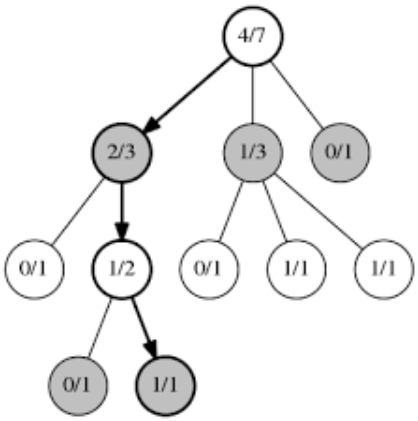
Projet C# - M2 ISIFAR

Lucien Mauffret et Yoann Pruvot

Décembre 2019



1 Introduction

Dans le cadre du Master 2 ISIFAR, pour le cours de C#, il nous a été donné l'occasion d'étudier l'implémentation de l'algorithme Monte Carlo Tree Search (MCTS), ou Recherche Arborescente de Monte Carlo, un algorithme de prise de décision. Il est notamment utilisé dans des jeux où le nombre des coups possibles est limité, afin de créer des adversaires artificiels intelligents. Deux jeux nous sont proposés pour pouvoir l'appliquer : le jeu des allumettes et le puissance quatre. L'algorithme se divise en quatre phases (les deux premières sont sous le même nom dans le code: *Sélection*) :

- Sélection : A partir d'un noeud, on sélectionne successivement les positions filles jusqu'à atteindre les dernières positions filles possibles (le choix se fait par une fonction de "choix" explicité plus bas :

$$\phi(a, W, C) = \frac{a + W}{a + C}$$

- Expansion : Si cette position fille n'est pas finale, on étant notre arbre et on continue à explorer notre arbre.
- Simulation : On explore la fin du graphe en simulant une partie aléatoirement jusqu'à atteindre une position terminale.
- Rétro-propagation : On injecte le résultat de la partie simulé sur chaque noeud du graphe parcouru lors de la selection.

2 Question préliminaires : 1 - 5

Ces questions correspondent à la mise en place du jeu :

- des allumettes pour les joueurs et JMCTS
- du puissance quatre pour les joueurs et JMCTS

Il nous a semblé simple de faire jouer deux humains ensemble; la difficulté se trouve dans le fait de s'adapter à la classe JMCTS. Il fallait aussi faire attention à l'implémentation des vérifications en ligne, en colonne et en diagonale pour le puissance quatre. Enfin, une subtilité résidait dans la prise en compte des colonnes éventuellement pleines, toujours pour le puissance quatre, impliquées dans la compréhension du paramètre i dans la classe *PositionP4* qui représente la i$^{\text{ème}}$ colonne vide.

3 Réutilisation de l'arbre calculé et détermination empirique

Il nous est demandé désormais d'améliorer le joueur dirigé par MCTS (que l'on appelle JMCTS) afin qu'il devienne plus performant, autrement dit que l'algorithme garde en mémoire ce qui a été calculé lors des parcours de graphe réalisés auparavant afin de prendre une meilleure décision. Pour cela, nous avons modifié la classe *PositionP4* et utilisé la méthode *Equals* afin de comparer deux positions et choisir le noeud fils à garder. C'est-à-dire celui dont la position rattachée est la même que la position courante et donc déterminer la bonne branche de l'arbre dans laquel a joué l'adversaire. Un des problèmes auquel nous avons fait face a été de savoir comment initialiser notre position, chose que l'on a résolu en initialisant le noeud racine à $Null$ lors du lancement d'une nouvelle partie puis en l'initialisant à un Noeud ayant pour père $Null$ et pour position la position courante. On se propose de montrer, pour une meilleure visibilité, la partie de code concernée :

```
public override int Jouer(Position p)
{
    sw.Restart();
    Func<int, int, float> phi = (W, C) => (a + W) / (b + C);
    if (racine == null)
    {
        racine = new Noeud(null, p);
    }
    else
    {
        racine = racine.MeilleurFils();
        for (int i = 0; i < racine.p.NbCoups; i++)
        {
            if ((racine.fils[i] != null) && (racine.fils[i].p.Equals(p)))
            {
                racine = racine.fils[i];
                break;
            }
        }
    }
```

Figure 1: Caption

Rappelons que MCTS dépend d'un paramètre $a > 0$ qui intervient dans le choix de la position fille (où $\frac{W+a}{C+a}$ est maximum). La question devient alors de savoir, pour le jeu du puissance quatre, quelle est une valeur optimale de a ? On essaye donc d'y répondre en organisant un championnat. On prend 100 joueurs avec des valeurs différentes de $a \in [1, 100]$, et tous vont jouer les uns contre les autres deux fois, une fois en commençant et une fois en étant deuxième. On considèrera alors que celui qui a le plus de victoire est meilleur, et ainsi on arrivera à trouver empiriquement un a optimal pour le jeu du puissance quatre.

