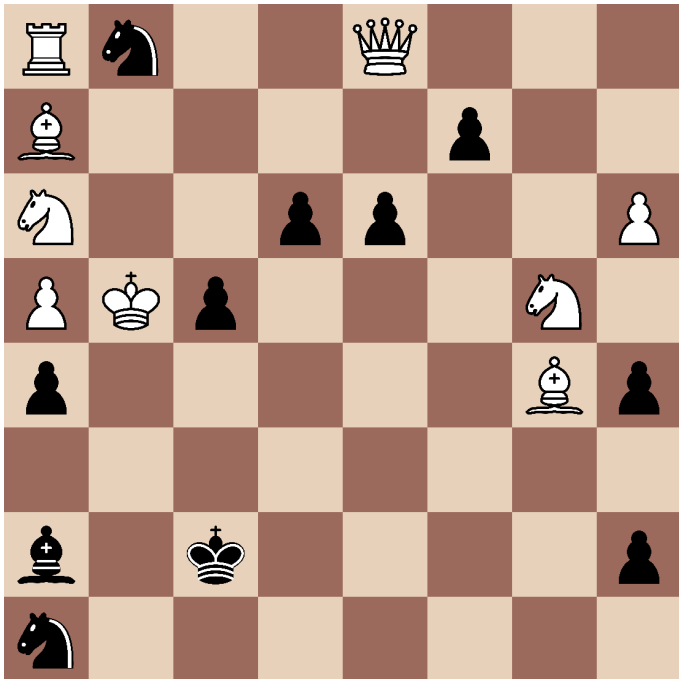
TIPE : Construction d'un moteur d'échecs

Martin CUINGNET
Numéro d'inscription : 22268

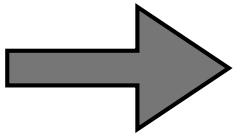
Sommaire

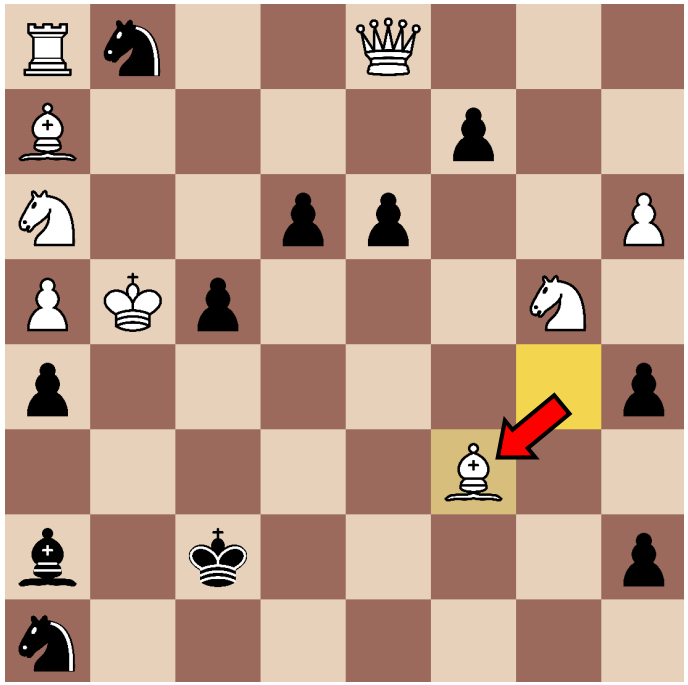
1. Objectifs
2. Structure du moteur
3. Implémentation
4. Heuristique évaluation
5. Recherche et optimisation
6. Score des mouvements
7. Recherche silencieuse
8. Transpositions
9. Approfondissement itératif
10. Résultats
11. Annexe
    - Code source
        - Représentation des pièces
        - Représentation des coups
        - Représentation de l'échiquier et génération des coups possibles
        - Trouver le meilleur coup à partir d'une position
        - Gérer les transpositions
    - Références

Objectifs



À partir d'une position quelconque...

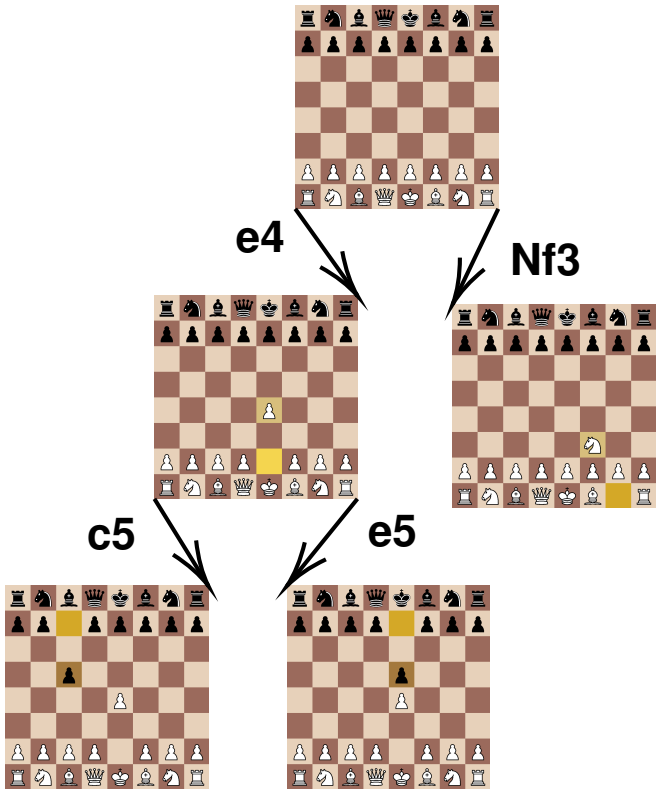




...trouver le meilleur coup

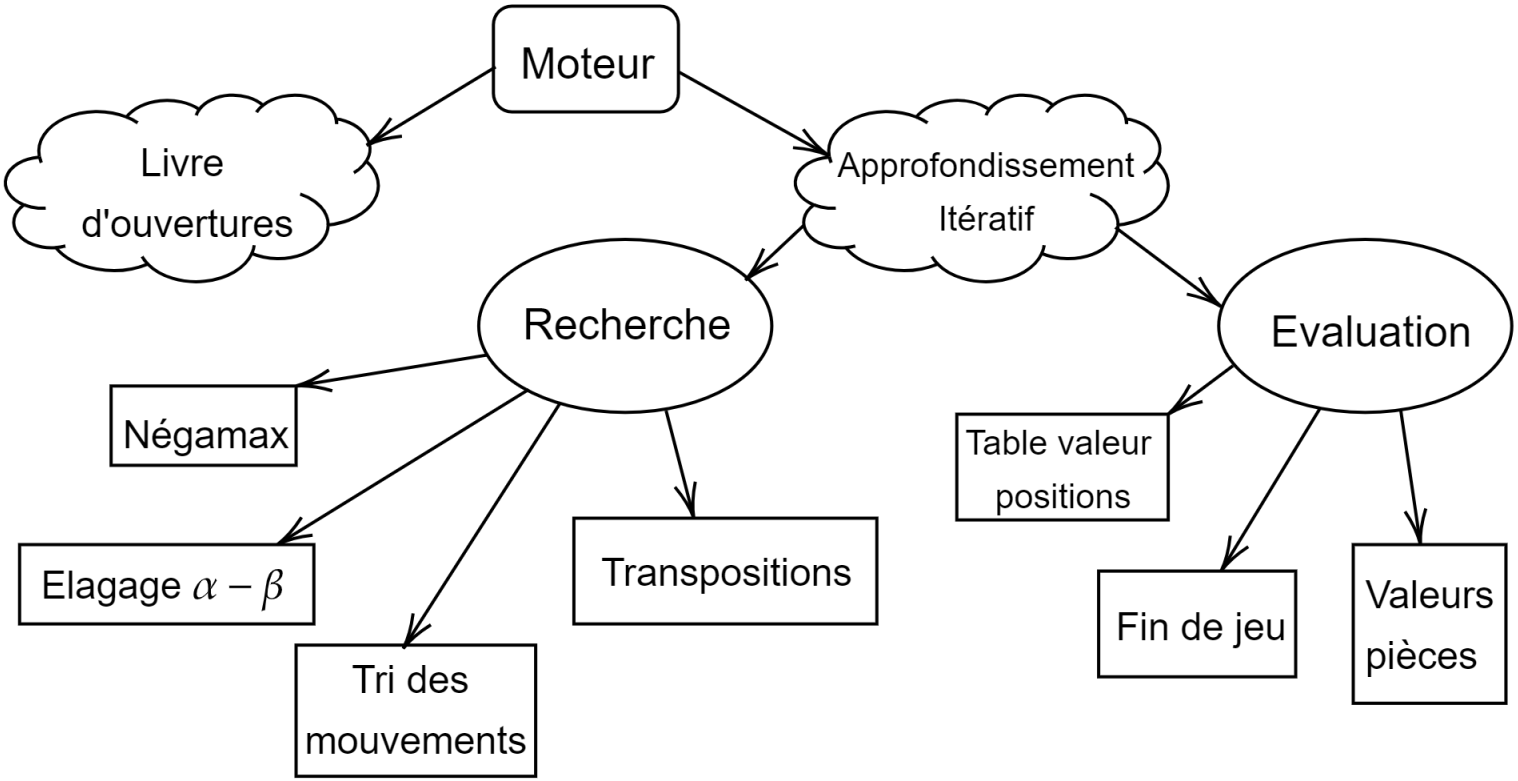
Paradigme utilisé

Recherche couplée à une fonction d'évaluation



Le moteur parcourt l'arbre de jeu jusqu'à atteindre la profondeur maximale où le moteur évalue alors la qualité de la position à l'aide d'une heuristique

