Awale

Emmanuel Vissuzaine, Pierre Tassel

Décembre 2017

1 Introduction

Il nous a été demandé comme projet en Résolution de Problèmes Complexes de créer une intelligence artificielle pour le jeu africain d'Awale (mais avec 10 cases au lieu de 6, afin d'augmenter la complexité du jeu). Ce rapport explique comment nous avons procédé pour répondre à la demande.

2 Représentation du jeu

```
#define LOG_MAX_CAILLOUX 8
#define LOG_MAX_CAILLOUX_MAIN 6
#define NB_CASES 10

struct Position {
        Cases_joueur cases_jeux[2 * NB_CASES];
        Main_joueur pris_joueur;
        Main_joueur pris_ordi;
        bool ordi_joue;
        bool ordi_joueur1;
};

struct Main_joueur {
        unsigned main_joueur : LOG_MAX_CAILLOUX_MAIN;
};

struct Cases_joueur {
        unsigned case_joueur : LOG_MAX_CAILLOUX;
};
```

On représente le jeu par un tableau dont chaque case correspond à un "trou" dans le plateau. Chacune d'elle fait 7 bits, car il y a 80 cailloux sur le plateau

(4 par cases) donc dans le pire des cas, il y aura 80 cailloux dans chaque case (impossible mais c'est la borne supérieure).

Sachant que la partie s'arrête quand l'un des deux joueur possède plus de 40 cailloux dans la main (la moitié des cailloux du jeu), 6 bits devraient normalement suffire, mais avec la règle de l'affamé on peut manger d'un seul coup tous les cailloux restants du côté de son adversaire, et donc dépasser cette limite. On borne donc le nombre de cailloux que l'on peut posséder au nombre total de cailloux présents au début du jeu.

Une position est donc représentée par l'état de chaque case du plateau et les cailloux détenus par chacun des joueur. Mais il nous faut aussi savoir si l'IA est à la place du joueur en première position ou en deuxième. En effet le Joueur 1 possède les cases 1 à 10, et le Joueur 2 les cases 11 à 20, savoir celles que l'on manipule avec l'IA est donc nécessaire.

Enfin, on a aussi un booléen qui représente le joueur courant, si *ordi_joue* est vrai alors c'est au tour de l'IA de jouer.

3 MinMax

```
int valeurMinMax(Position *courante, int profondeur, int profondeur_max
, int bound_a_b) {

        Position prochaine_position;
        /*On initialise la valeur courante d'alpha/beta
        a une valeur qui poussera l'algorithme a en changer
        des que possible (negative dans le cas d'une recherche de
        maximum, et inversement)*/
        int alp_bet_val = courante->ordi_joue ? -1000000 : 1000000;
        int tab_valeurs[NB_CASES];
        if (positionFinale(courante)) {
                if (courante->pris_ordi.main_joueur >
                        courante->pris_joueur.main_joueur) {
                        return (40000 * (profondeur_max - profondeur));
                }
                else {
                        if (courante->pris_ordi.main_joueur <
                                courante->pris_joueur.main_joueur) {
                                return -(40000 *
                                        (profondeur_max - profondeur));
                        }
                        return 0;
                }
        }
        if (profondeur == profondeur_max) {
```

```
        return evaluation(courante);
}
for (int i = 0; i < NB_CASES; i++) {
        /*si on calcule le coup de l'ordinateur (ie on prend le max)
        mais que la valeur d'alpha beta du pere
        (ie le coup joue par l'adversaire, donc le min des fils)
        est inferieure a la valeur courante
        d'alpha beta, alors l'adversaire ne devrait pas choisir ce
        fils donc on ne calcul pas le reste de l'arbre
        inversement si la valeur d'alpha beta est inferieure a la
        valeur alpha beta du pere (ie coup de l'ordi, le max des
        fils)
        on ne choisira pas cette branche donc on ne calcule pas le
        reste de l'arbre*/
        if ((courante->ordi_joue && (alp_bet_val > bound_a_b)) ||
                (!courante->ordi_joue && (alp_bet_val < bound_a_b))) {
                return alp_bet_val;
        }
        if (coupValide(courante, i + 1)) {
                jouerCoup(&prochaine_position, courante, i + 1);
                // pos_next devient la position courante
                tab_valeurs[i] = valeurMinMax(&prochaine_position,
                        profondeur + 1, profondeur_max, alp_bet_val);
                /*la valeur d'alpha beta devient le min/max de la
                valeur calculee et de la valeur alpha beta courante
                (min/max selon le joueur ordi ou adversaire)*/
                if ((courante->ordi_joue &&
                        (alp_bet_val < tab_valeurs[i])) ||
                        (!courante->ordi_joue &&
                        (alp_bet_val > tab_valeurs[i]))) {
                        //on met a jour la valeur alpha/beta
                        alp_bet_val = tab_valeurs[i];
                }
        }
        /*Sinon, si le coup n'est pas valide on lui donne
        une valeur extreme qui l'empechera d'etre selectionne*/
        else {
                if (courante->ordi_joue) {
                        tab_valeurs[i] = -1000000;
                }
                else {
                        tab_valeurs[i] = 1000000;
                }
        }
}
/*on renvoie la valeur de l'alpha beta
```

```
    qui est le min/max calcule dynamiquement*/
    return alp_bet_val;
```

L'algorithme de MinMax est donc un algorithme qui parcours l'arbre des configurations possibles après chaque coup. Une fois arrivé à la profondeur d'arbre voulue il applique une fonction d'évaluation sur la configuration pour lui affecter une valeur qui permettra de faire un choix optimal sur le coup à jouer.

