

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

23 mars 2022



Table des matières

1	Objectifs généraux du projet	2
2	Analyse de l'existant	2
2.1	Algorithmes de recherche	2
2.2	Logiciels de jeu d'échecs existants	3
3	Description des besoins	4
3.1	Liste des besoins fonctionnels	4
3.2	Liste des besoins non-fonctionnels	6
4	Idée de structure	6
5	Liste des outils	7
6	Annexe	7
6.1	Pseudo-code des différentes techniques d'exploration d'arbre classiques :	7
6.1.1	Minmax-ab :	7
6.1.2	Negamax :	8
6.1.3	Negascout :	8
6.1.4	MTD(f) :	8
6.1.5	Monte-Carlo Tree Search (MCTS) :	9

1 Objectifs généraux du projet

Le but du projet est de programmer un joueur d'échecs 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 pièce par couleur stockent donc les positions des différentes pièces du plateau de jeu. Les bitboards permettent également d'indiquer les déplacements légaux avec une plus grande aisance qu'en parcourant le plateau et chaque pièce grâce à des opérations bit à bit simples et efficaces. [1].

Notre objectif étant d'étudier les différents algorithmes et non de programmer un jeu d'échec pour des joueurs humains, nous nous contenterons d'une interface utilisateur en mode texte. 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 [2], Negamax [3], Negascout [4], MTD(f) [5], Monte-Carlo Tree Search (MCTS)[6]. Ainsi qu'au développement d'heuristiques propres au jeu dans l'idée d'éliminer le plus efficacement les branches "perdantes" [7].

2 Analyse de l'existant

2.1 Algorithmes de recherche

Dans le cas de la théorie des jeux, un arbre de jeu est composé de nœuds représentant les positions dans un jeu, les arêtes représentant alors les mouvements. Choisir le meilleur coup revient à rechercher dans l'arbre de jeu l'heuristique la plus pertinente.

Il est possible de trouver le pseudo-code et le principe de fonctionnement des algorithmes Minimax-ab [2], Negamax [3], Negascout [4], MTD(f) [5], Monte-Carlo Tree Search (MCTS)[6] 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.[7].

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. Elles sont des variables qui contiennent à chaque moment du développement de l'arbre, respectivement, la valeur minimale et maximale que le joueur peut espérer obtenir pour le coup à jouer étant donné la position où il se trouve. Ainsi, alpha et bêta permettront à l'algorithme d'ignorer certaines racines de l'arbre. [7].

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[3].

L'algorithme Négascout recherche le premier coup normalement, sur une grande profondeur, puis de repérer le reste des coups possibles, sur une petite profondeur, et de se demander si la valeur de alpha obtenue lors du premier coup est la meilleure[4].

L'algorithme MTD(f) est basé sur la mémorisation d'un arbre de décision formée à partir de l'algorithme Minimax-ab et la formation de deux bornes, une borne supérieure et une borne inférieure. La mémorisation de l'arbre de décision permet de déterminer la valeur heuristique des coups possibles et d'effectuer un second parcours de l'arbre afin de récupérer la valeur de chaque coups et de modifier la valeur des bornes.

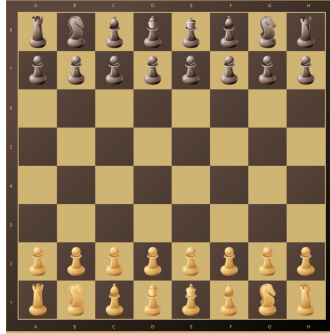
Le but étant de réduire l'intervalle entre les bornes jusqu'à trouver la valeur minimax correspondant au meilleur coup à jouer[5].