Structure générale

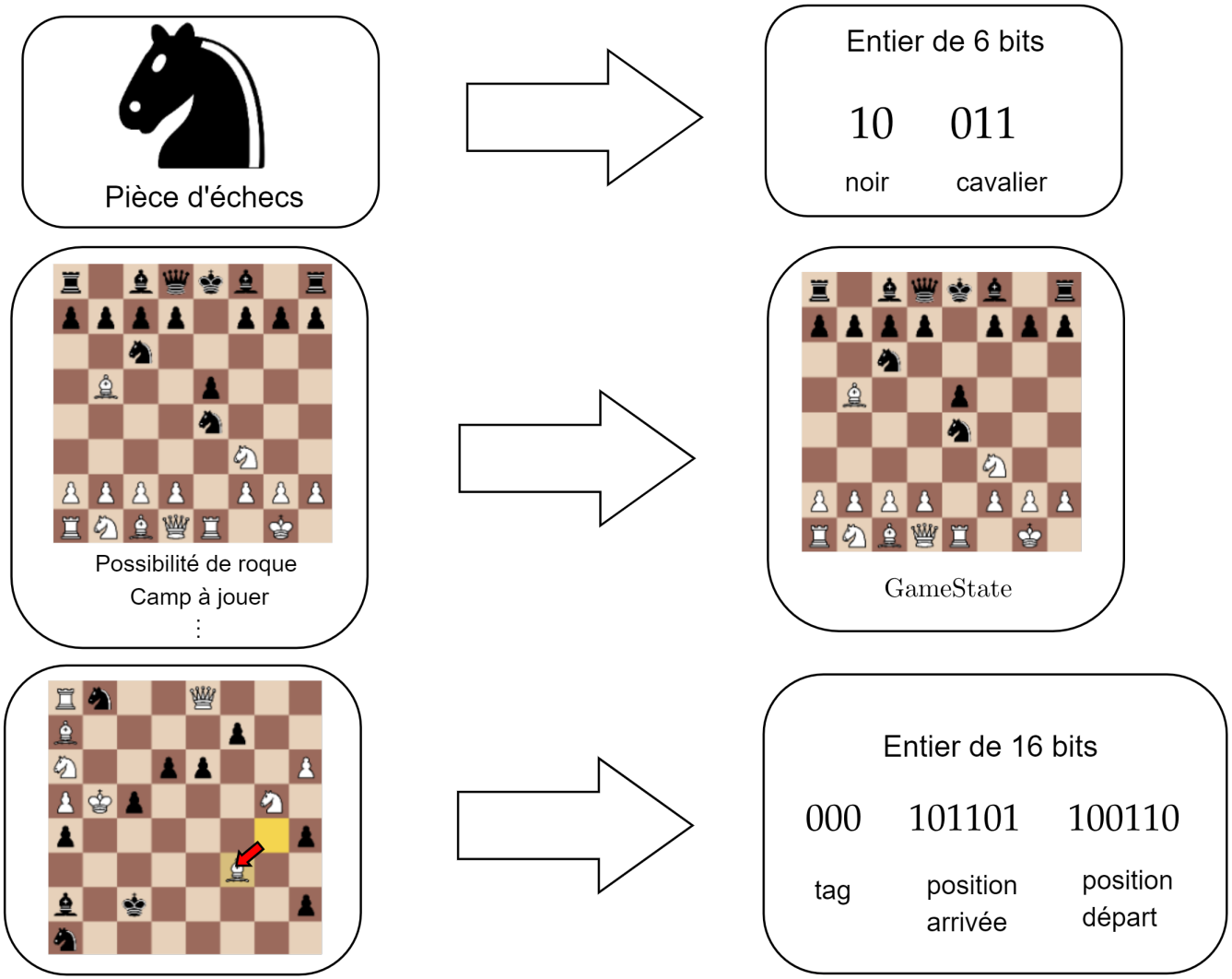


Principe de base de la recherche

Variation de l'algorithme du minimax $\rightarrow$ négamax (somme nulle)

- On suppose que les 2 joueurs jouent de manière parfaite
- Le coup maximisant les chances de victoires du camp devant jouer minimise celles de l'adversaire

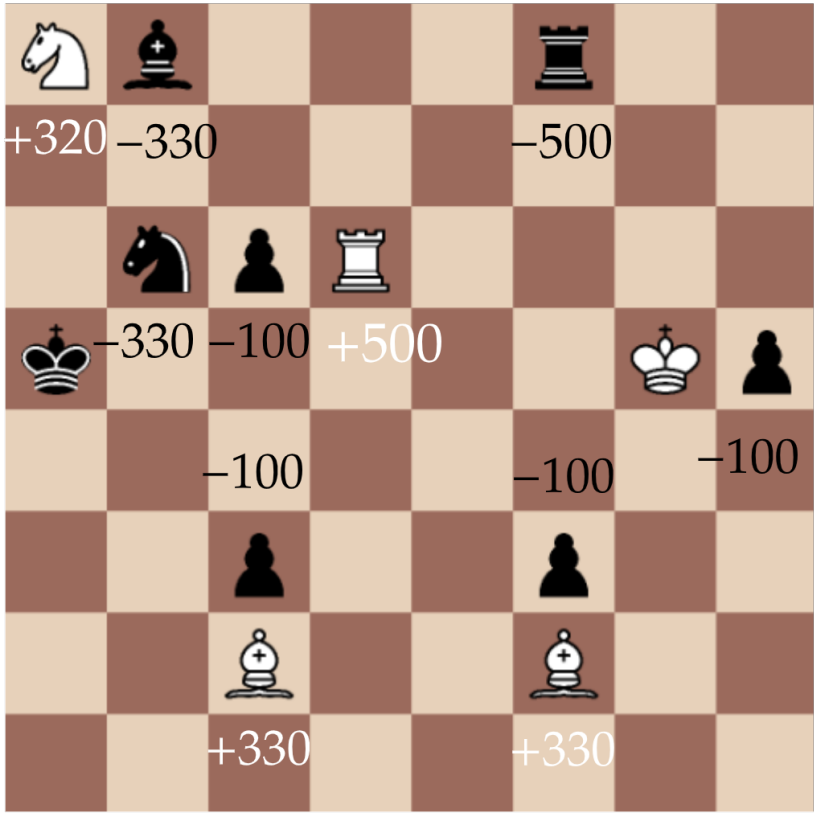
Implémentation pratique



Evaluation basique

Pièce	Valeur
Pion	100
Cavalier	320
Fou	330
Tour	500
Dame	900
Roi	20000

Table – Valeur des pièces proposée dans [Mic21]



