Première version du cahier d'analyse des besoins à propos d'un programme de jeux d'échecs

Rossignon Morgan, Daniel Karl, Salomode Florian, Beites Marvin, Zucchelli Thomas

15 mars 2022

Table des matières

1 Objectifs généraux du projet

Le but du projet est de programmer un joueur d'échec complet en appliquant les algorithmes les plus fréquents et en utilisant au mieux les bitboards qui sont un moyen de représenter l'échiquier à l'aide de douze variables de 64bits chacune (comme le nombre de case d'un échiquier).
Ces douze variables, une pour chaque type de pièces par couleur stockent donc les positions des différentes pièces du plateau de jeu. Le site suivant l'explique d'ailleurs de manière précise [**?**].

L'interface utilisateur sera purement en mode texte (ce n'est pas le but du projet). Si possible, plusieurs approches pourront être implantées et comparées via des tournois entre les différentes stratégies.

Le projet se dirigera vers des techniques d'exploration d'arbre classiques telles que Minimax-ab [**?**], Negamax [**?**], Negascout [**?**], MTD(f) [**?**], Monte-Carlo Tree Search (MCTS)[**?**]. Ainsi qu'au développement d'heuristiques propres au jeu dans l'idée d'éliminer le plus efficacement les branches "perdantes" [**?**].

2 Analyse de l'existant

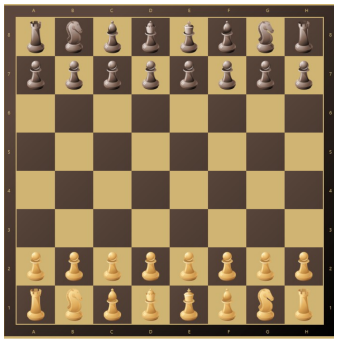
Il est possible de trouver le pseudo-code et le principe de fonctionnement des algorithmes Minimax- ab [**?**], Negamax [**?**], Negascout [**?**], MTD(f) [**?**], Monte-Carlo Tree Search (MCTS)[**?**] sur internet, ces pseudo-codes sont disponibles en annexe.
Afin de discuter du fonctionnement de l'algorithme Minimax-ab il faut parler du fonctionnement de l'algorithme Minimax.
L'algorithme Minimax a pour but de trouver la liste des meilleurs coups à jouer, pour cela on va générer tous les coups possibles dans un arbre jusqu'à une certaine profondeur, puis en partant des feuilles en remontant jusqu'à la racine on remonte alternativement le meilleur coup favorisant notre victoire et le meilleur coup favorisant le coup de notre adversaire[**?**].

L'algorithme Minimax-ab a le même principe à la différence que lorsqu'il remonte l'arbre des coups possibles il ne prend pas en la peine de juger l'efficacité de certains coups en fonction des autres coups qu'il a déjà jugé plus efficace, au travers d'une comparaison entre deux variables, alpha et bêta[**?**].

L'algorithme Negamax se différencie de l'algorithme Minimax-ab par le fait que plutôt que de maximiser le coup du joueur et minimiser le coup de l'adversaire il inverse le signe des évaluations à chaque niveau de profondeur de l'arbre de décision et ne cherche plus qu'à maximiser la valeur du coup à évaluer[**?**].

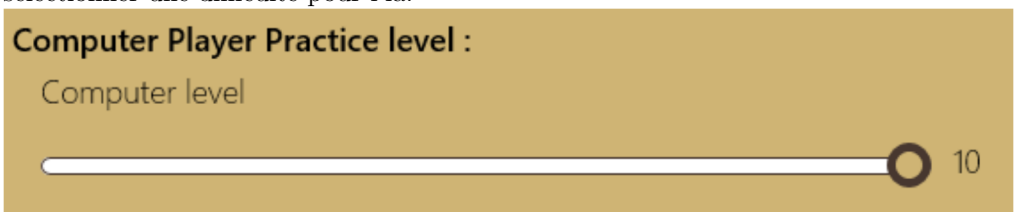
Avec l'algorithme Monte-Carlo Tree Search, un nœud de l'arbre de décision correspond à un état du jeu mais possède aussi deux valeurs, le nombre de simulations gagnantes et le nombre de simulations totales sur la branche. L'algorithme fonctionne en quatre étapes[**?**] :

- La sélection successive des enfants de la racine jusqu'à atteindre une feuille.
- L'expansion de l'arbre, en ajoutant un enfant à la feuille si celle-ci n'est pas finale.
- La simulation d'une partie au hasard depuis l'enfant rajouté jusqu'à atteindre une fin de partie.
- La remontée du résultat de la partie par la mise à jour du nombre de simulation victorieuse et du nombre de simulation totale pour chaque nœuds.