La valeur d'un nœud correspond au maximum ou au minimum des valeurs des nœuds suivants. La différence du choix est dûe au fait que l'adversaire va chercher à choisir la pire configuration pour l'IA, contrairement à celle-ci qui va chercher à choisir la meilleure.

Si l'algorithme tombe sur une configuration où le jeu se termine, alors on lui donne une valeur positive si l'IA gagne, négative si c'est l'adversaire qui l'emporte, et 0 sinon. On applique ensuite un coefficient en fonction de la profondeur qui fera privilégier à l'algorithme les fins de jeu les plus rapides. La profondeur est modifié dynamiquement, elle est au minimium de 8, mais si le coup précedent à été très rapide (moins de 0,25 secondes) alors on l'augmente d'un cran. Au contraire, si le coup précédent à pris plus de 1,5 secondes on réduit la profondeur jusqu'a minimum 8. De même, si le coup précedent a pris plus de 2 seconde, alors on remet la profondeur a 8 directement. Cela permet d'augmenter un peu la profondeur en fonction du plateau, au début il y a peut de coupe alpha/béta et beaucoup de coup possibles donc on ne peut pas augmenter grandement la profondeur, mais généralement en milieu/fin de partie la profondeur peut être augmenter de 1 ou 2 en fonction des cas.

L'introduction d'une valeur d'alpha/beta permet d'appliquer à l'arbre des coupes sur les branches (l'algorithme ne calcule donc pas toutes les configurations possibles). En effet la valeur d'alpha/beta courante d'un nœud est le maximum (ou le minimum) de toutes les valeurs des nœuds fils. Ces nœuds fils sont inversement les minima (maxima) des nœuds de la génération suivante. Ainsi, comme l'algorithme manipule des valeurs extrêmes, on peut prédire si c'est utile de continuer la recherche ou non, auquel cas il remonte directement la valeur courante trouvée.

NB : La valeur alpha/beta de la génération précédente est passée sous forme de paramètre lors de l'appel de la fonction ("bound_a_b").

Lors du premier appel de l'algorithme MinMax, cette valeur est initialisée à une valeur extrême qui pousse l'algorithme a calculer tous les nœuds de la première génération de fils.

4 Calcul du prochain coup

Lorsque notre IA doit jouer, on lance l'algorithme MinMax sur tous les coups possibles, et on sélectionne ensuite la configuration avec la plus haute valeur de retour.

5 Fonction d'évaluation

Afin de créer notre IA, nous avons cherché à optimiser au mieux notre fonction d'évaluation. En effet c'est grâce à celle-ci, appelée par l'algorithme Min-Max, qu'il est possible de gagner énormément en efficacité, et faire ainsi en sorte de gagner le plus souvent possible.

Nous sommes initialement partis d'une fonction d'évaluation basique, qui évaluait une configuration uniquement selon le nombre de cailloux pris par notre IA et son adversaire. Même si cette fonction donnait des résultats corrects, le jeu est suffisamment complexe pour qu'existent de nombreuses stratégies à mettre en place afin d'obtenir une victoire.

Nous nous sommes donc renseignés plus en profondeur sur des études scientifiques concernant ce jeu. En effet il existe pléthore de documents à propos de l'Awale, et il nous a suffit de choisir ceux qui correspondaient le plus à ce que nous cherchions(cf Ressources).

Ainsi nous avons pu trouver une fonction d'évaluation bien plus complète que celle que nous avions, et qui prenait en compte de nombreux paramètres tirés de stratégies de victoire (ex :*nombre de cases atteignables par soi/l'adversaire, nombre de trous avec 2 ou 3 cailloux que l'adversaire peut créer et ramasser, etc...*).

Voici les paramètres que nous avons implémentés :

- Nb de cases avec 2 cailloux
- Nb de cases avec 1 caillou
- Nb de cases vides
- Nb de cases accessibles par l'IA
- Nb de cases accessibles par l'adversaire
- Nb de cases avec suffisamment de cailloux pour faire un tour complet du plateau
- Nb de cailloux pris par l'IA
- Nb de cailloux pris par l'adversaire

Chacun de ces paramètres est ensuite pondéré par un coefficient qui correspond à son influence sur la situation. Ces coefficients été fournis à la base pour un Awale à 6 cases [1], nous avons rendu négatif certains coefficient pour coller à certaines règles modifiées en rendant négatif les coefficients défavorables (comme le fait d'avoir beaucoup de puit vide, qui est défavorable dans notre cas). Nous avons essayé de calculer les meuilleurs coefficient pour l'awale a 10 cases (voir fichier TestGenerationCoefficient.cpp), mais malheuresement, cella prennait beaucoup trop de temps

Références

[1] Mohammad Daoud, Nawwaf N. Kharma, A. Haidar, J. Popoola *Ayo, the Awari player, or how better representation trumps deeper search*

https://www.researchgate.net/publication/4089851_Ayo_the_
Awari_player_or_how_better_representation_trumps_deeper_search