Evaluation positionelle

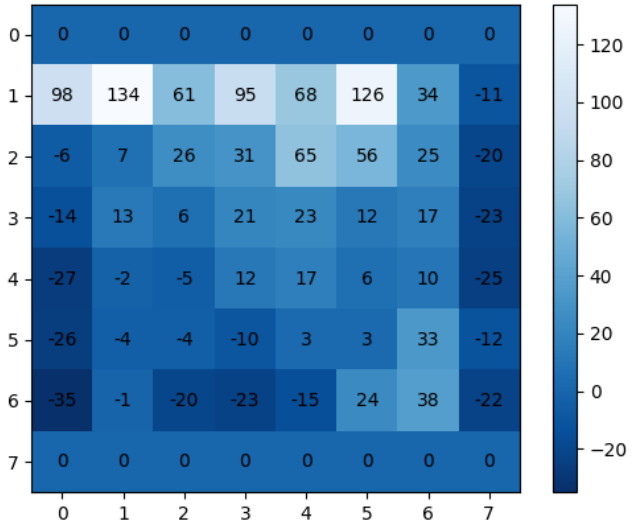


Figure – Table de valeurs pour le pion

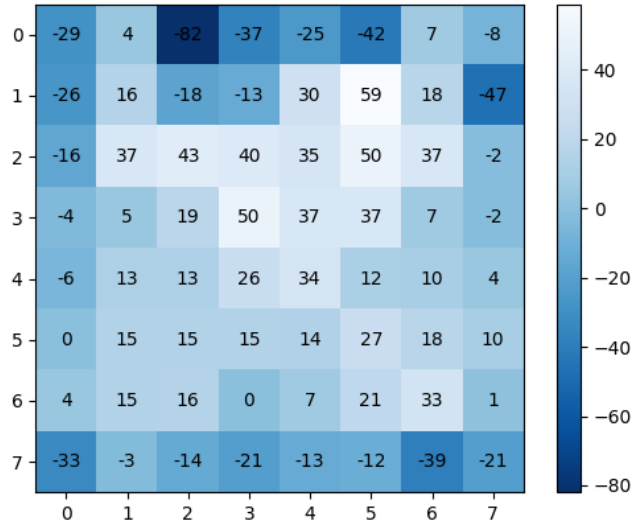


Figure – Table de valeurs pour le fou

Evaluation de fin de jeu

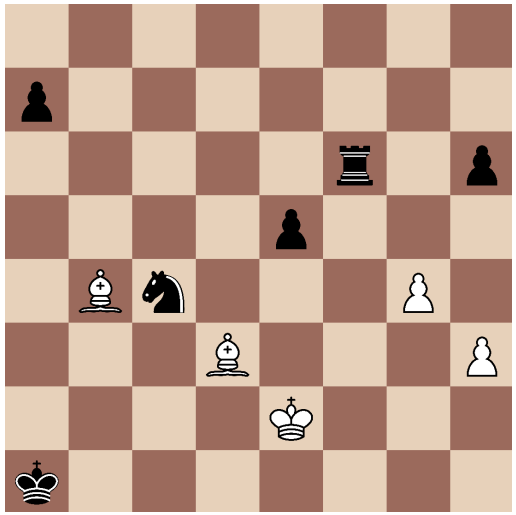


Figure –
Heuristique du bord

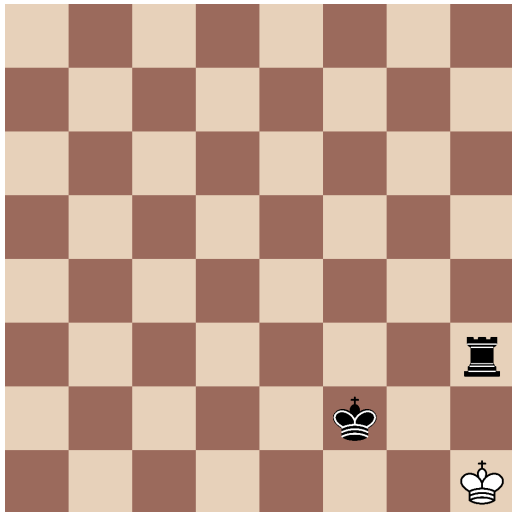
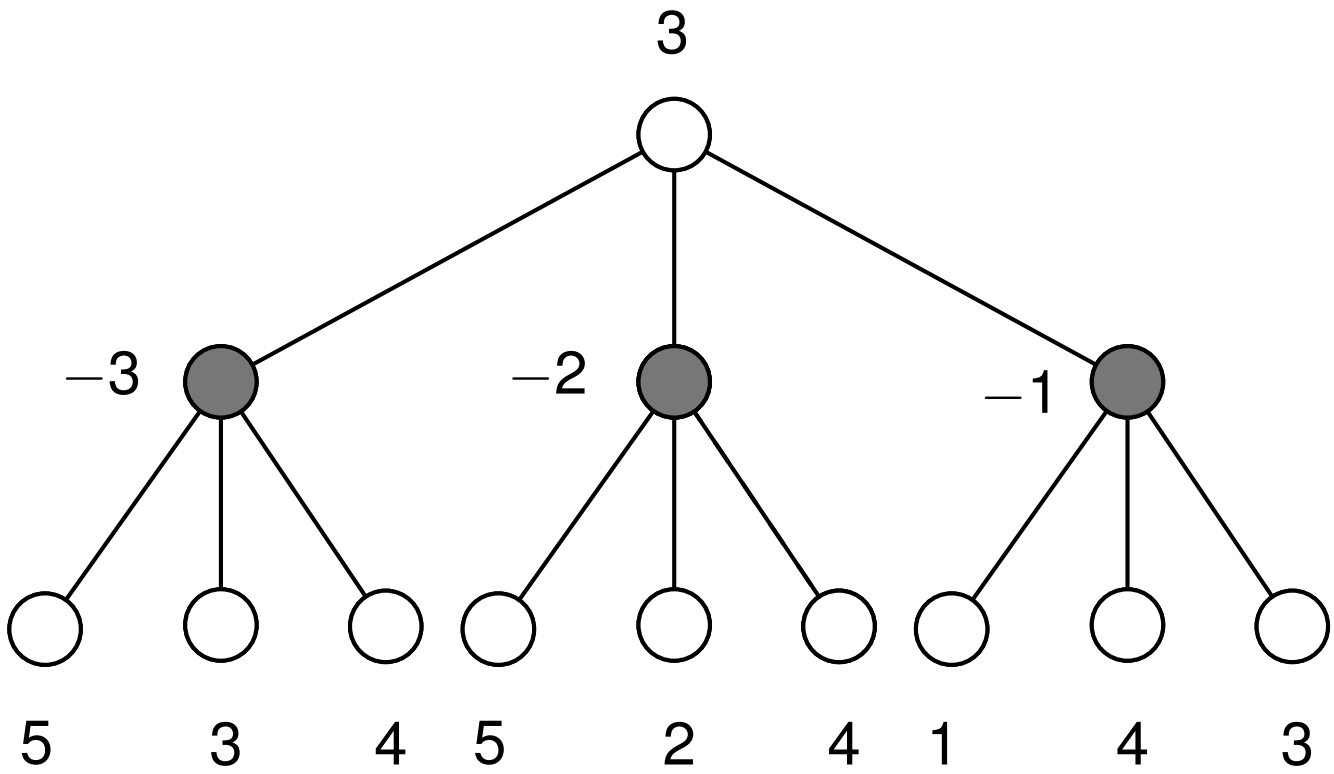
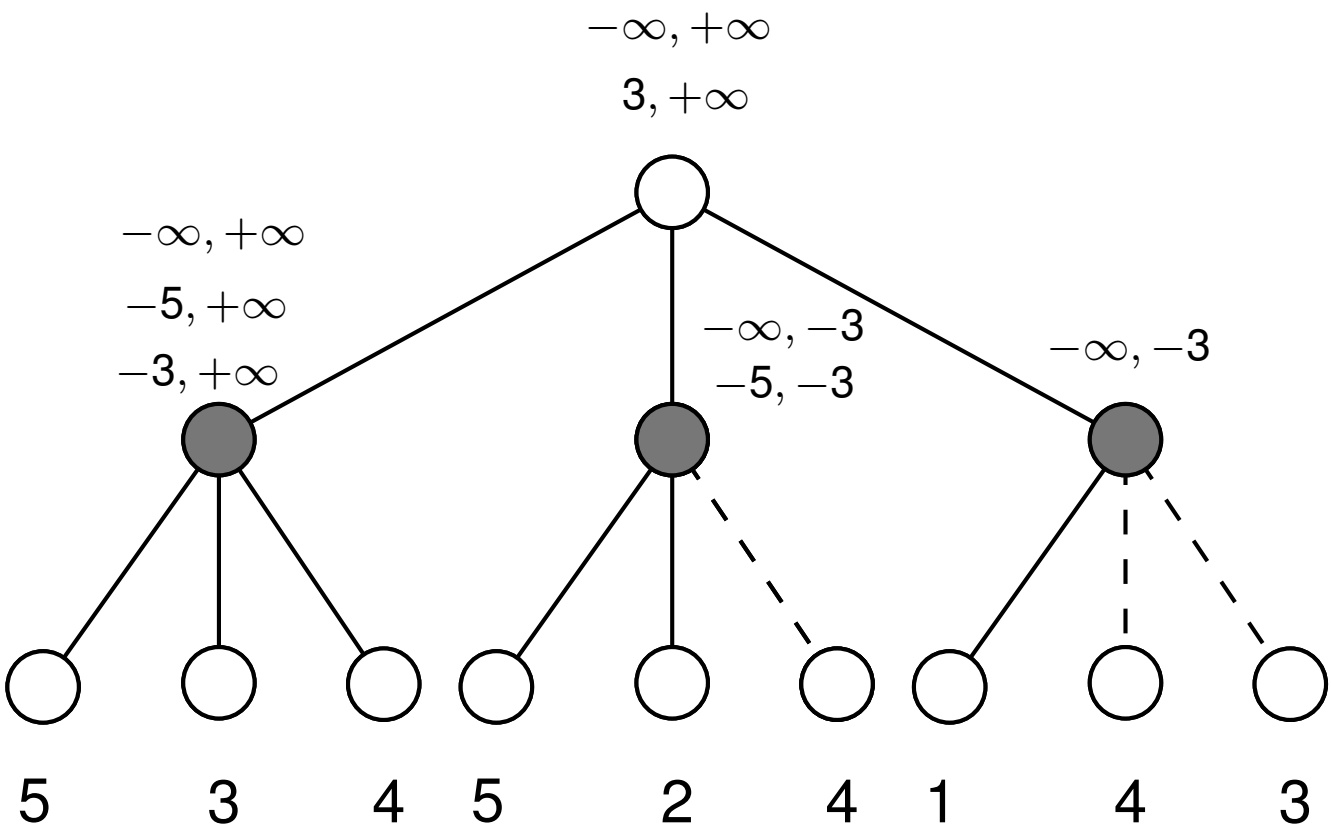


Figure –
Heuristique roi offensif

Exemple de parcours



Elagage $\alpha - \beta$

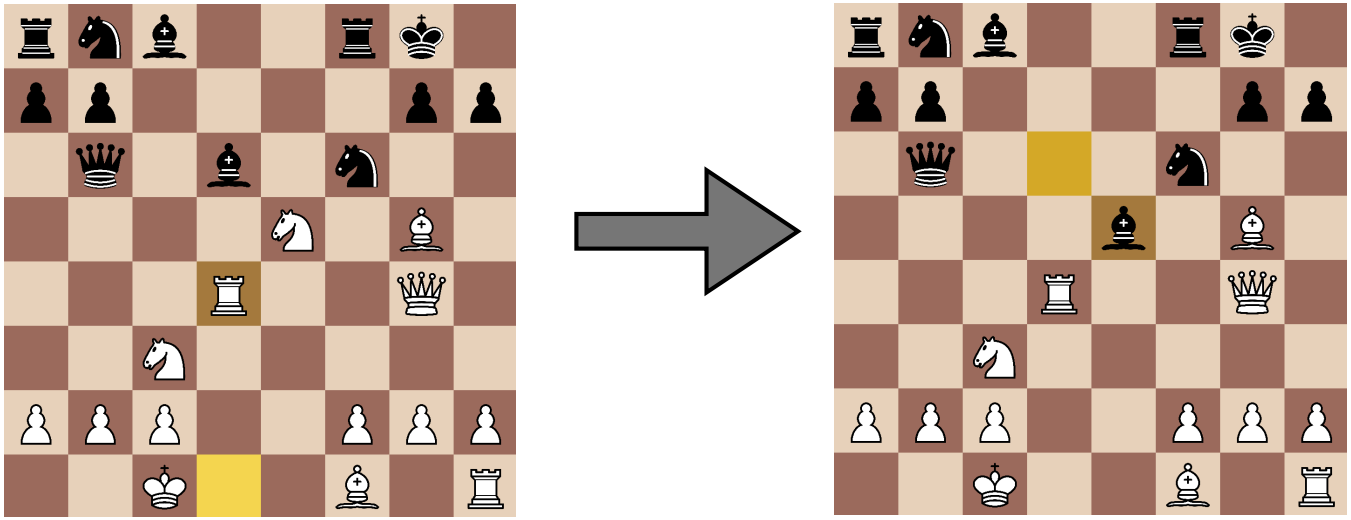


Tri des mouvements

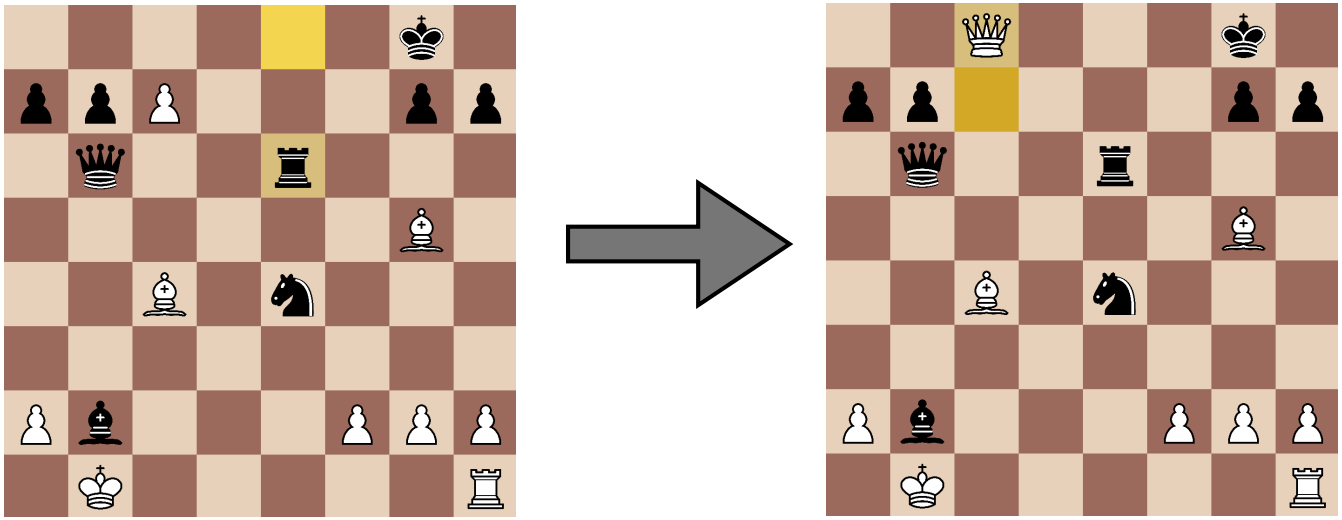
Nombre de noeuds visités $\rightarrow$ dépend de l'ordre de visite

On cherche donc à trier les mouvements possibles depuis une position à explorer $\rightarrow$ nécéssité d'attribuer un score aux mouvements

Score des mouvements



Capture



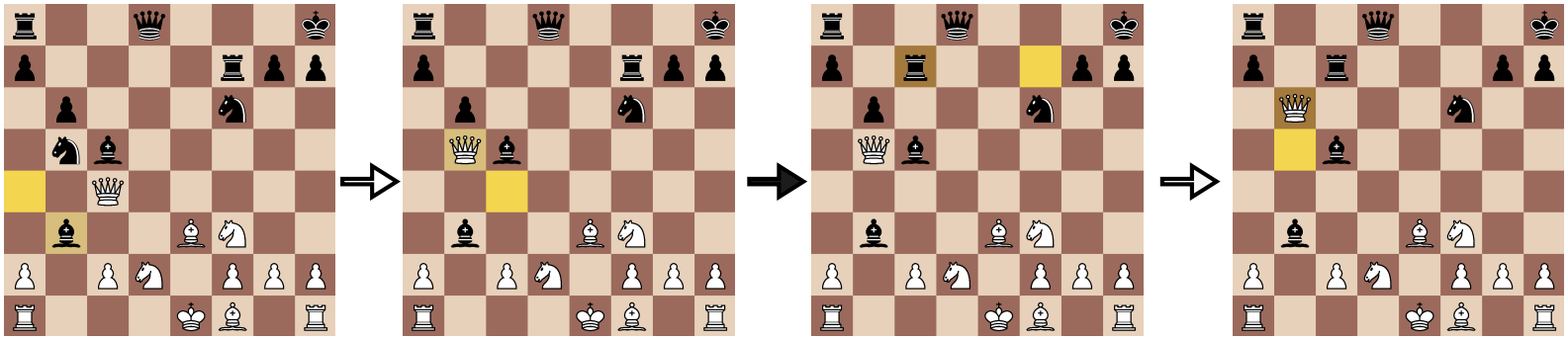
Promotion

L'effet d'horizon

Quand on arrive à la profondeur maximale $\rightarrow$ la fonction d'évaluation ne prend pas en compte les différentes captures

