Projet de programmation d'un bot jouant au méta-morpion

Cocheton Alexis
Collenot Heiarii
Poitelea Mihai
Ruiz Nicolas
Serva Benjamin
Tisserant Tom

Encadrants :

Marie-Laure Mugnier
Guillaume Pérution-Kihli
Florent Tornil

23/05/2022

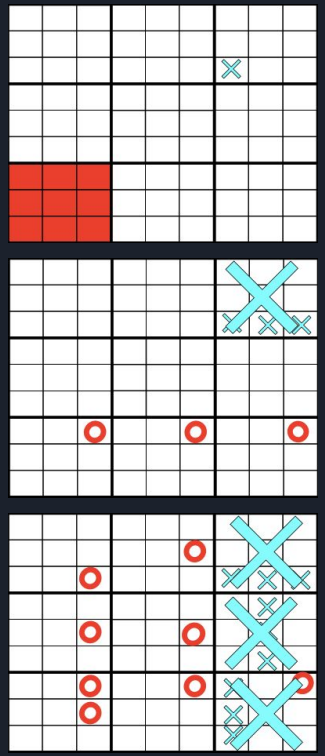
Sommaire

Index

1. Le méta-morpion
2. Cahier des charges
3. CODINGAME
4. Uml
5. Minimax
    a. *Qu'est-ce que minimax ?*
    b. *Élagage alpha-bêta*
    c. *L'évaluation statique*
    d. *Pseudo-code*
6. MCTS
    a. *Qu'est-ce que le Monte Carlo Tree Search ?*
    b. *Adaptation dans Codingame pour le méta-morpion*
    c. *Composition et déroulement de notre algorithme*
    d. *Difficultés et optimisation*
7. Débogage
8. Conclusion

Le méta-morpion

Règles



- Au premier coup le joueur joue où il veut
- Les coups “envoient“ l'adversaire dans la sous-grilles correspondante
- Lorsque un morpion est gagné il est considéré comme appartenant au vainqueur
- Une fois le morpion gagné, on ne peut plus jouer dedans, si un joueur est envoyé dans ce dernier il pourra choisir de jouer où il veut parmi les cases libres
- Pour gagner la partie il faut gagner trois sous-grilles alignées horizontalement , verticalement ou en diagonal

Cahier des charges

Objectifs

Code :

- Coder une IA capable de jouer au méta-morpion sur le site codingame
- L'IA créée doit être un minimax ou un Monte-Carlo.
- Utilisation du C++ pour l'implémentation de cette IA.
- Implémentation en utilisant des classes
- Emploi de la généricité
- Emploi de structures dynamiques vector / pointeurs intelligents (smart pointers)

Gestion :

- *Répartir le travail équitablement*

Objectif final :