SNous organisons le tournoi pour le jeu du puissance quatre car l'organiser pour le jeu des allumettes n'aurait pas eu de sens. En effet, la stratégie à adopter pour gagner est relativement simple. Il suffit de laisser 1 modulo 4 allumettes à l'adversaire après chaque coup pour gagner. Ainsi en commençant, le joueur est certain de gagner et les résultats du tournoi n'auraient pas été explicatifs.
On a pris $a \in [0, 100]$. Dans un premier temps, nous n'avions pas réussi la question 6). Nous avons donc effectué le procédé suivant et nous avons trouvé comme paramètre optimale : $a = 10$, avec un tournoi à 100. Voyant que le nombre était proche de 10, par soucis de temps de calcul (huit heures contre deux heures environ), une fois la question 6 répondu, nous avons appliqué le même procédé avec $a \in [0, 50]$. Ainsi, on obtient le graphe suivant:

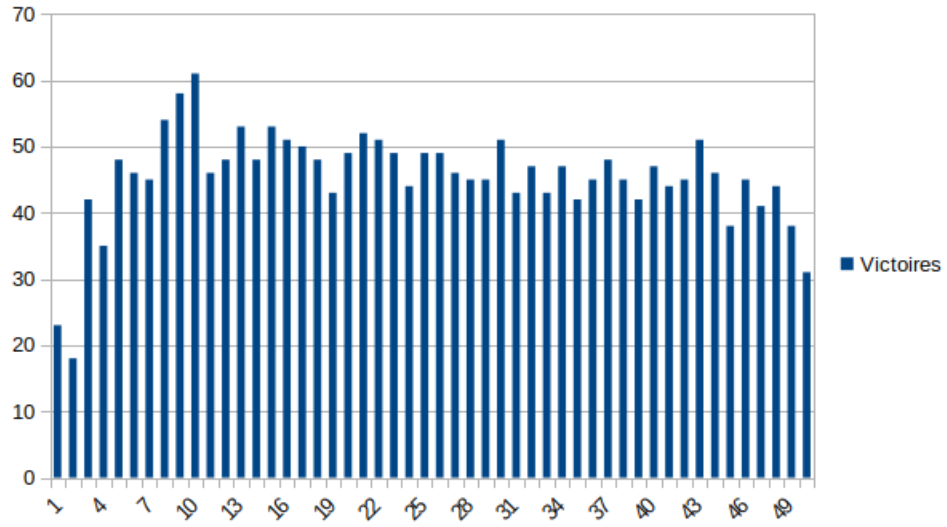


Figure 2: Nombre de victoires de JMCTS selon les valeurs de a

On en déduit un paramètre optimal :

$$a = 10$$

4 JMCTSp et JMCTSP: Threading et optimisation

On cherche désormais à savoir si l'on ne pourrait pas améliorer JMCTS lors de ses recherches de coups possible en parallélisant certaines phases de l'algorithme. On cherchera donc dans un premier temps à paralléliser la phase de simulation (JMCTSp), puis dans un deuxième temps toutes les phases (JMCTSP).

4.1 Multi-threading de la simulation

Une fois cette parallélisation effectuée, on se pose la question de savoir, pour un temps de réflexion de $100ms$ et un nombre de threads possibles $= 4$, qui de JMCTS ou JMCTSp est le meilleur, et de la même manière que précédement, on se demande quel est le paramètre optimal a. Les observations sont les suivantes :

- de la même manière que pour le JMCTS (avec un championnat), on trouve pour JMCTSp un paramètre optimale $a = 11$.