Position initiale

Position de profondeur maximale



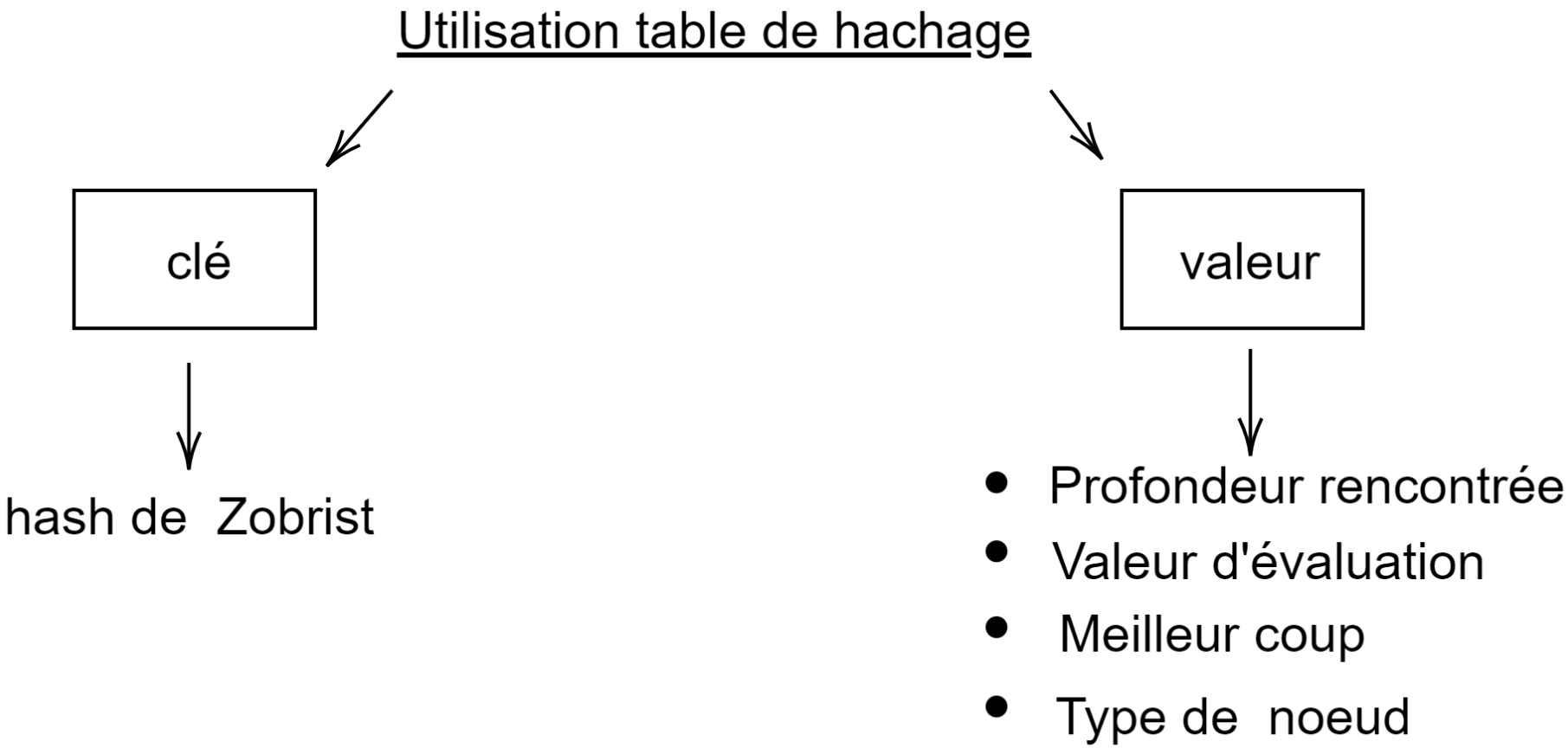
La recherche silencieuse

Pour palier l'effet d'horizon :
Nouvelle recherche après recherche principale $\rightarrow$ Recherche silencieuse
On ne considère que les captures

Mémoïsation

Lors d'une recherche, il est fréquent de rencontrer plusieurs fois une même position $\rightarrow$ Calculs inutiles

Stockage des informations



Hash de Zobrist

Chaque uplet (couleur,pièce,position) $\longrightarrow$ $2 \cdot 6 \cdot 64$

Prises en passants possible $\longrightarrow$ 8

Possibilités de roque $\longrightarrow$ 4

Camp devant jouer $\longrightarrow$ 1

$$h\left(\text{  }\right) = \bigoplus_{\substack{\text{pour chaque élément} \\ \text{de la position}}} \text{aléa}\left[\text{id}_{\text{élément}}\right]$$

Approfondissement itératif

Recherches successives de profondeurs croissantes jusqu'à dépasser le temps imparti :
Le meilleur coup de la dernière recherche ayant terminé est alors renvoyé

Le test STS

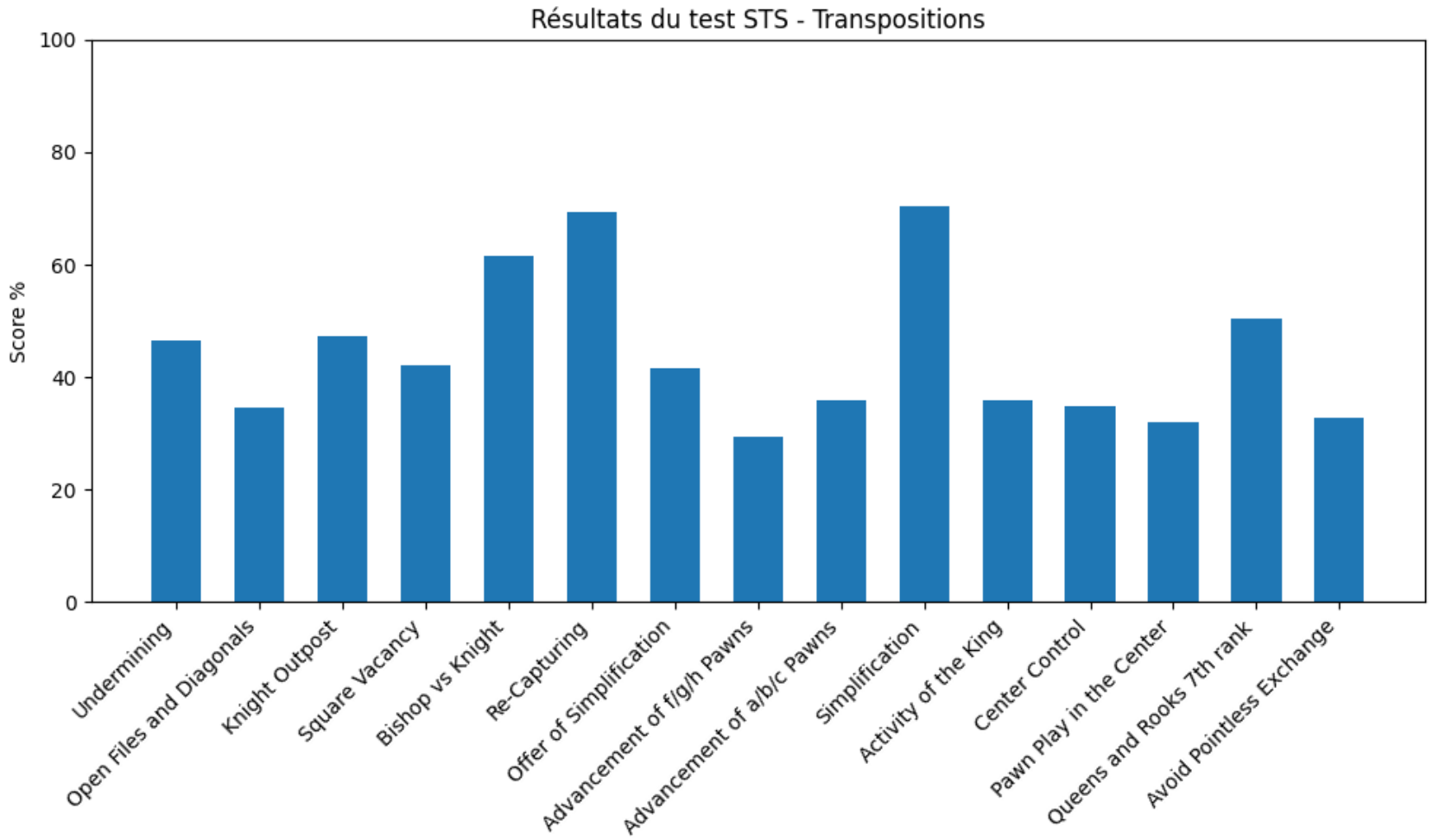


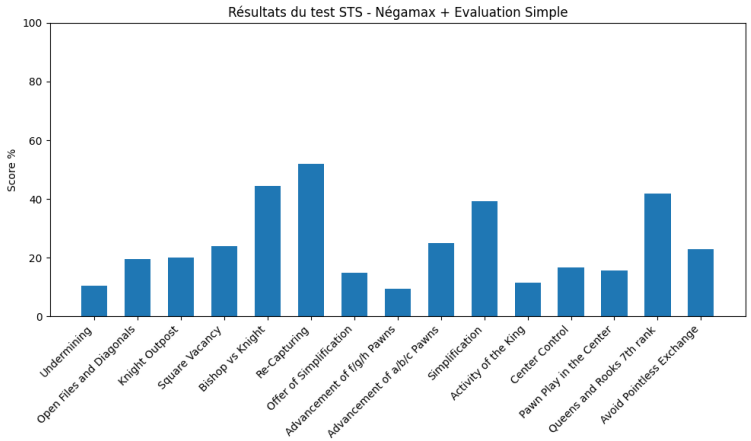
Figure – Résultats du test STS

ELO par améliorations

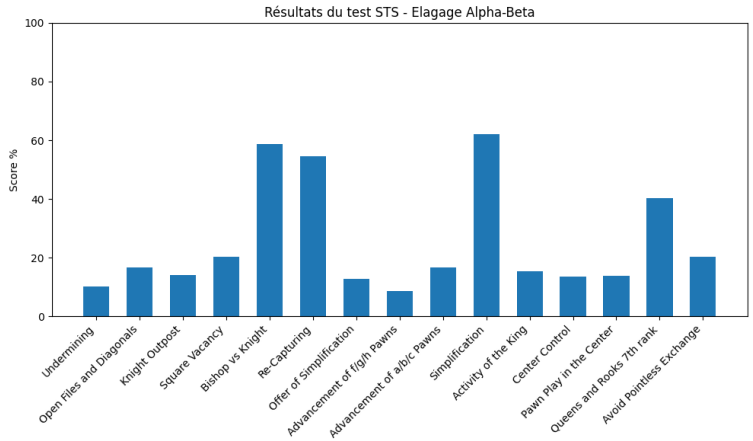
Amélioration	Score ELO
Négamax + Évaluation Simple	850
Élagage Alpha-Bêta	880
Tri des mouvements	893
Recherche Silencieuse	927
Table de valeurs pour les pièces	1377
Heuristique de fin de jeu	1392
Transpositions	1447

