Jeux et Intelligence Artificielle

Monte Carlo

Slides librement inspirés de l'excellent article de Tristan Cazenave

http://www.lamsade.dauphine.fr/~cazenave/papers/jeux.pdf

Algorithmes historiques

Les plus étudiés : algorithmes de type Minimax avec coupure Alpha Beta

- Jeux à deux joueurs à somme nulle, alternés
- Développés depuis 1950 pour les échecs, jusqu'à leur apothéose (1997 : Deep Blue bat le champion du monde Garry Kasparov (1963-))
- Pas d'article "fondateur" ni de réelle paternité, mais de nombreuses améliorations:
  - Tables de transposition (certains coups commutent)
  - Iterative Deepening
  - Fenêtre de largeur minimale (ne pas partir avec alpha=-∞ et beta=+∞)
- Coupure Alpha Beta (environ ⅓ des noeuds), dépend beaucoup de l'ordre des noeuds
- Variante NegaMax la plus utilisée

Recherche Monte-Carlo

Historique : approche MiniMax excellente pour échecs (Deep Blue), mais trop limitée pour jeux plus complexes (exemple Go)

- 1998 : “Many Faces of Go” perd contre 6eme Dan amateur avec 29 pierres de handicap en utilisant une approche “classique”
- 2008 : MoGo (école française Monte-Carlo) gagne contre 8eme Dan pro avec 9 pierres de handicap
- La révolution “Monte-Carlo” est lancée avec des applications à beaucoup d’autres domaines
    - apprentissage actif, optimisation de grammaires,
    - optimisation non-linéaire, planification

Evaluation Monte-Carlo

Limitation : méthodes heuristiques classiques nécessitent une fonction d' évaluation complexe à déterminer pour un jeu comme le Go.

Solution : jouer un certain nombre de parties, aléatoirement, afin de déterminer une probabilité de gain approchée : **Monte-Carlo**

Historique : méthode de Monte-Carlo, développé par les Etats-Unis pour le projet Manhattan (bombe atomique) et auparavant utilisée comme technique d'évaluation de π.

Résultats : elles dépassent immédiatement les méthodes classiques

Calcul de π

Méthode du quart de cercle

Méthode de l’aiguille de Buffon

Comment générer des nombres (pseudo-)aléatoires ?

Aléas du hasard informatique

Algorithme 1 : Monte-Carlo

```
Entrée : une position p, un nombre n de simulations.

for i ∈ {1, . . . , n} do
    Soit p'= p.
    while p' n'est pas un état final do
        c = coup aléatoire parmi les coups légaux en p'
        p' = transition(p', c)
    end while
    if p' est gagnant pour joueur 1 then ri = 1 else ri = 0 end if
end for  // partie nulle ?


Sortie : somme(ri)/n = probabilité estimée de gain pour joueur 1
```

Amélioration : Fouille d'Arbre Monte-Carlo

Idées de l'algorithme :

- Construction incrémentale d'un arbre des coups joués : après chaque simulation aléatoire, on stocke pour chaque sommet, le nombre de victoire pour le joueur 1 et pour le joueur 2
- Utilisation de la formule dite du "bandit" pour donner un score à chaque sommet en se basant sur les valeurs stockées dans l'arbre
- On utilise les résultats des simulations précédentes pour choisir "moins" aléatoirement les nouvelles simulations

Algorithme 2

```
Entrée : une position p, un nombre n de simulations.
T ← structure vide.
for i ∈ {1, . . . , n} do
    Soit p' = p, q = ∅, partie = ∅.
    while p' n'est pas un état final do
        if p' est dans T then
            j = joueur en p'
            for c = coup légal en p' do
                p"=transition(p',c)
                Score(c)=bandit(T(¬j,p"),T(j,p"),T(j,p')+T(¬j,p'))
            end for
            c = coup parmi les coups légaux en p' maximisant Score(c)
        else
            if q = ∅ then
                if T ne contient pas q then  q = p'
                end if
                partie← partie + p'
            end if
            c =coup aléatoire parmi les coups légaux en p'
        end if
        p' = transition(p', c)
    end while
```

```
       Ajouter q dans T
   if p' est gagnant pour joueur 1
then
        ri = 1
    else
        ri = 0
    end if
    for p' in partie do
        T(ri, p') = T(ri, p') + 1
    end for
end for

Sortie : 1/n*somme(ri) = probabilité
estimée de gain pour joueur 1
```

Choix de la formule du “bandit”

- UCT (Upper Confidence Tree) :
  - utilisée pour le Go
  - K : constante empirique, n : nombre de simulations pour le sommet,
  - v : nombre de victoires et d : nombre de défaites
  - (v+d)/d : terme d’exploitation (favorise les coups ayant un bon taux de succès)
  - second terme (racine), favorise l’exploration
  - si (v+d)=0, initialisation par une constante

$$bandit(v,d,n) = v/(v+d) + \sqrt{((K\log(n)/(v+d)))}$$

- RAVE (Rapid Action Value Estimates):
  - v’ (resp d’) : nombre de victoires (resp. défaites) où le coup a été joué en premier par le joueur courant puis par son adversaire
  - α(.): fonction tendant vers 1, on utilise v’ et d’ qu’après un certains nombre de simulations

$$bandit(v,d,v',d') = \alpha(v+d)v/d + (1-\alpha)v'/d'$$

- Heuristique + BDD:
  - h(p’,c): heuristique estimant la fréquence de jouer le coup c en position p’ dans cette situation (utilisation d’une BDD de situations)

$$bandit(v,d,p',c) = v/(v+d) + Kh(p',c)/(v+d)$$

Conclusion partielle

- Monte Carlo + Fouille Arbre : méthodes optimales jusqu'au début des années 2010 pour les jeux complexes de type Go
- Permet le passage à l'échelle et le parallélisme massif
- Seul problème, biais liés au caractère aléatoire, parfois limitant

Objectif principal : réduire ce biais, ce qui a été réalisé par les méthodes d'apprentissage par renforcement à partir des années 2010.

Nous verrons ces techniques d'apprentissage au prochain cours.

Et pour les jeux à 1 joueur ?

Comme nous l'avons déjà vu, les jeux à 1 joueur peuvent être résolues par l'algorithme A* (Puzzle, Rubik's Cube, Taquin, Sokoban, Jeux vidéos, ...)

A chaque fois, il faut déterminer une heuristique admissible (inférieure au nombre de coup réel), calculée à chaque position (exemple de la distance de manhattan pour sortir d'un labyrinthe)

Lorsqu'aucune "bonne" heuristique n'est connue, possibilité d'utiliser **Monte-Carlo** (Sudoku, Kakuro, Morpion, Solitaire).