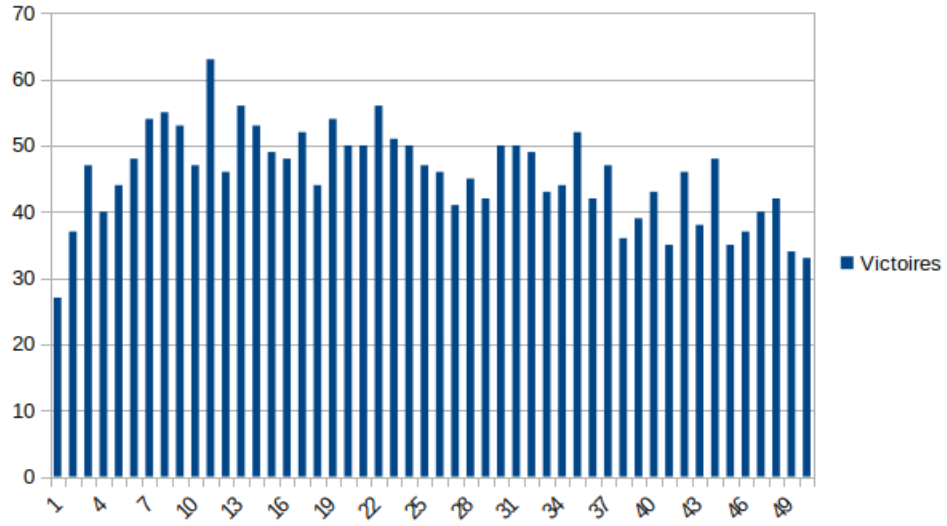


Figure 3: Nombre de victoires de JMCTSp selon les valeurs de a

- à temps de réflexion égal ($100ms$), avec 4 threads possibles pour JMCTSp, JMCTSp est légèrement meilleur que JMCTS; pour 100 matchs joués (en alternant les joueurs qui débutent), on a un score de :
  - $a = 1$ pour les deux joueurs : 30 à 65 pour JMCTSp.
  - $a = 10$ pour les deux joueurs : 46 à 47 pour JMCTS.
  - $a = 11$ pour les deux joueurs : 52 à 40 pour JMCTSp.
  - $a = 10$ pour JMCTS et $a = 11$ pour JMCTSp : 53 à 39 pour JMCTS.

4.2 Multi-threading: deuxième approche

Comme précédemment, on se pose deux questions : qui de JMCTS, JMCTSp ou JMCTSP est le meilleur pour un temps de réflexion de $100ms$ et un nombre de threads $= 4$, et quel est le paramètre optimal a pour JMCTSP. On a remarqué que JMCTSp est meilleur que JMTCS pour $100ms$ (vu précédemment), on se pose

alors la question de savoir qui est meilleur entre JMCTSp et JMCTSP dans un premier temps. Nous avons malheureusement pas réussi à implémenter le JMCTSP; nous pensons toutefois être très proche du but, nous nous permettons de vous proposer notre réflexion relative à cette implémentation.
Le but étant de lancer plusieurs threads afin d'actualiser $win[i]$ et $cross[i]$ (issu de la classe NoeudP que l'on vous laisse le soin de regarder directement dans le code), on a décidé de mettre sous forme de fonction toute la partie suivante :

```
public void Calcul(NoeudP no, Func<int, int, float> phi, int i
{
    do // Sélection
    {
        no.CalculMeilleurFils(phi);
        no = no.MeilleurFils(i);
    } while (no.cross[i] > 0 && no.fils.Length > 0);


    re[i] = JeuHasard(no.p, i); // Simulation

    while (no != null) // Rétropropagation
    {
        no.cross[i] += 2;
        no.win[i] += re[i];
        no = no.pere;
    }
}
```

Figure 4: Fonction *Calcul* de JMCTSP

Ainsi, on obtient la nouvelle fonction *Jouer* :

```
public override int Jouer(Position p)
{
    sw.Restart();
    Func<int, int, float> phi = (W, C) => (a + W) / (b + C);

    racine = new NoeudP(null, p, this.N);
    int iter = 0;
    while (sw.ElapsedMilliseconds < temps)
    {
        this.TaskList = new List<Task>();
        NoeudP no = racine;
        for (int i = 0; i < this.N; i++)
        {
            int j = i;
            TaskList.Add(Task.Run(() => Calcul(no, phi, j)));
            iter++;
        }

        Task.WaitAll(TaskList.ToArray());
    }
    racine.CalculMeilleurFils(phi);
    Console.WriteLine("{0} itérations", iter);
    Console.WriteLine(racine);
    return racine.indiceMeilleurFils;

}
```

Figure 5: Fonction *Jouer* de JMCTSP

Malheureusement, nous avons rencontré des problèmes, ce joueur ne semble pas "intelligent" comme il devrait l'être, jouant toujours dans la même colonne.

5 Conclusion

Au cours de ce projet, nous avons réussi :

- dans un premier temps à intégrer au sein de deux jeux différents, le jeu des allumettes et le jeu du puissance quatre, l'algorithme Monte Carlo Tree Search afin d'avoir une intelligence artificielle de prise de décision performante.
- dans un second temps, on a cherché à améliorer notre algorithme en conservant la mémoire et aussi par le biais de la parallélisation, au travers du JMCTSp et JMCTSP

De plus, cela nous a permit de renforcer dans un cas concret nos connaissances en C# afin de pouvoir avoir des bases solides utilisable en entreprises.