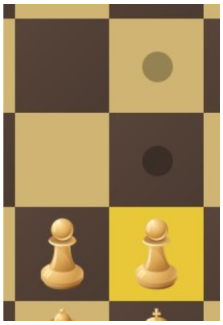


La plupart des logiciels de jeu d'échecs proposés au grand public, disponibles dans les boutiques virtuelles comme le Microsoft store, optent pour un affichage graphique afin de jouer aux échecs, toutefois on peut se rendre compte que ces logiciels permettent aux joueurs de :

— sélectionner qui joue en premier. (qui joue les pièces blanches)
— sélectionner le nombre de joueurs. (l'ia contre l'ia, 1 joueur contre l'ia ou 2 joueurs)
— sélectionner une difficulté pour l'ia.



— visualiser les mouvements d'une pièce



— sauvegarder une partie pour la reprendre plus tard. -charger une partie sauvegardée.

Mais il existe également des logiciels, quant-à eux plus scientifique, avec pour vocation d'exercer des algorithmes contre des champions humains ou contre d'autres logiciels.
C'est le cas de Kaissa qui fût le premier programme à avoir gagné le championnat ACM d'échecs informatiques de 1974. Ce dernier utilise un algorithme Minimax alpha beta [**?**] pour calculer ses coups et un système de bitboard pour déterminer les mouvements légaux d'une pièce [**?**].
Nous pouvons cité Deep Thought qui remporta le 19ème championnat ACM d'échecs informatiques, il s'agit du premier ordinateur avec le niveau d'un grand maître et joua contre Garry Kasparov en 1989 et fut battu[**?**].
Jusqu'à Deep Blue qui possède une puce lui permettant de paralléliser l'algorithme de recherche alpha-bêta[**?**] et fut le premier ordinateur à avoir battu Kasparov en 1996[**?**].

3 Description des besoins

3.1 Liste des besoins fonctionnels

— **Création du plateau de jeu :**

— Représentation du plateau à l'aide de bitboards [**?**], chaque bitboard représentera l'emplacement des pièces d'un même type et d'une même couleur. (c-a-d qu'il y aura un bitboard pour l'emplacement de tous les pions blanc, un bitboard pour l'emplacement de tous les Cavaliers blanc,...).

— **Afficher le plateau de jeu :**

— Affichage en texte, une lettre représente un pion (p = pion, k = roi, q = reine, b = fou, r = tour, n = cavalier). Les cases vides, quand à elle contiendront un point "." dans un souci de lisibilité.

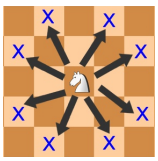
— Afin de différencier les pions des 2 joueurs du plateau de jeu, le joueur blanc aura des pions en majuscule et le joueur noir en minuscule.

— Les coordonnées des cases seront les mêmes que sur le jeu d'échec original, les numéros représentent les lignes et les lettres pour les colonnes, par exemple (a1 = 1ère case en bas à gauche ...).

— **Lister les mouvements possibles :**
En cas de jeu de l'utilisateur contre un des différents algorithmes, une commande "playable" permet d'afficher la liste de tous les coups jouables pour l'utilisateur (par exemple : h2h4 ...).

— **Déplacer une pièce :**
Le programme doit faire respecter les règles du jeu aux différents algorithmes, ou joueur humain. Le déplacement d'une pièce doit vérifier au préalable que le mouvement est possible, et si c'est le cas, ce que ce mouvement engendre.

1. Le programme doit vérifier que le mouvement demandé correspond à un des mouvements possibles de la pièce. Suivant les pièces :
   — **Pion (p,P) :** Le pions avance d'une case en ligne droite uniquement (ni sur les cotés ni en arrière) hormis le premier déplacement de chaque pions qui peut être de 1 ou 2 cases au choix. Hormis quand le pion "mange" une pièce, dans ce cas la le déplacement se fait en diagonale d'une portée de 1.
   — **Roi (k,K) :** Le roi peut aller dans toutes les directions mais d'une seule case.
   — **Reine (q,Q) :** La reine peut aller dans toutes les directions et n'est pas limitée par le nombre de cases de déplacement.
   — **Fou (b,B) :** Le fou se déplace uniquement en diagonale et n'est pas limité par le nombre de cases de déplacement.
   — **Tour (r,R) :** La tour est le complémentaire du fou, elle se déplace uniquement en ligne horizontale ou verticale sans limitation du nombre de cases de déplacement.
   — **Cavalier (n,N) :** Le cavalier à le déplacement le plus atypique, il peux sauter au dessus des autres pièces durant son déplacement, il se déplace en "L" dans le sens qu'il veut.