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[8] :

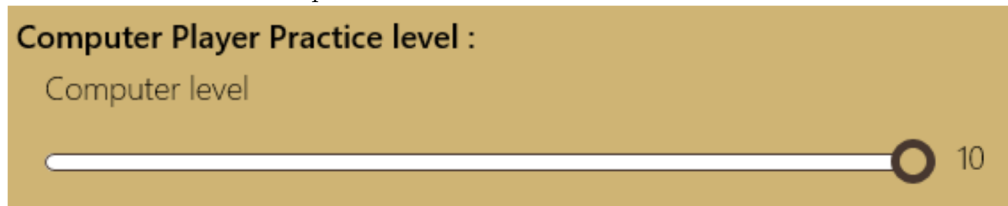
- La sélection successive des enfants de la racine jusqu'à atteindre une feuille.
- L'expansion de l'arbre, en ajoutant un enfant à la feuille si celle-ci n'est pas finale.
- La simulation d'une partie au hasard depuis l'enfant rajouté jusqu'à atteindre une fin de partie.
- La remontée du résultat de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

2.2 Logiciels de jeu d'échecs existants

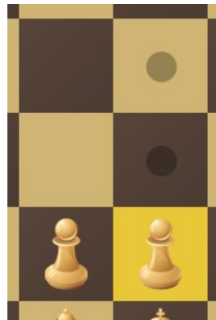


La plupart des logiciels de jeu d'échecs proposés au grand public, disponible dans les boutiques virtuelles comme le Microsoft Store, optent pour un affichage graphique afin de jouer aux échecs, toutefois on peut se rendre compte que ces logiciels permettent aux joueurs de :

- 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 scientifiques, 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 [9] pour calculer ses coups et un système de bitboards pour déterminer les mouvements légaux d'une pièce. [10][11][12]. Nous pouvons citer Deep Thought qui remporta le 19ème championnat ACM d'échecs informatiques, il s'agit du premier ordinateur avec le niveau d'un grand maître qui joua contre Garry Kasparov en 1989 et fut battu.[13].

Jusqu'à Deep Blue, doté d'une architecture massivement parallèle, lui permettant de paralléliser l'algorithme de recherche alpha-bêta[14] et fut le premier ordinateur à avoir battu Kasparov en 1996.[13].

3 Description des besoins

3.1 Liste des besoins fonctionnels

— **Création du plateau de jeu :**

- Représentation du plateau à l'aide de bitboards [1], chaque bitboard représentera l'emplacement des pièces d'un même type et d'une même couleur. 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'échecs original, les numéros représentent les lignes et les lettres pour les colonnes, par exemple (a1 = 1ère case en bas à gauche ...).

r	n	b	q	k	b	n	r
p	p	p	p	p	p	p	p
.
.
.
.
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

— **Lister les mouvements possibles :**

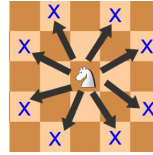
En cas de jeu de l'utilisateur contre un des différents algorithmes, une commande "playable" permet d'afficher la liste de tous les coups jouables pour l'utilisateur (par exemple : h2h4 ...).

r	n	b	q	k	b	n	r
p	p	p	p	p	p	p	p
.
.
.	P	.
.
P	P	P	P	P	P	P	.
R	N	B	Q	K	B	N	R

— **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 pion 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-là 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 limité 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 a le déplacement le plus atypique, il peut 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 d'arrivée et l'ensemble des cases constituant le chemin vers l'arrivée ne sont pas occupés par une pièce alliée (hormis le cavalier qui doit juste vérifier la case d'arrivée). Dans le cas de la présence d'une autre pièce, 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 changée 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 jeu (3 derniers coups de chaque joueur) 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 laquelle 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" où chaque algorithme jouerait 2 fois contre tous les autres, une fois en tant que joueur blanc et une fois en tant que joueur noir.

