Projet M2 GIL - IA

22 Janvier 2017

Axelle	BOUCHER	Florian	INCHINGOLO
Yohann	HENRY	Jérémie	PANTIN
		Simon	MARTIN

Table des matières

1 Introduction

Le but du projet ci-présent est l'implantation de plusieurs intelligences artificielles qui ont pour but de résoudre le jeu des dames internationales. En effet, les élèves doivent se basés sur les algorithmes vus en cours afin de répondre à cette problématique : MinMax, NegMax, Elagage Alpha-Beta, SSS*, etc ... Ils doivent donc retenir trois solutions afin de concorder avec les enjeux de la plateformes dont il est nécessaire de présenter trois niveaux de difficulté.

2 Algorithmes

Nous avons choisi trois algorithmes du cours :
— NegaMax
— NegaMax avec Elagage Alpha-Beta
— SSS*
Nous allons ainsi voir dans la suite ce que représente les forces de chacun de ceux-ci et pourquoi leur choix.

2.1 NegaMax

NegaMax est une amélioration/simplification de l'algorithme MinMax consistant à minimiser la perte maximum (le pire des cas). Il s'applique sur les jeux à deux joueurs à somme nulle. L'algorithme est récursif et s'attèle à choisir, dans le cadre d'un n joueur, le prochain mouvement selon une valeur déterminé par un parcours. Cette valeur est calculée au moyen d'une fonction d'évaluation de position et indique dans quelle mesure le coup à jouer est optimal. Nous verrons un petit peu plus tard la fonction d'évaluation développée et choisie.
Pourquoi NegaMax et non MinMax ? NegaMax reproduit exactement le même schéma que MinMax, simplement que les comparaisons entre les différentes valeurs de chaque étage sont facilitées par l'emploi d'une vérification unique : `MAX`.

2.2 NegaMax - Élagage Alpha-Beta

L'élagage Alpha-Beta est un algorithme de recherche qui vise à diminuer le nombre de noeuds à évaluer par l'algorothme MinMax. En effet il consiste à complètement arrêter l'évaluation d'un mouvement quand au moins une possibilité trouvée prouve que le passage courant fait partie des pires cas examinés. L'avantage de cet algorithme réside dans le fait que les branches de l'arbre de recherche peuvent être éliminées. Ainsi, le temps de recherche peut être limité à la seule arborescence la plus prometteuse. Cette optimisation permet donc la réduction de la profondeur effective à presque la moitié d'un simple MinMax (si les noeuds sont biens disposés).
Nous basons cet algorithme sur l'arbre de recherche NegaMax pour les raisons citées plus haut. En effet, comme pour le MinMax, nous avons directement appliqué l'amélioration NegaMax.

2.3 SSS*

Cet algorithme est un algorithme de recherche voulant, comme pour MinMax, minimiser la perte maximum, et donc le pire des cas. Il effectue une recherche en traversant un arbre de jeu, tel que MinMax.
La force de SSS* est qu'il permet d'élaguer des branches que l'algorithme à élagage Alpha-Beta n'élague pas, et vice-versa. Nous l'avons ainsi appliqué sur l'élagage Alpha-Beta afin de répondre au mieux à cette utilisation.

3 Choix d'implémentation

Lors de notre implémentation nous avons voulu optimiser au mieux notre code et également les opportunités que nous offraient le langage C++. Ainsi nous avons décidé de construire l'arbre de recherche tout en sauvegardant les données nous intéressant. Nous pouvons donc voir que la recherche du meilleur coup se fait au même moment que la construction. Nous gagnons ainsi du temps avec le parcours final.
La machine de test sur laquelle a été testé les algorithmes est une machine ayant 6 ans, notre volonté dans ce choix est de démontrer que l'optimisation logicielle et algorithmique est une partie importante du projet.

- Profondeur NegaMax : 6
- Profondeur SSS* : 8
- Profondeur NegaMax Alpha-Beta : 10

Sur des machines possédant de plus fortes configurations, nous pouvons voir à la hausse les profondeurs. Notamment pour SSS*, qui demande plus de ressources mémoires que son concurrent Élagage Alpha-Beta.

4 NegaMax

```
/**
 * Represente un noeud de l'arbre NegaMax.
 */
class NegMax {
private:
    /**
     * Represente le tour de ce noeud.
     */
    Turn *m_turn;
    /**
     * Represente la liste des enfants de ce noeud.
     */
    std::list<NegMax *> *m_children;
    /**
     * Represente le parent du noeud.
     */
    NegMax *m_parent;
    /**
     * La sequence du dernier mouvement.
     */
    MoveStack *m_lastMove;
    /**
     * La sequence representant le meilleur mouvement.
     */
    MoveStack *m_bestMove;
    /**
     * L'heuristic du noeud.
     */
    int m_heuristic;
    /**
     * La couleur du noeud.
     */
    bool m_isWhite;
```

Comme indiqué plus haut, nous cherchons à faire la recherche durant la construction. Pour

ce faire nous décidons de prendre plus d'espace mémoire que d'espace en temps pour faire ceci. Dans la structure plus haute nous voyons qu'un noeud sauvegarde le tour sur lequel on joue mais également le dernier mouvement joué et le meilleur trouvé.