- *Arriver à battre le bot de l'encadrant*

Codingame

Démonstration

Démonstration vidéo

UML

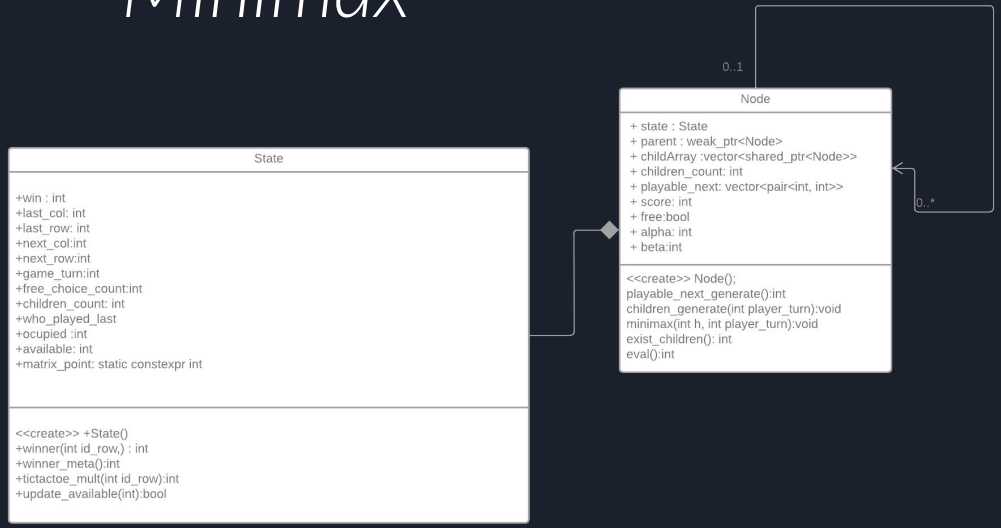
Modélisation

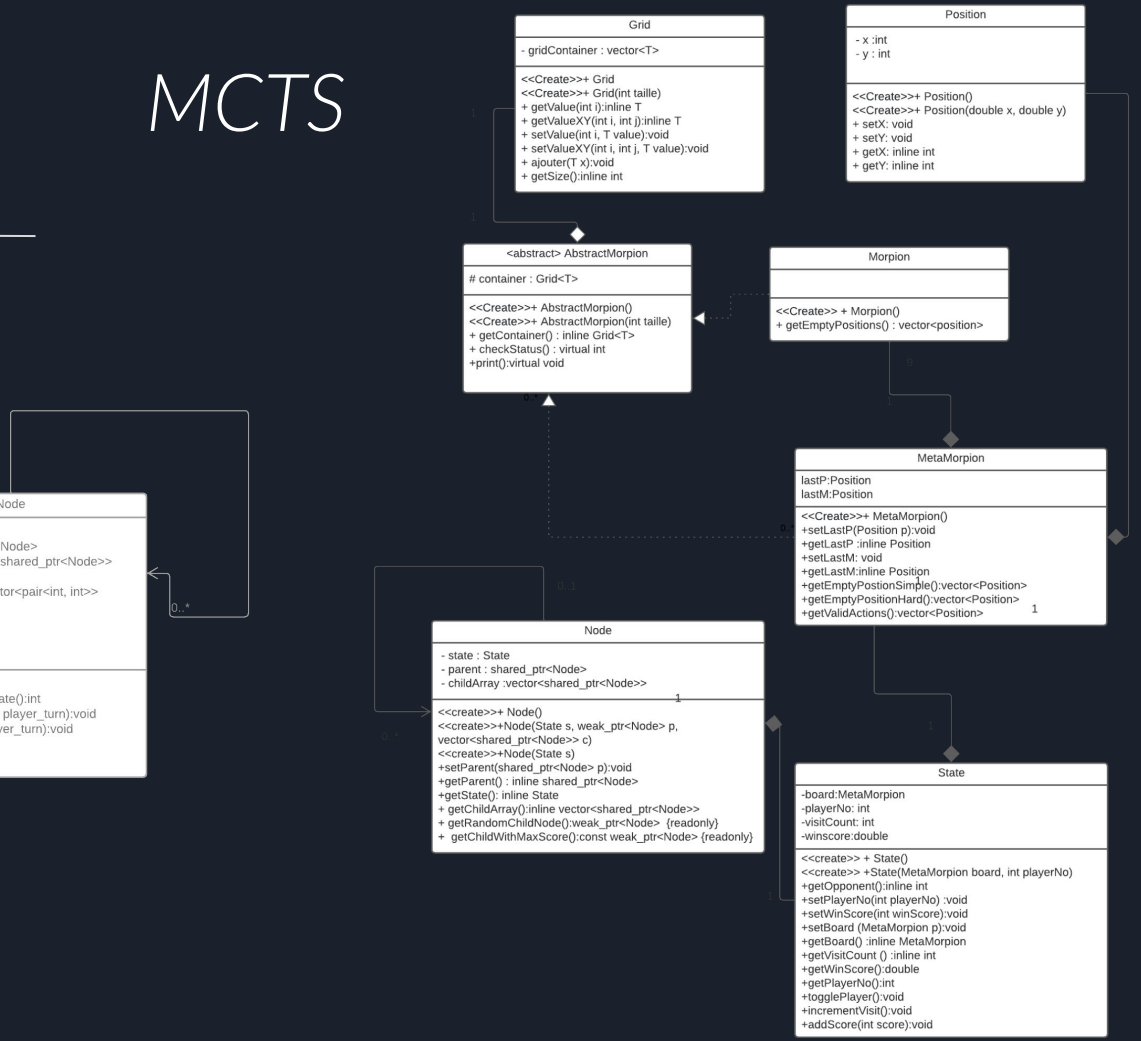
UML

Modélisation

MCTS

Minimax



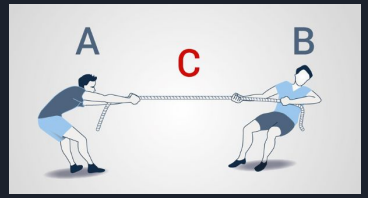


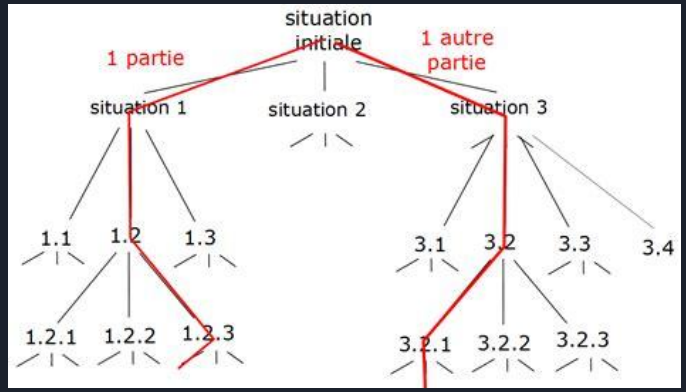
Minimax

Qu'est-ce que minimax ?

- Théorie des jeux : mathématiques des stratégies entre agents (jeux/économie)

- Jeux à somme nulle
  - lose = -1
  - win = *nbLoser/nbWinner*





- Meilleur coup : gain maximum ? perte minimum ?
  - Perte minimum : le plus grand des petits scores : max(min(...))

L = Totale

H = Partielle/Totale

- Recherche dans l'arbre :
  - Largeur (possibilités)
  - Hauteur/horizon (coups)
  - Optimisation : élagage alpha-bêta
  - Évaluation statique

Minimax

L'évaluation statique

victoire = maximum de points