2. Si le mouvement est correct pour la pièce, il doit vérifier que la case ou le chemin venant vers la case n'est pas occupé par une pièce alliée (hormis le cavalier qui doit juste vérifier la case d'arrivée), auquel cas, le mouvement n'est pas valide.
3. Si la case n'est pas occupée par une pièce alliée, le programme vérifie si une pièce ennemie se trouve sur la case. Si c'est le cas la pièce ennemie est détruite du plateau de jeu. A noter que le pion ne "mange" pas de la même façon qu'il se déplace, en effet il ne peut manger qu'en diagonale à distance de 1.
4. Enfin, la pièce se déplace sur la case cible. À noter que si un pion se retrouve au bout de l'échiquier la pièce doit pouvoir être changer en la pièce du choix de l'IA / l'utilisateur.

— **Garder une trace de l'historique :**
Le programme doit garder en mémoire les 6 derniers plateaux de jeux (3 derniers coups de chaque joueurs) afin de les comparer et d'annoncer un match nul si le même plateau est répété 3 fois.

— **Choisir les algorithmes à utiliser :**
En début de partie, une liste des différents algorithmes est proposée à l'utilisateur qui n'a plus qu'à recopier dans le terminal celle contre qui il veut jouer ou tout simplement celles qu'il veut voir s'affronter, ainsi que ses paramètres (profondeur de recherche, attribution des rôles Joueur_1 et Joueur_2 ...).

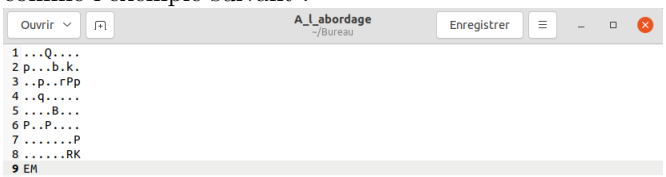
— **Afficher des statistiques :**
Le logiciel doit pouvoir afficher les statistiques de la dernière partie lancée.
Il faudra donc que le logiciel affiche le type d'algorithme utilisé et ses paramètres, quelle ia a gagné ou non, le nombre de tours pour finir la partie, pour chaque tour le temps pris par les ia ou joueurs pour réaliser une action, et le temps moyen d'un tour des ia.

— **Lancer un tournoi entre algorithmes :**
Il est possible de lancer un tournoi entre différents algorithmes afin de voir lequel est le plus performant en rajoutant l'option "-tournament NAME" à l'appel du logiciel. Le tournoi serait une forme de "ligue" ou chaque algorithme jouerait 2 fois contre tous les autres, une fois en tant que joueur blanc une fois en tant que joueur noir.
À la suite de ce tournoi, les informations suivantes sont stockés dans un fichier du même nom que celui du tournoi (NAME) :
- Un classement de tous les algorithmes suivant le nombre de victoires et basé surtout sur le nombre de coups joués jusqu'à la victoire.
- Un cour résumé sur chaque match de la forme suivante " Minmax-ab Blanc 1 MCTS Noir 0 36cp" qui signifie ici une victoire de minmax-ab sur MCTS en 36 coups.

```
MTD(f) Blanc 1 Negamax Noir 0 40cp
```

— **Lancer des Algorithmes sur des énigmes :**
Dans le monde des Échecs il existe des "problèmes" d'échec [**?**], c-à-d d'après un plateau donné il faut réussir à mettre en place des stratégies comme mettre le roi adverse en échec et mat en un certain nombre de coups.

- Les énigmes sont stockées dans le dossier enigm dans un fichier au nom de l'énigme comme l'exemple suivant :



- Dans les fichiers enigm/*, à la suite de l'échiquier de départ se trouve les conditions de victoires de l'énigme : "EM" pour un echec et mat, "EMC5" Si il faut un echec et mat en moins de 5 coups ...

Les options -Enigm list affichent la liste des noms des énigmes connu par le programme. il sera donc possible de lancer les algorithmes sur ces "problèmes". Les problèmes nécessitent

un joueur adverse, le programme utiliseras donc l'algorithme de MCTS en second joueur. Le programme doit donc générer la partie selon le modèle de l'énigme.

3.2 Liste des besoins non-fonctionnels

— **Performances :** Une partie jouée entre deux Algorithmes ne doit pas non plus durer des heures, les algorithmes ne devrons pas dépasser un certain temps pour chaque coup.