Table – Score ELO après chaque amélioration

STS par améliorations I

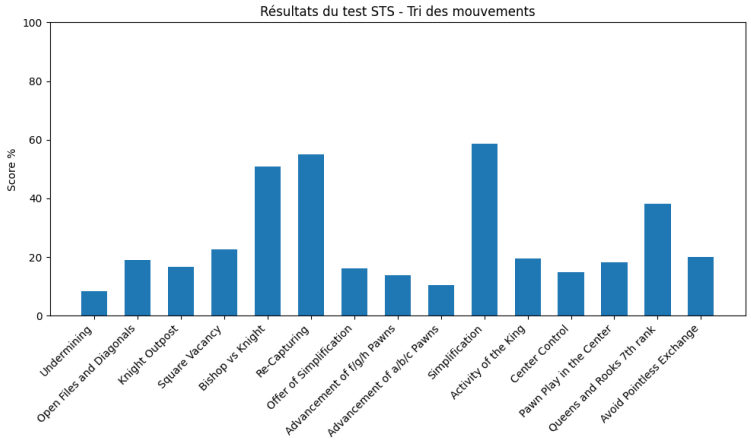


(a) Négamax + Évaluation Simple - ELO 850

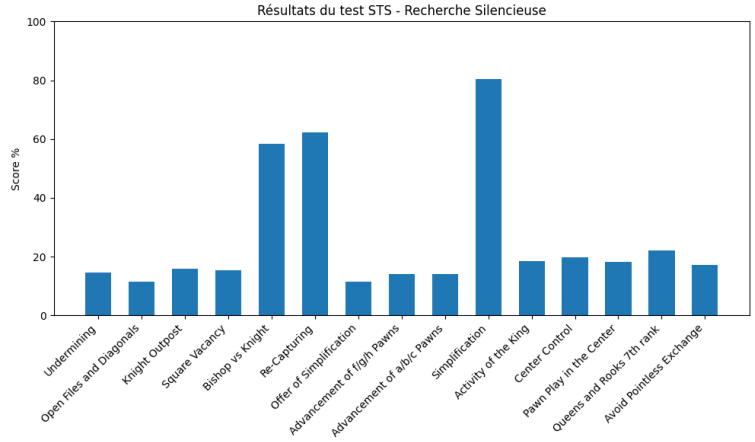


(b) Élagage Alpha-Bêta - ELO 880

STS par améliorations II

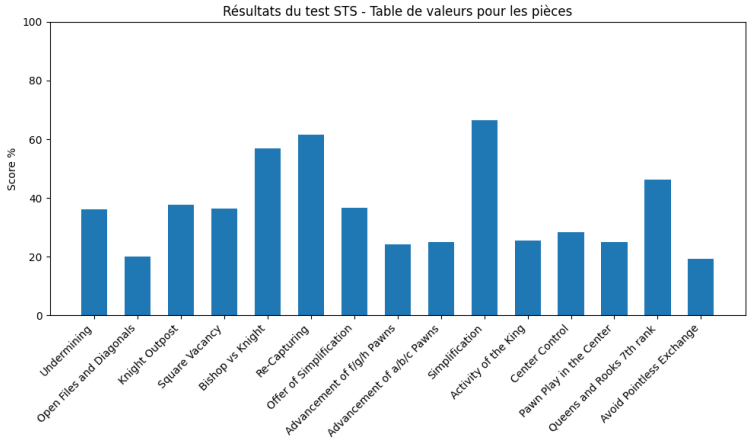


(c) Tri des mouvements - ELO 893

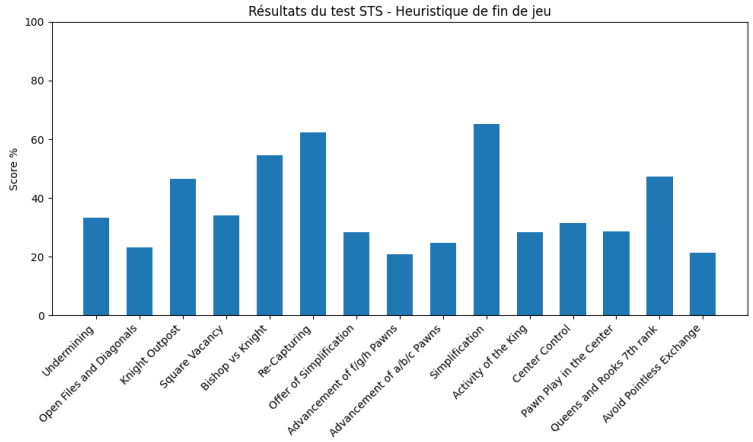


(d) Recherche Silencieuse - ELO 927

STS par améliorations III

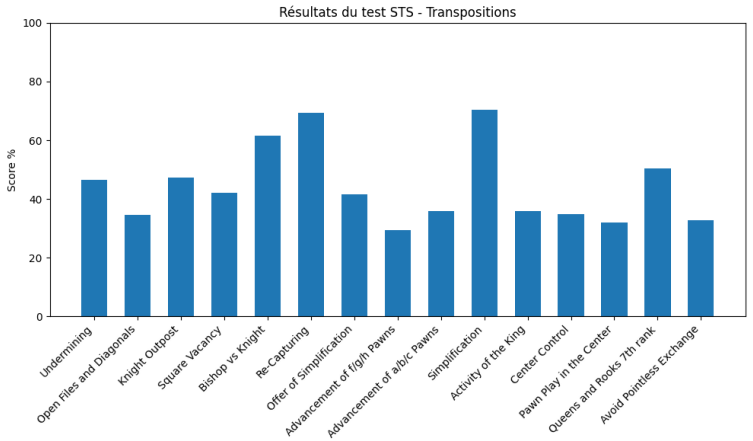


(e) Table de valeurs pour les pièces - ELO 1377



(f) Heuristique de fin de jeu - ELO 1392

STS par améliorations IV



(g) Transpositions - ELO 1447

Figure – Résultats du STS après chaque amélioration

Annexe

Code source : Représentation des pièces I

```
1   enum Pieces
2   {
3     None = 0,
4     Pawn = 1,
5     Rook = 2,
6     Knight = 3,
7     Bishop = 4,
8     Queen = 5,
9     King = 6,
10
11    White = 8,
12    Black = 16
13  };
14
15  bool isPieceWhite(int piece);
16  int pieceType(int piece);
17  int pid(int i, int j);
```

Code source : Représentation des pièces I

```
1   #include "piece.h"
2
3   bool isPieceWhite(int piece){
4     return (piece >> 3) == 1;
5   }
6   int pieceType(int piece){
7     return piece & 7;
8   }
9   int pid(int i, int j){
10    return i + 8 * j;
11  }
```

Code source : Représentation des coups I

```
1   #define positionsMoveMask 4095
2   #define startPosMask 63
3   #define endPosMask 4032
4   #define tagMask 61440
5
6   enum Tag{
7     QuietMove = 0,
8     DoublePawnPush = 1,
9     KingCastle = 2,
10    QueenCastle = 3,
11    Capture = 4,
12    EPCapture = 5,
13    KnightProm = 8,
14    BishopProm = 9,
15    RookProm = 10,
16    QueenProm = 11,
17    KnightPromCapture = 12,
18    BishopPromCapture = 13,
19    RookPromCapture = 14,
20    QueenPromCapture = 15
21  };
22
23  int genMove(int startPos, int endPos, int tag);
24  int genMove(int startPosx, int startPosy, int endPosx, int endPosy, int tag);
25  int discardTag(int move);
26  int endPos(int move);
27  int startPos(int move);
28  int tag(int move);
29  std::string standardNotation(int move);
30  bool isCapturingTag(int tag);
```

Code source : Représentation des coups II

```
31    int standPosToInt(char c1, char c2);
32    int standNotToMove(std::string standNot);
```

Code source : Représentation des coups I

```
1   #include <move.h>
2
3   int genMove(int startPos, int endPos, int tag){
4     if (startPos >= 0 && startPos < 64 && endPos >= 0 && endPos < 64){
5       return startPos | endPos << 6 | tag << 12;
6     }
7     else{
8       return 0;
9     }
10  }
11
12  int genMove(int startPosx, int startPosy, int endPosx, int endPosy, int tag){
13    return (startPosx + 8 * startPosy) | (endPosx + 8 * endPosy) << 6 | tag << 12;
14  }
15  int discardTag(int move){
16    return move & positionsMoveMask;
17  }
18  int endPos(int move){
19    return (move >> 6) & 63;
20  }
21  int startPos(int move){
22    return move & 63;
23  }
24  int tag(int move){
25    return (move & tagMask) >> 12;
26  }
27  std::string standardPos(int pos){
28    std::string res;
29    res.push_back('a' + pos%8);
30    res.push_back('0'+ 7 - pos/8 + 1);
```

Code source : Représentation des coups II

```
31      return res;
32    }
33    std::string standardNotation(int move){
34      int t = tag(move);
35      char c = ' ';
36      switch (t)
37     {
38        case QueenProm:
39          c = 'q';
40          break;
41        case QueenPromCapture:
42          c = 'q';
43          break;
44        case KnightProm:
45          c = 'k';
46          break;
47        case KnightPromCapture:
48          c = 'k';
49          break;
50        case RookProm:
51          c = 'r';
52          break;
53        case RookPromCapture:
54          c = 'r';
55          break;
56        case BishopProm:
57          c = 'b';
58          break;
59        case BishopPromCapture:
```

Code source : Représentation des coups III

```
60          c = 'b';
61          break;
62        }
63        return standardPos(startPos(move)) + standardPos(endPos(move)) + c;
64      }
65      bool isCapturingTag(int tag){
66        if ( (tag <= 3) || ( (8 <= tag) && (tag <= 11) )){
67          return false;
68        }
69        else{
70          return true;
71        }
72      }
73      int standPosToInt(char c1, char c2){
74
75        return (c1 - 'a') + 8 * (7 - (c2 - '1'));
76      }
77      int standNotToMove(std::string standNot){
78
79        return genMove(standPosToInt(standNot[0],standNot[1]),
        ↪  standPosToInt(standNot[2],standNot[3]), 0);
80      }
```

Code source : Représentation de l'échiquier et génération des coups possibles I

```
1   #include "move.h"
2   #include "piece.h"
3
4   #define northMask 7
5   #define southMask 56
6   #define eastMask 448
7   #define westMask 3584
8
9   struct GameState{
10    int capturedPiece;
11    bool canWhiteKingCastle;
12    bool canWhiteQueenCastle;
13    bool canBlackKingCastle;
14    bool canBlackQueenCastle;
15    int doublePushFile;
16    int moveCount;
17    int whiteKingPos;
18    int blackKingPos;
19    uint64_t zobristKey;
20    bool hasWhiteCastled;
21    bool hasBlackCastled;
22  };
23  enum Directions{
24    South = 8,
25    North = -8,
26    East = 1,
27    West = -1,
```