les 10 premiers coups

matrice de points :

30	20	30	24	16	24	30	20	30
20	25	20	16	20	16	20	25	20
30	20	30	24	16	24	30	20	30
24	16	24	36	24	36	24	16	24
16	20	16	24	30	24	16	20	16
24	16	24	36	24	36	24	16	24
30	20	30	24	16	24	30	20	30
20	25	20	16	20	16	20	25	20
30	20	30	24	16	24	30	20	30

après les 10 premiers coups

points par morpions gagnés :

1000	800	1000
800	1200	800
1000	800	1000

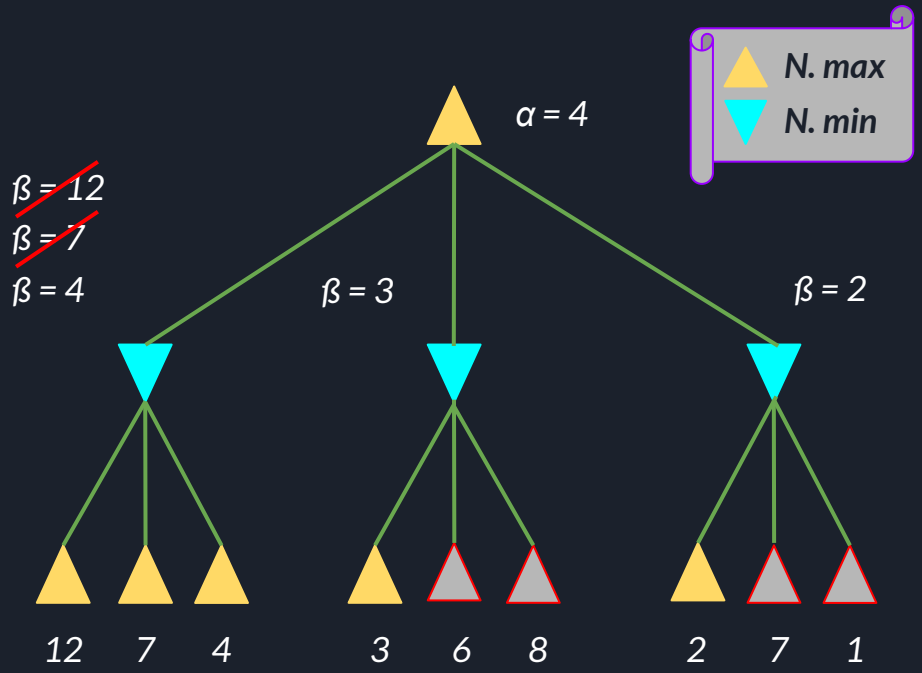
on ajoute des points si deux pions ou morpions alignés :

-10 à chaque fois que l'adversaire a le choix

Minimax

Élagage alpha-bêta

- *Ne modifie pas le résultat*
- *Économiser des ressources en ne faisant qu'un parcours partiel de l'arbre*
- *Dépend de la qualité des heuristiques*
- *Plus efficace si situations ordonnées (tri)*
- *Risque de perdre l'horizon d'avance d'une implémentation arborescente*
  Prévoir le reset de l'arbre



