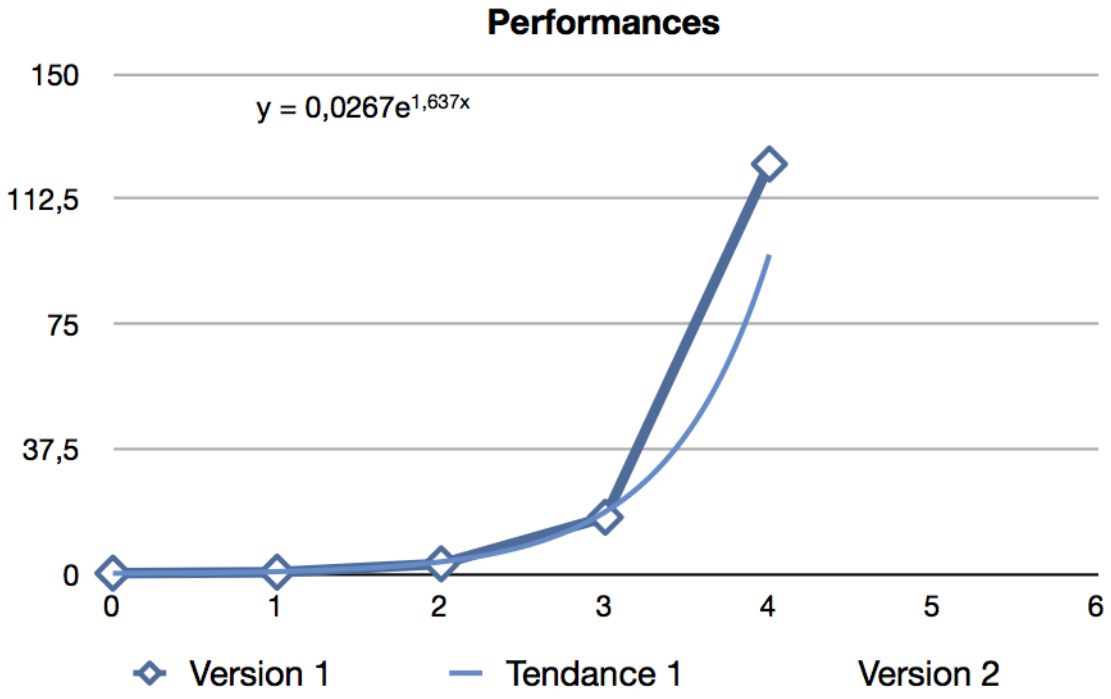
Où les fonction *bestCoin* et *worstCoin* sélectionnent respectivement le meilleur ou moins bon coup a jouer pour le joueur (*worstCoin* sélectionne le meilleur coup adverse).

3.2 Performances

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

3.3 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)


Performances
150
112,5
75
37,5
0
y = 0,0267e1,637x
0
1
2
3
4
5
6
Version 1
Tendance 1
Version 2