À la suite de ce tournoi, les informations suivantes sont stockées dans un fichier du même nom que celui du tournoi (NAME) :

- Un classement de tous les algorithmes suivant le nombre de victoire et basé surtout sur le nombre de coups joués jusqu'à la victoire.
- Un court résumé sur chaque match de la forme suivante "Minimax-ab Blanc 1 MCTS Noir 0 36cp" qui signifie ici une victoire de minimax-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 [15], 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 trouvent les conditions de victoire de l'énigme : "EM" pour un échec et mat, "EMC5" si il faut un échec 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 utilisera donc l'algorithme de MCTS en second joueur. Le programme doit donc générer la partie selon le modèle de l'énigme.

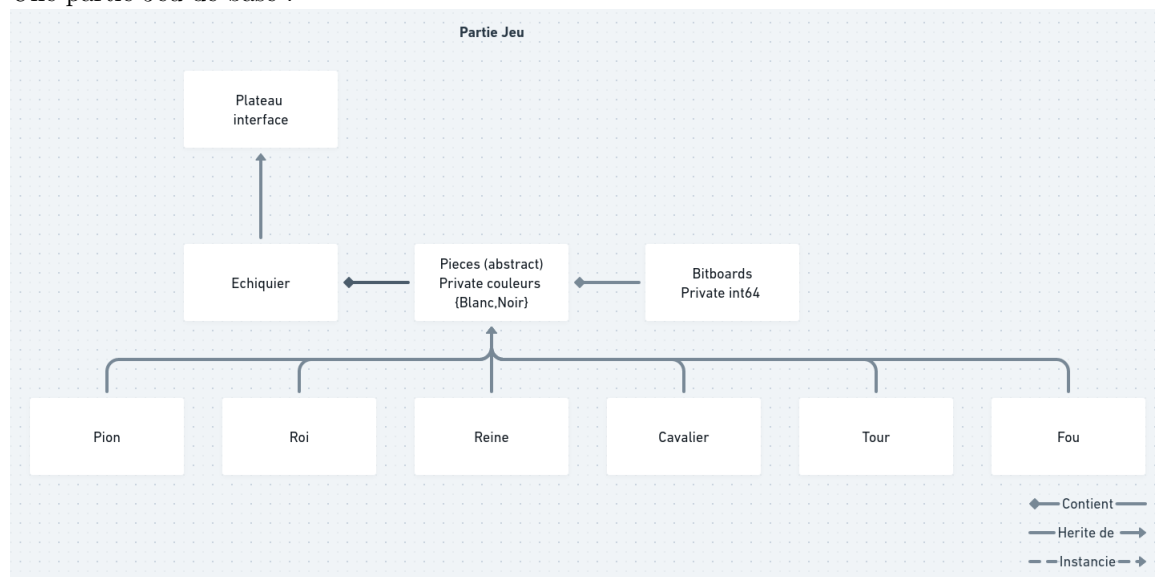
3.2 Liste des besoins non-fonctionnels

- **Performances** : Une partie jouée entre deux Algorithmes ne doit pas non plus durer des heures, les algorithmes ne devons 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.