— **Facilité d'utilisation :** Toutes les commandes possibles doivent être listées à l'utilisateur, ainsi que leurs fonctions, afin de le guider et qu'il ne perde pas de temps à comprendre le fonctionnement du programme.

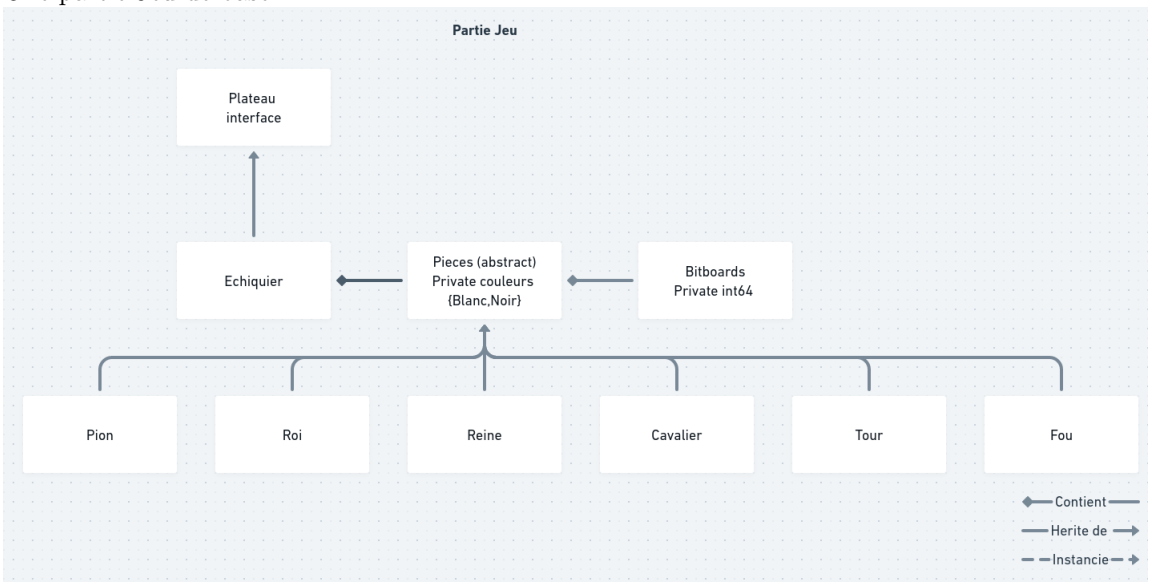
— **Domaine d'action :** Les utilisateurs seront des informaticiens. Le programme étant basé avant tout sur l'interaction d'une IA contre une autre plutôt que du joueur contre une IA.

— **Portabilité :**
Le programme doit être exécutable sous les systèmes d'exploitation suivantes : Ubuntu 20.04, Ubuntu 21.10 ainsi que sur mac OS Catalina.