Cela se justifie par l'emploie de la fonction :

```
int NegMax::negMax(int depth, bool isWhite) {
    if (depth == 0 ||
        (m_turn->getBoard().getString().find("b") == string::npos &&
         m_turn->getBoard().getString().find("B") == string::npos) ||
        (m_turn->getBoard().getString().find("w") == string::npos &&
         m_turn->getBoard().getString().find("W") == string::npos)) {
        int heur;
        (isWhite == m_isWhite) ? heur = m_lastMove->getHeuristic(*(m_parent->m_turn)) :
                heur = -m_lastMove->getHeuristic(*(m_parent->m_turn));
        return heur;
    }
    generateChildren();
    if (m_children->size() == 0) {
        if (isWhite == m_isWhite) {
            return -4483;
        } else {
            return 4483;
        }
    }
    int temp = -4483;
    for (auto it = m_children->begin(); it != m_children->end(); it++) {
        int val = - (*it)->negMax(depth - 1, !isWhite);
        if (val > temp) {
            temp = val;
            m_bestMove = (*it)->m_lastMove;
        }
    }

    return temp;
}
```

-4483 étant la représentation de l'infini, nous initialisons toutes les nodes feuilles à celles-ci afin qu'ils prennent la valeur du max. Un appel récursif est fait lorsque nous appelons la valeur la plus basse de chaque enfants. Nous gardons ainsi le meilleur mouvement à chaque fois.

5 NegaMax - Alpha Beta

```
class NegMaxNode {
private:
    /**
     * Tour de jeu représenté par ce noeud
     */
    Turn *m_turn;
    /**
     * Liste des enfants de ce noeud
     */
    std::list<NegMaxNode *> *m_children;
    /**
     * Parent de ce noeud
     */
    NegMaxNode *m_parent;
    /**
     * Coup effectué pour arriver à ce tour
     */
    MoveStack *m_lastMove;
    /**
     * Permet la transmission du meilleur coup
     */
    MoveStack *m_bestMove;
    /**
     * La couleur du joueur qui applique l'algorithme
     */
    bool m_isWhite;
```

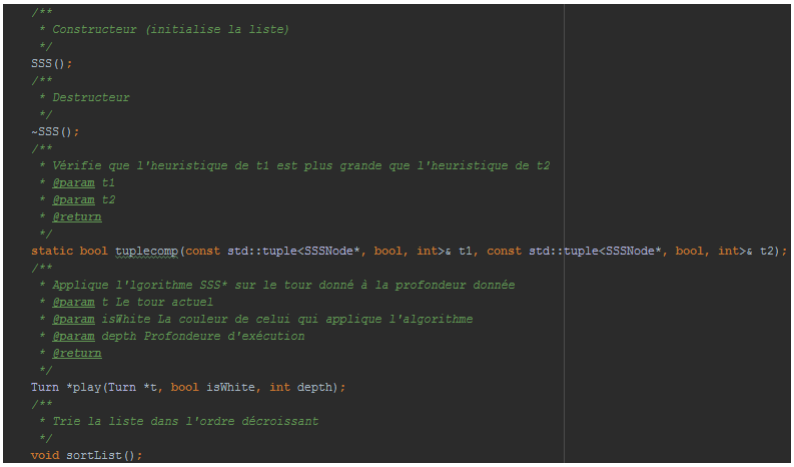
Le code de l'élagage Alpha Beta reprend exactement les mêmes attributs que l'algorithme NegaMax. Nous allons seulement effectuer un changement dans l'appel récursif afin de prendre en compte Alpha et Beta :

```
int NegMaxNode::negMax(int depth, int alpha, int beta, bool isWhite) {
    if (depth == 0 ||
            (m_turn->getBoard().getString().find("b") == string::npos &&
                    m_turn->getBoard().getString().find("B") == string::npos) ||
            (m_turn->getBoard().getString().find("w") == string::npos &&
                    m_turn->getBoard().getString().find("W") == string::npos)) {
        int heur;
        (isWhite == m_isWhite) ? heur = m_lastMove->getHeuristic(*(m_parent->m_turn)) : heur = -m_lastMove->getHeuristic(*(m_parent->m_turn));
        return heur;
    }
    generateChildren();
    if (m_children->size() == 0) {
        if (isWhite == m_isWhite) {
            return -4483;
        } else {
            return 4483;
        }
    }
    for (auto it = m_children->begin(); it != m_children->end(); it++) {
        int val = - (*it)->negMax(depth - 1, -beta, -alpha, !isWhite);
        if (val > alpha) {
            m_bestMove = (*it)->m_lastMove;
            alpha = val;
        }
        if (alpha >= beta) {
            return alpha;
        }
    }
    return alpha;
}
```

L'élagage est fait au fur et à mesure de la construction et la provocation d'un alpha inférieur arrête la construction courante.