Code source : Représentation de l'échiquier et génération des coups possibles II

```
28      SouthWest = 7,
29      NorthWest = -9,
30      SouthEast = 9,
31      NorthEast = -7
32    };
33    enum DirectionsID{
34      NorthID = 0,
35      SouthID = 1,
36      EastID = 2,
37      WestID = 3,
38      NorthEastID = 4,
39      NorthWestID = 5,
40      SouthEastID = 6,
41      SouthWestID = 7
42    };
43
44    class BoardManager{
45    public:
46      BoardManager();
47
48      void makeMove(int move);
49      void unmakeMove(int move);
50
51      std::vector<int> generatePseudoMoves();
52      std::vector<int> generateMoves(bool onlyCaptures);
53
54      int get(int pos);
```

Code source : Représentation de l'échiquier et génération des coups possibles III

```
55      int get(int x, int y);
56      bool isSquareEmpty(int i, int j);
57      bool isSquareEmpty(int pid);
58      bool isSquareFree(int i, int j);
59      bool isSquareFree(int pid);
60      bool isSquareEnemy(int pid);
61      bool isSquareEnemy(int i, int j);
62      bool isSquareFriendly(int pid);
63
64      std::string startingFen = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0
        ↪  1";
65      void loadFen(std::string fen);
66
67      uint64_t piecesZobrist[8][2][64];
68      uint64_t doublePushFileZobrist[9];
69      uint64_t whiteToMoveZobrist;
70      uint64_t castlingRightZobrist[4];
71
72      void initZobrist();
73      uint64_t computeZobrist();
74      uint64_t zobristKey;
75      std::vector<uint64_t> zobristHistory;
76
77      void controlledSquares();
78      void assign(int i, int j);
79      void resetControl();
80
```

Code source : Représentation de l'échiquier et génération des coups possibles IV

```
81      bool isChecked();
82      bool controlled[8][8];
83
84      bool whiteToMove = true;
85      int board[8][8];
86      GameState currentGameState;
87      std::stack<GameState> gameStateHistory;
88    };
```

Code source : Représentation de l'échiquier et génération des coups possibles I

```
1   #include "boardManager.h"
2   {
3   int numSquares[64][8];
4
5   BoardManager::BoardManager(){
6     for (int i = 0 ; i < 8 ; ++i){
7       for (int j = 0 ; j < 8 ; ++j){
8         int numNorth = j;
9         int numSouth = 7 - j;
10        int numEast = 7 - i;
11        int numWest = i;
12        numSquares[i + j*8][0] = numNorth;
13        numSquares[i + j*8][1] = numSouth;
14        numSquares[i + j*8][2] = numEast;
15        numSquares[i + j*8][3] = numWest;
16        numSquares[i + j*8][4] = min(numNorth, numEast);
17        numSquares[i + j*8][5] = min(numNorth, numWest);
18        numSquares[i + j*8][6] = min(numSouth, numEast);
19        numSquares[i + j*8][7] = min(numSouth, numWest);
20      }
21    }
22    currentGameState.capturedPiece = 0;
23    currentGameState.canWhiteKingCastle = true;
24    currentGameState.canWhiteQueenCastle = true;
25    currentGameState.canBlackKingCastle = true;
26    currentGameState.canBlackQueenCastle = true;
27    currentGameState.hasWhiteCastled = false;
```

Code source : Représentation de l'échiquier et génération des coups possibles II

```
28      currentGameState.hasBlackCastled = false;
29      currentGameState.doublePushFile = 0;
30      currentGameState.moveCount = 0;
31
32      loadFen(startingFen);
33      initZobrist();
34      computeZobrist();
35
36      currentGameState.zobristKey = zobristKey;
37    }
38
39    bool BoardManager::isChecked(){
40      whiteToMove = !whiteToMove;
41        controlledSquares();
42        whiteToMove = !whiteToMove;
43        if (whiteToMove){
44          return controlled[currentGameState.whiteKingPos /
            ↪  8][currentGameState.whiteKingPos % 8];
45        }
46        else{
47          return controlled[currentGameState.blackKingPos /
            ↪  8][currentGameState.blackKingPos % 8];
48        }
49    }
50
51    bool BoardManager::isSquareEmpty(int i, int j){
52      return get(i,j) == 0;
```

Code source : Représentation de l'échiquier et génération des coups possibles III

```
53    }
54    bool BoardManager::isSquareEmpty(int pid){
55      return get(pid) == 0;
56    }
57    bool BoardManager::isSquareEnemy(int i, int j){
58      return isPieceWhite(get(i,j)) != whiteToMove && (get(i,j) > 0);
59    }
60    bool BoardManager::isSquareEnemy(int pid){
61      return isPieceWhite(get(pid)) != whiteToMove && (get(pid) > 0);
62    }
63    bool BoardManager::isSquareFree(int i, int j){
64      return isPieceWhite(get(i,j)) != whiteToMove || (get(i,j) == 0);
65    }
66    bool BoardManager::isSquareFree(int pid){
67      return isPieceWhite(get(pid)) != whiteToMove || (get(pid) == 0);
68    }
69    bool BoardManager::isSquareFriendly(int pid){
70      return isPieceWhite(get(pid)) == whiteToMove && (get(pid) > 0);
71    }
72
73    void BoardManager::assign(int i, int j){
74      if (i >= 0 && i < 8 && j >= 0 && j < 8){
75        controlled[i][j] = true;
76      }
77    }
78
79    void BoardManager::controlledSquares(){
```

Code source : Représentation de l'échiquier et génération des coups possibles IV

```
80
81      resetControl();
82
83      for (int i = 0 ; i < 8 ; ++i){
84        for (int j = 0 ; j < 8 ; ++j){
85          int piece = get(i,j);
86          int currentPID = pid(i,j);
87
88          if(piece > 0 && isPieceWhite(piece) == whiteToMove){
89
90            if (pieceType(piece) == Pawn){
91
92              if (isPieceWhite(piece)){
93                if (numSquares[currentPID][NorthEastID] >= 1 && isSquareFree(i+1,j-1)){
94                  assign(j-1,i+1);
95                }
96                if (numSquares[currentPID][NorthWestID] >= 1 && isSquareFree(i-1,j-1)){
97                  assign(j-1,i-1);
98                }
99
100             }
101             else{
102               if (numSquares[currentPID][SouthEastID] >= 1 && isSquareFree(i+1,j+1)){
103                 assign(j+1, i+1);
104               }
105               if (numSquares[currentPID][SouthWestID] >= 1 && isSquareFree(i-1,j+1)){
106                 assign(j+1,i-1);
```

Code source : Représentation de l'échiquier et génération des coups possibles V

```
107             }
108           }
109
110         }
111
112         if (pieceType(piece) == King){
113           for (int dirID = 0 ; dirID <= 7 ; ++dirID){
114             int targetPos = currentPID + directions[dirID];
115
116             if (numSquares[currentPID][dirID] >= 1 && isSquareFree(targetPos)){
117               assign(targetPos/8, targetPos % 8);
118             }
119           }
120         }
121
122         if (pieceType(piece) == Knight){
123           if (numSquares[currentPID][NorthID] >= 2 && numSquares[currentPID][EastID]
              ↪  >= 1 && isSquareFree(i+1,j-2)){
124             assign(j-2,i+1);
125           }
126           if (numSquares[currentPID][NorthID] >= 2 && numSquares[currentPID][WestID]
              ↪  >= 1 && isSquareFree(i-1,j-2)){
127             assign(j-2,i-1);
128           }
129           if (numSquares[currentPID][SouthID] >= 2 && numSquares[currentPID][EastID]
              ↪  >= 1 && isSquareFree(i+1,j+2)){
130             assign(j+2,i+1);
```

Code source : Représentation de l'échiquier et génération des coups possibles VI

```
131             }
132             if (numSquares[currentPID][SouthID] >= 2 && numSquares[currentPID][WestID]
                ↪  >= 1 && isSquareFree(i-1,j+2)){
133                 assign(j+2,i-1);
134             }
135             if (numSquares[currentPID][EastID] >= 2 && numSquares[currentPID][NorthID]
                ↪  >= 1 && isSquareFree(i+2,j-1)){
136                 assign(j-1,i+2);
137             }
138             if (numSquares[currentPID][EastID] >= 2 && numSquares[currentPID][SouthID]
                ↪  >= 1 && isSquareFree(i+2,j+1)){
139                 assign(j+1,i+2);
140             }
141             if (numSquares[currentPID][WestID] >= 2 && numSquares[currentPID][NorthID]
                ↪  >= 1 && isSquareFree(i-2,j-1)){
142                 assign(j-1,i-2);
143             }
144             if (numSquares[currentPID][WestID] >= 2 && numSquares[currentPID][SouthID]
                ↪  >= 1 && isSquareFree(i-2,j+1)){
145                 assign(j+1,i - 2);
146             }
147         }
148
149         if (pieceType(piece) == Rook || pieceType(piece) == Bishop ||
            ↪  pieceType(piece) == Queen){
150
151           int startDir = (pieceType(piece) == Bishop) ? 4 : 0;
```

Code source : Représentation de l'échiquier et génération des coups possibles VII

```
152             int endDir = (pieceType(piece) == Rook) ? 3 : 7;
153
154             for (int dirID = startDir ; dirID <= endDir ; ++dirID){
155               for (int i = 0 ; i < numSquares[currentPID][dirID]; ++i){
156                 int targetPos = currentPID + directions[dirID] * (i+1);
157
158                 if (isSquareFriendly(targetPos)){
159                   break;
160                 }
161
162                 assign(targetPos / 8, targetPos % 8);
163
164                 if (isSquareEnemy(targetPos)){
165                   break;
166                 }
167               }
168             }
169           }
170         }
171       }
172     }
173
174   }
175
176   void BoardManager::resetControl(){
177     for (int i = 0 ; i < 64; ++i){
178       controlled[i % 8][i / 8] = false;
```

Code source : Représentation de l'échiquier et génération des coups possibles VIII

```
179       }
180     }
181
182     std::vector<int> BoardManager::generateMoves(bool onlyCaptures){
183       std::vector<int> pseudoMoves = generatePseudoMoves();
184       std::vector<int> legalMoves;
185
186       for (int pseudoMove : pseudoMoves){
187         if (onlyCaptures && !isCapturingTag(tag(pseudoMove))){
188           continue;
189         }
190         makeMove(pseudoMove);
191         whiteToMove = !whiteToMove;
192         if (!isChecked()){
193           legalMoves.push_back(pseudoMove);
194         }
195         whiteToMove = !whiteToMove;
196         unmakeMove(pseudoMove);
197       }
198       return legalMoves;
199     }
200
201
202     std::vector<int> BoardManager::generatePseudoMoves(){
203       std::vector<int> moves;
204
205       whiteToMove = !whiteToMove;
```