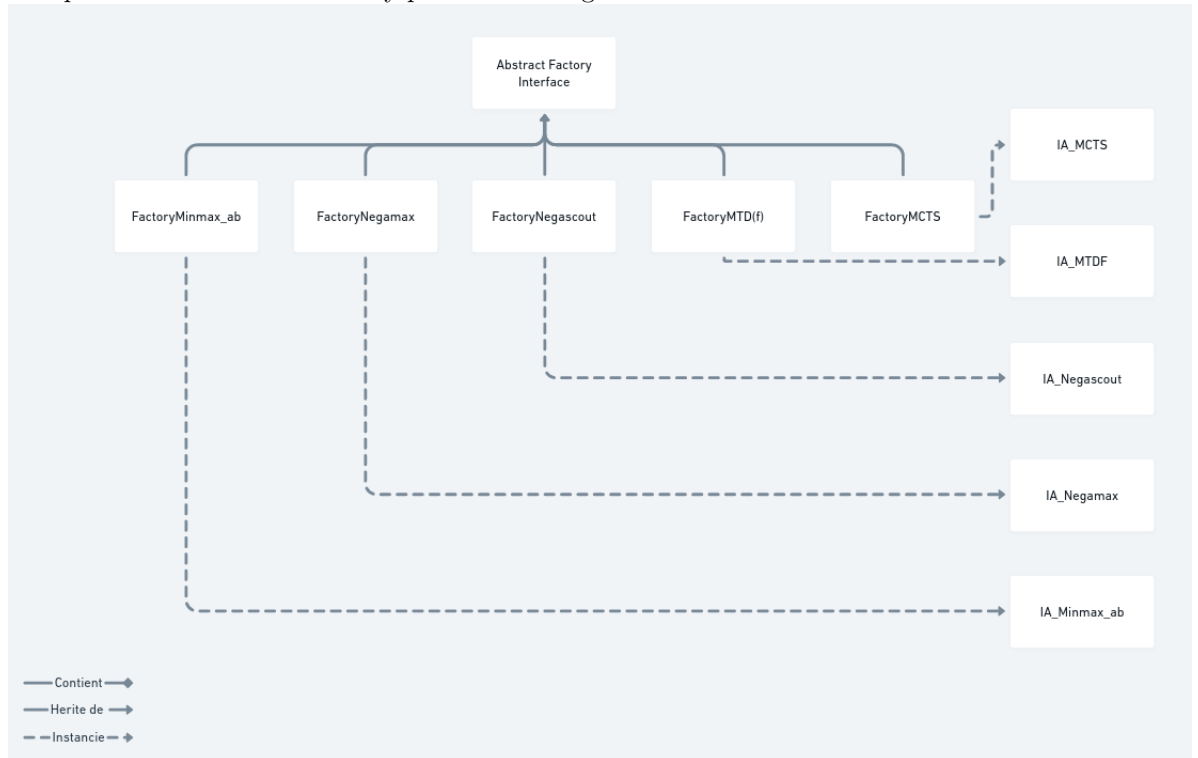
4 Idée de structure

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,  $\alpha$ ,  $\beta$ , maximizingPlayer) is
  if depth = 0 or node is a terminal node then
    return the heuristic value of node
  if maximizingPlayer then
    value :=  $-\infty$ 
    for each child of node do
      value := max(value, alphabeta(child, depth - 1,  $\alpha$ ,  $\beta$ , FALSE))
      if value  $\geq \beta$  then
        break (*  $\beta$  cutoff *)
     $\alpha$  := max( $\alpha$ , value)
    return value
  else
    value :=  $+\infty$ 
    for each child of node do
      value := min(value, alphabeta(child, depth - 1,  $\alpha$ ,  $\beta$ , TRUE))
      if value  $\leq \alpha$  then
        break (*  $\alpha$  cutoff *)
     $\beta$  := min( $\beta$ , value)
    return value
  
```

Source pseudo-code Minmax-ab [16].

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 :=  $-\infty$ 
  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 [17].

6.1.3 Negascout :

```
function pvs(node, depth,  $\alpha$ ,  $\beta$ , 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, - $\beta$ , - $\alpha$ , -color)
    else
      score := -pvs(child, depth - 1, - $\alpha$  - 1, - $\alpha$ , -color) (* search with a null window *)
      if  $\alpha$  < score <  $\beta$  then
        score := -pvs(child, depth - 1, - $\beta$ , -score, -color) (* if it failed high, do a full re-search *)
   $\alpha$  := max( $\alpha$ , score)
  if  $\alpha \geq \beta$  then
    break (* beta cut-off *)
  return  $\alpha$ 
```

Source pseudo-code Negascout [18].

6.1.4 MTD(f) :