6 SSS*

```cpp
/**
 * Classe effectuant l'algorithme SSS*.
 */
class SSS {
private :
    /**
     * Liste contenant les triplets rangés dans l'ordre décroissant de sheuristiques.
     * Les triplets sont composés de : - Le noeud associé
     *                                 - Le statut vivant (false) ou résolu (true)
     *                                 - l'heuristique
     */
    std::list<std::tuple<SSSNode*, bool, int>> *l;
    /**
     * La racine de l'arbre utilisé par l'algorithme
     */
    SSSNode *root;
```



```cpp
/**
 * Constructeur (initialise la liste)
 */
SSS();
/**
 * Destructeur
 */
~SSS();
/**
 * Vérifie que l'heuristique de t1 est plus grande que l'heuristique de t2
 * @param t1
 * @param t2
 * @return
 */
static bool tuplecomp(const std::tuple<SSSNode*, bool, int>& t1, const std::tuple<SSSNode*, bool, int>& t2);
/**
 * Applique l'lgorithme SSS* sur le tour donné à la profondeur donnée
 * @param t Le tour actuel
 * @param isWhite La couleur de celui qui applique l'algorithme
 * @param depth Profondeure d'exécution
 * @return
 */
Turn *play(Turn *t, bool isWhite, int depth);
/**
 * Trie la liste dans l'ordre décroissant
 */
void sortList();
```

7 Heuristique

Pour l'heuristique, on va calculer la valeur de chaque pion (alliée comme ennemi) puis les additionner (les ennemis donnant évidemment des points négatifs). Pour chaque pion : - 1 point + 1 point pour chaque ligne après la première ligne (pour marquer l'avancement d'un pion) - *2 si le pion est sur la couronne intérieur (1 case avant le bord) et *4 si le pion est sur la couronne extérieure (les case sur le bord) (ces cases étant difficile à capturer, on va grandement les privilégiés) Une dame étant quelque chose de décisif dans le jeu de dame, sa valeur de base correspond à un pion sur le bord en face (là où il se transforme en dame) ce qui nous donne (1 + 6) * 4 = 28. Les dames étant trop décisif, on voudra éviter que l'adversaire n'ait de dames, ainsi une dame adverse vaudra 2 dames alliées soit 56. La plus grosse valeur d'une dame alliée est donc 28 *4 = 112 et la plus grande valeur d'une dame ennemi est 56 * 4 = 224. On cherchera à ce qu'une victoire soit plus grande que le meilleur des plateaux, idem pour la défaite, plus petit que le pire des plateau. On préfèrera un nul à une défaite mais on préfèrera un plateau quelconque à un nul. Ainsi on fait le classement suivant :

— Victoire
— Plateau quelconque
— Nul
— Défaite

La plus grande valeur absolue d'un plateau théorique serait de 20 dames sur la couronne extérieure soit 20 * 224 = 4480. Donc un nul vaut -4481, une victoire 4482 et une défaite -4482. On considèrera l'infini à partir de 4483, et moins l'infini à partir de -4483.

8 La stratégie gagnante

Vous vous en doutez, nous n'avons pas réussi à résoudre les dames internationales. Du moins nous avons réussi à sortir des stratégies permettant de favoriser une victoire. En effet, lorsqu'il y a une dame, dans le jeu des dames internationales, les chances de victoires sont augmentées de plus de 75% pour celui en possédant une le premier. Il est également à noter que nous n'avons pas réussi à déterminer si l'un ou l'autre des joueurs possédait un coup gagnant dès le début. En effet si nous nous basons sur le Théorème de Von Newman, il existe un coup gagnant et si il y a possibilité d'égalité, égalité.

Ce projet a été une réelle épreuve intellectuelle, pouvant allier théorie et expérimentation, il a fallu réussir à faire progresser nos intelligences grandement dans le but de trouver une heuristique universelle et répondant convenablement au besoin.

9 Pour aller plus loin (MonteCarlo)

Comme vous pourrez le voir dans les sources, il existe une approche non développée pendant le cours qui est MonteCarlo. En effet nous avons réussi à faire une implémentation mais celle-ci ne répondait pas convenablement à notre besoin (manque de temps et d'expérience ?). Vous pouvez toutefois voir un exemple d'application sur un arbre NegMax :

```
/**
 * Fonction permettant de generer les enfants.
 */
void generateChildren();
/**
 * Premiere phase de MonteCarlo, il faut selectionner quel noeud nous prenons
 *  pour propager l'algorithme.
 */
Turn &selectionCarlo();
/**
 * Une fois le noeud choisi dans la premiere phase, nous etendons celui-ci
 *  pour faire de nombreux tirages aleatoires permettant de nous assurer
 *  qu'il existe une strategie gagnante sur ce noeud.
 */
void expandCarlo();
/**
 * Une fois etendu, nous provoquons une simualtion sur tous les noeuds de l'arbre
 * , y compris  ceux precendents la selection.
 */
void simulateCarlo(int depth);
/**
 * Nous remontons le resultat des simulations jusque la racine.
 */
Turn &backPropCarlo();
```