Code source : Représentation de l'échiquier et génération des coups possibles IX

```
206     controlledSquares();
207     whiteToMove = !whiteToMove;
208
209     for (int i = 0 ; i < 8 ; ++i){
210       for (int j = 0 ; j < 8 ; ++j){
211         int piece = get(i,j);
212         int currentPID = pid(i,j);
213
214         if(piece > 0 && isPieceWhite(piece) == whiteToMove){
215           if (pieceType(piece) == Pawn){
216
217             if (isPieceWhite(piece)){
218               if (numSquares[currentPID][NorthID] >= 1 && isSquareEmpty(i,j-1)){
219                 if (j > 1){
220                   moves.push_back(genMove(i,j,i,j-1, QuietMove));
221                 }
222                 else{
223                   moves.push_back(genMove(i,j,i,j-1, KnightProm));
224                   moves.push_back(genMove(i,j,i,j-1, BishopProm));
225                   moves.push_back(genMove(i,j,i,j-1, RookProm));
226                   moves.push_back(genMove(i,j,i,j-1, QueenProm));
227                 }
228               }
229               if (j == 6 && isSquareEmpty(i,j-2) && isSquareEmpty(i, j-1)){
230                 moves.push_back(genMove(i,j,i,j-2, DoublePawnPush));
231               }
232               if (numSquares[currentPID][NorthEastID] >= 1 && isSquareEnemy(i+1,j-1)){
```

Code source : Représentation de l'échiquier et génération des coups possibles X

```
233                 if (j - 1 == 0){
234                   moves.push_back(genMove(i,j,i+1,j-1, KnightPromCapture));
235                   moves.push_back(genMove(i,j,i+1,j-1, BishopPromCapture));
236                   moves.push_back(genMove(i,j,i+1,j-1, RookPromCapture));
237                   moves.push_back(genMove(i,j,i+1,j-1, QueenPromCapture));
238                 }
239                 else{
240                   moves.push_back(genMove(i,j,i+1,j-1, Capture));
241                 }
242               }
243               if (numSquares[currentPID][NorthWestID] >= 1 && isSquareEnemy(i-1,j-1)){
244                 if (j - 1 == 0){
245                   moves.push_back(genMove(i,j,i-1,j-1, KnightPromCapture));
246                   moves.push_back(genMove(i,j,i-1,j-1, BishopPromCapture));
247                   moves.push_back(genMove(i,j,i-1,j-1, RookPromCapture));
248                   moves.push_back(genMove(i,j,i-1,j-1, QueenPromCapture));
249                 }
250                 else{
251                   moves.push_back(genMove(i,j,i-1,j-1, Capture));
252                 }
253               }
254
255               if (numSquares[currentPID][NorthEastID] >= 1 && j == 3 &&
                  ↪  isSquareEnemy(i+1,j) && pieceType(get(i+1,j)) == Pawn &&
                  ↪  isSquareEmpty(i+1, j-1) && currentGameState.doublePushFile - 1 ==
                  ↪  i+1){
256                 moves.push_back(genMove(i,j,i+1,j-1, EPCapture));
```

Code source : Représentation de l'échiquier et génération des coups possibles XI

```
257                 }
258                 if (numSquares[currentPID][NorthWestID] >= 1 && j == 3 &&
                    ↪  isSquareEnemy(i-1,j) && pieceType(get(i-1,j)) == Pawn &&
                    ↪  isSquareEmpty(i-1, j-1) && currentGameState.doublePushFile - 1 ==
                    ↪  i-1){
259                   moves.push_back(genMove(i,j,i-1,j-1, EPCapture));
260                 }
261               }
262               else{
263                 if (numSquares[currentPID][NorthID] >= 1 && isSquareEmpty(i,j+1)){
264                   if (j < 6){
265                     moves.push_back(genMove(i,j,i,j+1, QuietMove));
266                   }
267                   else{
268                     moves.push_back(genMove(i,j,i,j+1, KnightProm));
269                     moves.push_back(genMove(i,j,i,j+1, BishopProm));
270                     moves.push_back(genMove(i,j,i,j+1, RookProm));
271                     moves.push_back(genMove(i,j,i,j+1, QueenProm));
272                   }
273                 }
274                 if (j == 1 && isSquareEmpty(i,j+2) && isSquareEmpty(i, j+1)){
275                   moves.push_back(genMove(i,j,i,j+2, DoublePawnPush));
276                 }
277                 if (numSquares[currentPID][SouthEastID] >= 1 && isSquareEnemy(i+1,j+1)){
278                   if (j + 1 == 7){
279                     moves.push_back(genMove(i,j,i+1,j+1, KnightPromCapture));
280                     moves.push_back(genMove(i,j,i+1,j+1, BishopPromCapture));
```

Code source : Représentation de l'échiquier et génération des coups possibles XII

```
281                 moves.push_back(genMove(i,j,i+1,j+1, RookPromCapture));
282                 moves.push_back(genMove(i,j,i+1,j+1, QueenPromCapture));
283               }
284               else{
285                 moves.push_back(genMove(i,j,i+1,j+1, Capture));
286               }
287             }
288             if (numSquares[currentPID][SouthWestID] >= 1 && isSquareEnemy(i-1,j+1)){
289               if (j + 1 == 7){
290                 moves.push_back(genMove(i,j,i-1,j+1, KnightPromCapture));
291                 moves.push_back(genMove(i,j,i-1,j+1, BishopPromCapture));
292                 moves.push_back(genMove(i,j,i-1,j+1, RookPromCapture));
293                 moves.push_back(genMove(i,j,i-1,j+1, QueenPromCapture));
294               }
295               else{
296                 moves.push_back(genMove(i,j,i-1,j+1, Capture));
297               }
298             }
299
300             if (numSquares[currentPID][SouthEastID] >= 1 && j == 4 &&
               ↪  isSquareEnemy(i+1,j)&& pieceType(get(i+1,j)) == Pawn &&
               ↪  isSquareEmpty(i+1, j+1) && (currentGameState.doublePushFile - 1 ==
               ↪  i+1)){
301               moves.push_back(genMove(i,j,i+1,j+1, EPCapture));
302             }
303             if (numSquares[currentPID][SouthWestID] >= 1 && j == 4 &&
               ↪  isSquareEnemy(i-1,j) && pieceType(get(i-1,j)) == Pawn &&
               ↪  isSquareEmpty(i-1, j+1) && (currentGameState.doublePushFile - 1 ==
               ↪  i-1)){
```

Code source : Représentation de l'échiquier et génération des coups possibles XIII

```
304                 moves.push_back(genMove(i,j,i-1,j+1, EPCapture));
305               }
306             }
307           }
308
309           if (pieceType(piece) == King){
310
311             for (int dirID = 0 ; dirID <= 7 ; ++dirID){
312               int targetPos = currentPID + directions[dirID];
313
314               if (numSquares[currentPID][dirID] >= 1 && isSquareFree(targetPos)){
315                 moves.push_back(genMove(currentPID, targetPos, isSquareEnemy(targetPos)
                  ↪  * Capture));
316               }
317             }
318             if (!controlled[j][i]){
319               if ((currentGameState.canWhiteQueenCastle && whiteToMove) ||
                  ↪  (currentGameState.canBlackQueenCastle && !whiteToMove)){
320                 bool isOK = true;
321                 if (!isSquareEmpty(1,j)){
322                   isOK = false;
323                 }
324                 for (int p = 2; p <= 3; ++p ){
325                   if (!isSquareEmpty(p, j) || controlled[j][p]){
326                     isOK = false;
327                   }
328                 }
```

Code source : Représentation de l'échiquier et génération des coups possibles XIV

```
329                 if (isOK){
330                   moves.push_back(genMove(currentPID, currentPID - 2, QueenCastle));
331                 }
332               }
333               if ((currentGameState.canWhiteKingCastle && whiteToMove) ||
                  ↪  (currentGameState.canBlackKingCastle && !whiteToMove)){
334                 bool isOK = true;
335                 for (int p = 5; p <= 6; ++p ){
336                   if (!isSquareEmpty(p, j) || controlled[j][p]){
337                     isOK = false;
338                   }
339                 }
340                 if (isOK){
341                   moves.push_back(genMove(currentPID, currentPID + 2, KingCastle));
342                 }
343               }
344             }
345           }
346
347           if (pieceType(piece) == Knight){
348
349             if (numSquares[currentPID][NorthID] >= 2 && numSquares[currentPID][EastID]
                  ↪  >= 1 && isSquareFree(i+1,j-2)){
350               moves.push_back(genMove(i,j,i+1,j-2, isSquareEnemy(i+1,j-2) * Capture));
351             }
352             if (numSquares[currentPID][NorthID] >= 2 && numSquares[currentPID][WestID]
                  ↪  >= 1 && isSquareFree(i-1,j-2)){
```

Code source : Représentation de l'échiquier et génération des coups possibles XV

```
353             moves.push_back(genMove(i,j,i-1,j-2, isSquareEnemy(i-1,j-2) * Capture));
354           }
355           if (numSquares[currentPID][SouthID] >= 2 && numSquares[currentPID][EastID]
              ↪  >= 1 && isSquareFree(i+1,j+2)){
356             moves.push_back(genMove(i,j,i+1,j+2, isSquareEnemy(i+1,j+2) * Capture));
357           }
358           if (numSquares[currentPID][SouthID] >= 2 && numSquares[currentPID][WestID]
              ↪  >= 1 && isSquareFree(i-1,j+2)){
359             moves.push_back(genMove(i,j,i-1,j+2, isSquareEnemy(i-1,j+2) * Capture));
360           }
361
362           if (numSquares[currentPID][EastID] >= 2 && numSquares[currentPID][NorthID]
              ↪  >= 1 && isSquareFree(i+2,j-1)){
363             moves.push_back(genMove(i,j,i+2,j-1, isSquareEnemy(i+2,j-1) * Capture));
364           }
365           if (numSquares[currentPID][EastID] >= 2 && numSquares[currentPID][SouthID]
              ↪  >= 1 && isSquareFree(i+2,j+1)){
366             moves.push_back(genMove(i,j,i+2,j+1, isSquareEnemy(i+2,j+1) * Capture));
367           }
368           if (numSquares[currentPID][WestID] >= 2 && numSquares[currentPID][NorthID]
              ↪  >= 1 && isSquareFree(i-2,j-1)){
369             moves.push_back(genMove(i,j,i-2,j-1, isSquareEnemy(i-2,j-1) * Capture));
370           }
371           if (numSquares[currentPID][WestID] >= 2 && numSquares[currentPID][SouthID]
              ↪  >= 1 && isSquareFree(i-2,j+1)){
372             moves.push_back(genMove(i,j,i-2,j+1, isSquareEnemy(i-2,j+1) * Capture));
373           }
```