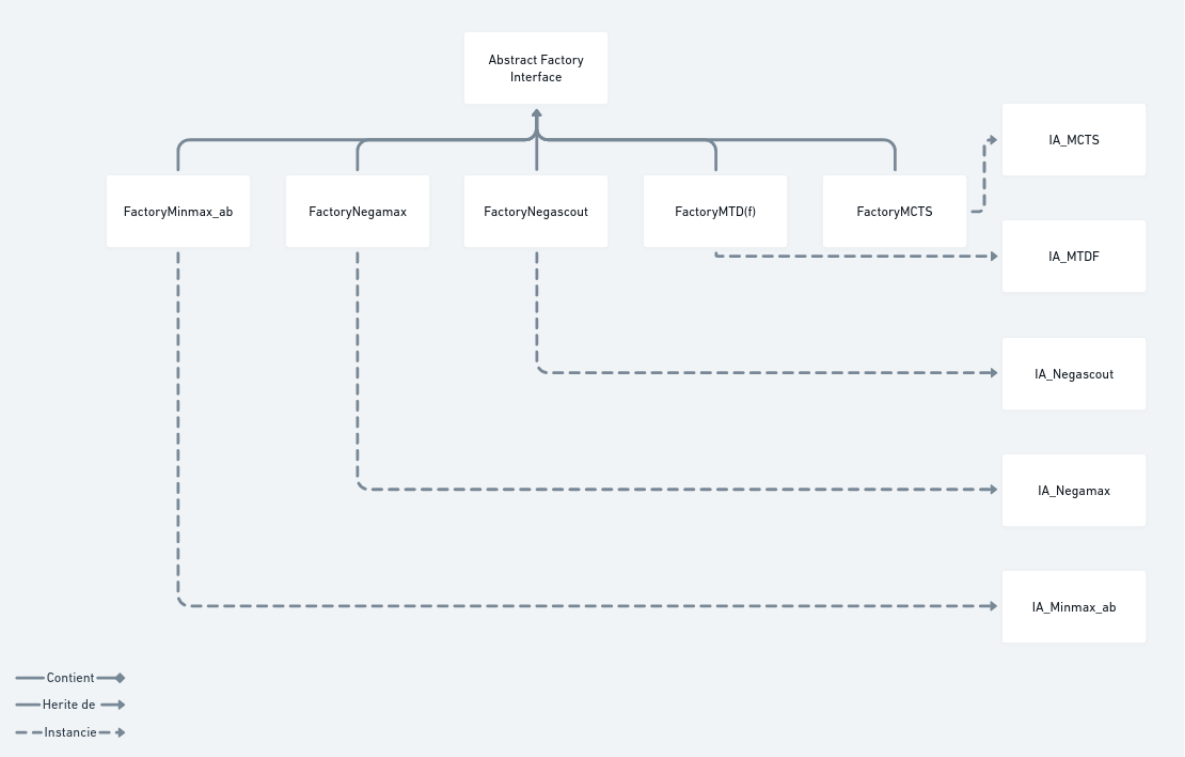
4 Idée de structure

Pour l'instant, nous avons une réflexion sur 2 partie de structure. (En cours de travail)

— Une partie Jeu de base :



— Une partie avec l'abstract factory permettant de générer les différentes "IA" :



5 Liste des outils

— Langage à utiliser : C++
— Utilisation de Bitboards pour représenter et gérer les pièces.

6 Annexe

6.1 Pseudo-code des différentes techniques d'exploration d'arbre classiques :

6.1.1 Minmax-ab :

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            if value ≥ β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            if value ≤ α then
                break (* α cutoff *)
            β := min(β, value)
        return value
```

Source pseudo-code Minmax-ab [**?**].

6.1.2 Negamax :

```
function negamax(node, depth, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    value := −∞
    for each child of node do
        value := max(value, −negamax(child, depth − 1, −color))
    return value
```

```
(* Initial call for Player A's root node *)
negamax(rootNode, depth, 1)
```

```
(* Initial call for Player B's root node *)
negamax(rootNode, depth, -1)
```

Source pseudo-code Negamax [**?**].

6.1.3 Negascout :

```
function pvs(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    for each child of node do
        if child is first child then
            score := −pvs(child, depth − 1, −β, −α, −color)
        else
            score := −pvs(child, depth − 1, −α − 1, −α, −color) (* search with a null window *)
            if α < score < β then
                score := −pvs(child, depth − 1, −β, −score, −color) (* if it failed high, do a full re-search *)
        α := max(α, score)
        if α ≥ β then
            break (* beta cut-off *)
    return α
```

Source pseudo-code Negascout [**?**].

6.1.4 MTD(f) :

```
function MTDF(root, f, d) is
    g := f
    upperBound := +∞
    lowerBound := −∞

    while lowerBound < upperBound do
        β := max(g, lowerBound + 1)
        g := AlphaBetaWithMemory(root, β − 1, β, d)
        if g < β then
            upperBound := g
        else
            lowerBound := g

    return g
```

Source pseudo-code MTD(f) [**?**].

6.1.5 Monte-Carlo Tree Search (MCTS) :

```
class Node:
  def __init__(self, m, p): # move is from parent to node
    self.move, self.parent, self.children = m, p, []
    self.wins, self.visits  = 0, 0

  def expand_node(self, state):
    if not terminal(state):
      for each non-isomorphic legal move m of state:
        nc = Node(m, self) # new child node
        self.children.append(nc)

  def update(self, r):
    self.visits += 1
    if r==win:
      self.wins += 1

  def is_leaf(self):
    return len(self.children)==0

  def has_parent(self):
    return self.parent is not None

def mcts(state):
  root_node  = Node(None, None)
  while time remains:
    n, s = root_node, copy.deepcopy(state)
    while not n.is_leaf():    # select leaf
      n = tree_policy_child(n)
      s.addmove(n.move)
    n.expand_node(s)          # expand
    n = tree_policy_child(n)
    while not terminal(s):    # simulate
      s = simulation_policy_child(s)
    result = evaluate(s)
    while n.has_parent():     # propagate
      n.update(result)
      n = n.parent

return best_move(tree)
```

Source pseudo-code MCTS [**?**].