Alpha : plus grand minimum

Bêta : plus petit maximum

Si Alpha >= Bêta alors élagage

Minimax

Pseudo-code

```
void Node::minimax(int horizon, int joueur)
{ // valeur retournée par effet de bord car void
    si (simulation_inexistante) alors {
        creer noeud_simulation
        creer_simulations_suivantes
        trier_simulation_suivantes
    }

    // cas d'arret
    si (horizon <=0 || pas_de_simulation) evaluation_statique
    sinon {
        score=infini*joueur
        boucle des simulations suivantes {
            charger_prochain_coup_à_simuler
            transmettre_alpha_&_beta
            minimax
            si (noeud_min) alors {
                beta = min (beta, valeur_minimax)
                score = min (score, valeur_minimax)
            } sinon { //noeud max

                alpha=max(alpha, valeur_minimax);
                score=max(score,valeur_minimax);
            }
            si (alpha>=beta)  break; //élagage
        }
    }
}
```

```
int main()
{
    Initialisation_objets_&_variables // 1 seule fois, début de partie
    tant que vrai {// tours de jeu
        synchronisation_&_traduction_variables // formatage des données

        si adversaire_a_joué alors { // jouer le coup adverse sur arbre
            recuperer_adresse_simulation
            si (simulation_inexistante) alors reset_arbre
            sinon descendre_sur_arbre
        }

        si premier_tour alors coup_determiné // pour prendre de l'avance

        si pas_de_simulations_connues alors creer_simulations

        trier_simulation
        definir_horizon
        score = -infini
        alpha = -infini

        boucle des simulations {
            beta = +infini
            charger_prochain_coup_à_simuler
            minimax
            si score<valeur_minimax alors {
                score = valeur_minimax
                stocker_meilleure_simulation
            }
        }

        descendre_sur_arbre
        traduction
        cout<<move<<endl; //on joue le coup
    } // fin de tour
}
```

MCTS

Qu'est-ce que le Monte Carlo Tree Search ?

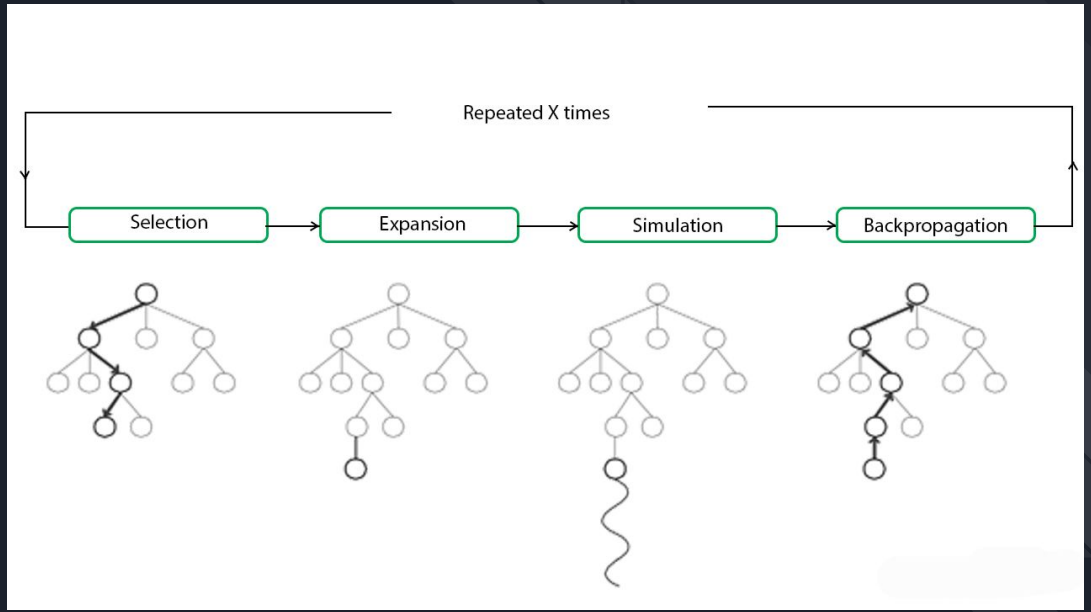
4 étapes

- Sélection grâce à la formule UCT

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

- Expansion
- Simulation
- Backpropagation

Arbre du MCTS pour nos 4 étapes



MCTS

Adaptation dans Codingame pour le méta-morpion

- Implémentation du MCTS en morpion
- Fonctions du MCTS
- Classes pour le morpion et l'arbre

morpion ⟶ méta-morpion:

- Adaptation des structures de données
- Ajout de nouvelles classes et méthodes
- Ajout des nouvelles règles

MCTS

Composition et déroulement de notre algorithme

- Fonction main

```
int main()
```

- Fonctions principales

```
void MonteCarloTreeSearch(shared_ptr<Node> rootNode, int playerNo)
inline void expandNode(shared_ptr<Node> node)
int simulateRandomPlayout(shared_ptr<Node> node)
void backPropagation(shared_ptr<Node> nodeToExplore, int playerNo, int playoutResult)
```

etc, ...