```
function MTDf(root, f, d) is
  g := f
  upperBound :=  $+\infty$ 
  lowerBound :=  $-\infty$ 

  while lowerBound < upperBound do
     $\beta$  := max(g, lowerBound + 1)
    g := AlphaBetaWithMemory(root,  $\beta$  - 1,  $\beta$ , d)
    if g <  $\beta$  then
      upperBound := g
    else
      lowerBound := g

  return g
```

Source pseudo-code MTD(f) [19].

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 [20].

Références

- [1] Bitboards. <http://www.frayn.net/beowulf/theory.html#bitboards>. Accessed : 2022-03-13.
- [2] Sangam Vamsi. Minimax algorithm with alpha-beta pruning. <http://theoryofprogramming.com/2017/12/24/minimax-algorithm-with-alpha-beta-pruning/>, 2017. Accessed : 2022-03-13.
- [3] Negamax chessprogramming. <https://www.chessprogramming.org/Negamax>, 2018. Accessed : 2022-03-13.
- [4] Negascout chessprogramming. <https://www.chessprogramming.org/NegaScout>, 2020. Accessed : 2022-03-13.
- [5] Plaat Aske. Mtd(f) a minimax algorithm faster than negascout. <https://people.csail.mit.edu/plaat/mtdf.html>, 1997. Accessed : 2022-03-13.
- [6] Ml — monte carlo tree search (mcts). <https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/>, 2022. Accessed : 2022-03-13.
- [7] Cazenave Tristan Bouzy Bruno and all. Jeux et recherche heuristique, 2014.
- [8] Mark Winands Guillaume Chaslot and all. Progressive strategies for monte-carlo tree search. <https://dke.maastrichtuniversity.nl/m.winands/documents/pMCTS.pdf>, 2008. Accessed : 2022-03-21.
- [9] Monroe Newborn David Levy. All about chess and computers. <https://books.google.fr/books?id=Ao6qCAAQBAJ&pg=PA110&lpg=PA110&dq=kaissa+algorithm&source=b1&ots=49p9c-j4HA&sig=ACfU3U2VTXyVLK0UxGIuzR5Zi0LrPs-N-A&hl=fr&sa=X&ved=2ahUKEwip5q-hscP2AhUnxIUKHbhbC6AQ6AF6BAG0EAM#v=onepage&q=bitboard&f=false>, 2012. Accessed : 2022-03-21.
- [10] Kaissa chessprogramming. <https://www.chessprogramming.org/Kaissa>, 2021. Accessed : 2022-03-21.
- [11] Reijer Grimbergen. Using bitboards for move generation in shogi. <https://www2.teu.ac.jp/gamelab/RESEARCH/ICGAJournal2007.pdf>, 2007. Accessed : 2022-03-21.
- [12] Sam Tannous. Avoiding rotated bitboards with direct lookup. <https://arxiv.org/pdf/0704.3773.pdf>, 2007. Accessed : 2022-03-21.
- [13] Pierre Nolot. Les échecs électroniques : histoire d’une confrontation entre l’humain et la machine. <https://interstices.info/les-echecs-electroniques-histoire-dune-confrontation-entre-lhumain-et-la-machine/>, 2012. Accessed : 2022-03-13.
- [14] Murray Campbell Feng-hsiung Hsu and all. Deep blue, 2002.
- [15] Krabbé Tim. Open chess diary 261-280. https://timkr.home.xs4all.nl/chess2/diary_14.htm, 2005. Accessed : 2022-03-05.
- [16] Elagage alpha beta. https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta. Accessed : 2022-03-13.
- [17] Negamax. <https://en.wikipedia.org/wiki/Negamax>. Accessed : 2022-03-13.
- [18] Principal variation search. https://en.wikipedia.org/wiki/Principal_variation_search. Accessed : 2022-03-13.
- [19] Mtd(f). [https://en.wikipedia.org/wiki/MTD\(f\)](https://en.wikipedia.org/wiki/MTD(f)). Accessed : 2022-03-13.
- [20] monte carlo tree search. <https://webdocs.cs.ualberta.ca/~hayward/396/jem/mcts.html>, 2018. Accessed : 2022-03-13.