- Classes et leurs méthodes

```
class Node
template <class T>
class AbstractMorpion
class MetaMorpion : public AbstractMorpion<Morpion>
class Morpion : public AbstractMorpion<int>
```

etc, ...

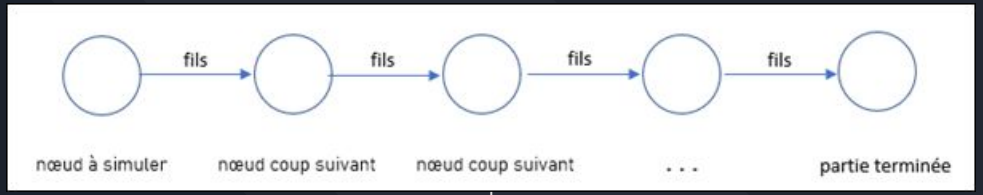
Partie Algorithme Spécifique MCTS

Partie Structures de Données

MCTS

Difficultés et optimisation

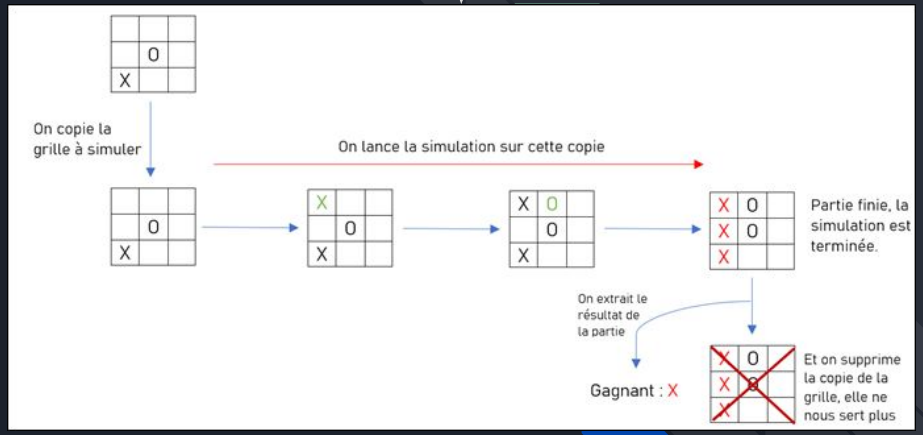
- Réappropriation du C++
- Apprentissage de nouvelles notions comme les pointeurs intelligents et application à notre algorithme
- Changement d'approches pour divers problèmes



Trop gourmand en temps ✖

Approche phase de simulation

Bien plus rapide ⭕



Optimisation du code : possibilité d'exécuter toujours plus de MCTS et donc d'avoir plus de simulations sur nos noeuds

> Nbr MCTS fait : 137 ⟶ > Nbr MCTS fait : 745 ⟶ > Nbr MCTS fait : 13569

Débogage

Résolution de problèmes

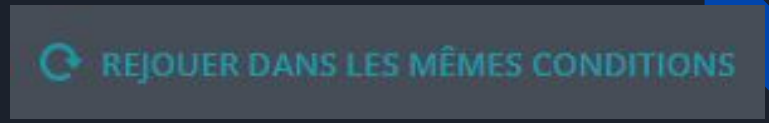
1) Utilisation du Cerr :

2) Terminal Linux

3) Pas à pas (test d'intégration)

Sortie d'erreur :
> coup jouable : 72 74 76 71 75

Sortie d'erreur :
> Nbr MCTS fait : 13569



Conclusion

Bilan

- *Les difficultés rencontrées*
  - Se réapproprier le c++
  - Devoir adapter l'uml
  - S'adapter à Codingame et sa structure

- *Perspective d'amélioration*
  - Optimisation du code
  - Combinaison des deux :
    - Utiliser minimax dans la partie simulation
    - Commencer le jeu avec Minimax puis terminer avec MCTS
  - Multi-thread