Code source : Représentation de l'échiquier et génération des coups possibles XVI

```
374
375             }
376
377             if (pieceType(piece) == Rook || pieceType(piece) == Bishop ||
                ↪  pieceType(piece) == Queen){
378
379               int startDir = (pieceType(piece) == Bishop) ? 4 : 0;
380               int endDir = (pieceType(piece) == Rook) ? 3 : 7;
381
382               for (int dirID = startDir ; dirID <= endDir ; ++dirID){
383                 for (int i = 0 ; i < numSquares[currentPID][dirID]; ++i){
384                   int targetPos = currentPID + directions[dirID] * (i+1);
385
386                   if (isSquareFriendly(targetPos)){
387                     break;
388                   }
389                   moves.push_back(genMove(currentPID, targetPos, isSquareEnemy(targetPos)
                      ↪  * Capture));
390
391                   if (isSquareEnemy(targetPos)){
392                     break;
393                   }
394                 }
395               }
396             }
397           }
398         }
```

Code source : Représentation de l'échiquier et génération des coups possibles XVII

```
399       }
400
401       return moves;
402     }
403
404     int BoardManager::isLegal(std::vector<int> moves, int move){
405       for (int movei : moves){
406         if (discardTag(movei) == move){
407           return movei;
408         }
409       }
410       return 0;
411     }
412
413     void BoardManager::loadFen(std::string fen){
414       gameStateHistory = std::stack<GameState>();
415
416       currentGameState.canBlackQueenCastle = false;
417       currentGameState.canWhiteQueenCastle = false;
418       currentGameState.canWhiteKingCastle = false;
419       currentGameState.canBlackKingCastle = false;
420
421       for (int ii = 0 ; ii <= 7; ++ii){
422         for (int jj = 0; jj <= 7; ++jj){
423           board[jj][ii] = None;
424         }
425       }
```

Code source : Représentation de l'échiquier et génération des coups possibles XVIII

```
426
427     int state = 0;
428     int i = 0;
429     int j = 0;
430
431     bool hasReadNb = false;
432     int nb = 0;
433
434     for(char c : fen){
435       if (c == ' '){
436         state +=1;
437         continue;
438       }
439       if (state == 0){
440         switch (c){
441           case '/':
442               j +=1;
443               i = 0;
444              break;
445           case '1' : case '2' : case '3' : case '4' : case '5' : case '6' : case '7' :
              ↪  case '8' :
446             i += c - '0';
447             break;
448           case 'r':
449             board[j][i] = Black | Rook;
450             i += 1;
451             break;
```

Code source : Représentation de l'échiquier et génération des coups possibles XIX

```
452             case 'n':
453               board[j][i] = Black | Knight;
454               i += 1;
455               break;
456             case 'b':
457               board[j][i] = Black | Bishop;
458               i += 1;
459               break;
460             case 'q':
461               board[j][i] = Black | Queen;
462               i += 1;
463               break;
464             case 'k':
465               board[j][i] = Black | King;
466               currentGameState.blackKingPos = i + j*8;
467               i += 1;
468               break;
469             case 'p':
470               board[j][i] = Black | Pawn;
471               i += 1;
472               break;
473             case 'R':
474               board[j][i] = White | Rook;
475               i += 1;
476               break;
477             case 'N':
478               board[j][i] = White | Knight;
```

Code source : Représentation de l'échiquier et génération des coups possibles XX

```
479             i += 1;
480             break;
481           case 'B':
482             board[j][i] = White | Bishop;
483             i += 1;
484             break;
485           case 'Q':
486             board[j][i] = White | Queen;
487             i += 1;
488             break;
489           case 'K':
490             board[j][i] = White | King;
491             currentGameState.whiteKingPos = i + j*8;
492             i += 1;
493             break;
494           case 'P':
495             board[j][i] = White | Pawn;
496             i += 1;
497             break;
498
499         }
500       }
501       if (state == 1){
502         whiteToMove = (c == 'w');
503       }
504       if (state == 2){
505         switch (c){
```

Code source : Représentation de l'échiquier et génération des coups possibles XXI

```
506             case 'Q':
507               currentGameState.canWhiteQueenCastle = true;
508               break;
509             case 'K':
510               currentGameState.canWhiteKingCastle = true;
511               break;
512             case 'q':
513               currentGameState.canBlackQueenCastle = true;
514               break;
515             case 'k':
516               currentGameState.canBlackKingCastle = true;
517               break;
518           }
519         }
520         if (state == 3){
521           switch (c){
522             case '1' : case '2' : case '3' : case '4' : case '5' : case '6' : case '7' :
                ↪  case '8' :
523               currentGameState.doublePushFile = (c - '0');
524               break;
525           }
526         }
527         if (state == 4){
528           if (!hasReadNb){
529             nb = c - '0';
530             hasReadNb = true;
531           }
```

Code source : Représentation de l'échiquier et génération des coups possibles XXII

```
532             else{
533               nb *= 10;
534               nb += c - '0';
535             }
536           }
537         }
538         currentGameState.moveCount = nb;
539       }
540
541       void BoardManager::makeMove(int move){
542         int startPosi = startPos(move);
543         int endPosi = endPos(move);
544         int tag = (move & tagMask) >> 12;
545
546         GameState newGameState;
547         newGameState.canBlackQueenCastle = currentGameState.canBlackQueenCastle;
548         newGameState.canWhiteQueenCastle = currentGameState.canWhiteQueenCastle;
549         newGameState.canWhiteKingCastle = currentGameState.canWhiteKingCastle;
550         newGameState.canBlackKingCastle = currentGameState.canBlackKingCastle;
551         newGameState.doublePushFile = 0;
552         newGameState.whiteKingPos = currentGameState.whiteKingPos;
553         newGameState.blackKingPos = currentGameState.blackKingPos;
554         newGameState.hasWhiteCastled = currentGameState.hasWhiteCastled;
555         newGameState.hasBlackCastled = currentGameState.hasBlackCastled;
556
557         sf::Vector2i startPos = posIntTo2D(startPosi);
558         sf::Vector2i endPos = posIntTo2D(endPosi);
```

Code source : Représentation de l'échiquier et génération des coups possibles XXIII

```
559
560       if (startPosi == endPosi){
561         return;
562       }
563
564       zobristKey ^= piecesZobrist[pieceType(get(startPosi))][whiteToMove][startPosi];
565       if (tag == Capture || tag == KnightPromCapture || tag == RookPromCapture || tag ==
          ↪  BishopPromCapture || tag == QueenPromCapture){
566         zobristKey ^= piecesZobrist[pieceType(get(endPosi))][!whiteToMove][endPosi];
567       }
568       zobristKey ^= piecesZobrist[pieceType(get(startPosi))][whiteToMove][endPosi];
569
570       if (tag == Capture || tag == KnightPromCapture || tag == RookPromCapture || tag ==
          ↪  BishopPromCapture || tag == QueenPromCapture){
571         newGameState.capturedPiece = get(endPosi);
572       }
573       if (tag == QuietMove || tag == Capture){
574         if ( (startPos.x == 0 && pieceType(get(startPosi)) == Rook) || (endPos.x == 0 &&
          ↪  pieceType(get(endPosi)) == Rook)  ){
575           if (whiteToMove){
576             newGameState.canWhiteQueenCastle = false;
577           }
578           else{
579             newGameState.canBlackQueenCastle = false;
580           }
581         }
582         if (  (startPos.x == 7 && pieceType(get(startPosi)) == Rook) || (endPos.x == 7 &&
          ↪  pieceType(get(endPosi)) == Rook)  ){
```

Code source : Représentation de l'échiquier et génération des coups possibles XXIV

```
583         if (whiteToMove){
584           newGameState.canWhiteKingCastle = false;
585         }
586         else{
587           newGameState.canBlackKingCastle = false;
588         }
589       }
590       if (pieceType(get(startPosi)) == King){
591         if (whiteToMove){
592           newGameState.canWhiteQueenCastle = false;
593           newGameState.canWhiteKingCastle = false;
594           newGameState.whiteKingPos = endPosi;
595         }
596         else{
597           newGameState.canBlackQueenCastle = false;
598           newGameState.canBlackKingCastle = false;
599           newGameState.blackKingPos = endPosi;
600         }
601       }
602       board[endPos.y][endPos.x] = board[startPos.y][startPos.x];
603       board[startPos.y][startPos.x] = None;
604     }
605
606     if (tag == KnightPromCapture || tag == RookPromCapture || tag == BishopPromCapture
      ↪  || tag == QueenPromCapture){
607       int piecePromoType = Queen;
608       switch (tag){
```

Code source : Représentation de l'échiquier et génération des coups possibles XXV

```
609         case KnightPromCapture:
610           piecePromoType = Knight;
611           break;
612         case BishopPromCapture:
613           piecePromoType = Bishop;
614           break;
615         case RookPromCapture:
616           piecePromoType = Rook;
617           break;
618         default:
619           piecePromoType = Queen;
620           break;
621       }
622
623
624       if (isPieceWhite(get(startPosi))){
625         board[endPos.y][endPos.x] = White | piecePromoType;
626         zobristKey ^= piecesZobrist[Pawn][1][endPosi];
627         zobristKey ^= piecesZobrist[piecePromoType][1][endPosi];
628       }
629       else{
630         board[endPos.y][endPos.x] = Black | piecePromoType;
631         zobristKey ^= piecesZobrist[Pawn][0][endPosi];
632         zobristKey ^= piecesZobrist[piecePromoType][0][endPosi];
633       }
634       board[startPos.y][startPos.x] = None;
635     }
```

Code source : Représentation de l'échiquier et génération des coups possibles XXVI

```
636
637       if (tag == DoublePawnPush){
638         board[endPos.y][endPos.x] = board[startPos.y][startPos.x];
639         board[startPos.y][startPos.x] = None;
640         newGameState.doublePushFile = startPos.x + 1;
641       }
642       if (tag == EPCapture){
643         board[endPos.y][endPos.x] = board[startPos.y][startPos.x];
644         board[startPos.y][startPos.x] = None;
645         board[startPos.y][endPos.x] = None;
646         zobristKey ^= piecesZobrist[Pawn][!whiteToMove][endPos.x + 8 * startPos.y];
647       }
648       if (tag == KingCastle || tag == QueenCastle){
649         board[endPos.y][endPos.x] = board[startPos.y][startPos.x];
650         board[startPos.y][startPos.x] = None;
651         if (whiteToMove){
652           newGameState.canWhiteKingCastle = false;
653           newGameState.canWhiteQueenCastle = false;
654           newGameState.whiteKingPos = endPosi;
655           newGameState.hasWhiteCastled = true;
656         }
657         else{
658           newGameState.canBlackKingCastle = false;
659           newGameState.canBlackQueenCastle = false;
660           newGameState.blackKingPos = endPosi;
661           newGameState.hasBlackCastled = true;
662         }
```

Code source : Représentation de l'échiquier et génération des coups possibles XXVII

```
663         if (tag == KingCastle){
664           board[startPos.y][5] = board[startPos.y][7];
665           board[startPos.y][7] = None;
666           zobristKey ^= piecesZobrist[Rook][whiteToMove][7 + startPos.y * 8];
667           zobristKey ^= piecesZobrist[Rook][whiteToMove][5 + startPos.y * 8];
668         }
669         else{
670           board[startPos.y][3] = board[startPos.y][0];
671           board[startPos.y][0] = None;
672           zobristKey ^= piecesZobrist[Rook][whiteToMove][startPos.y * 8];
673           zobristKey ^= piecesZobrist[Rook][whiteToMove][3 + startPos.y * 8];
674         }
675       }
676
677       if (tag == KnightProm || tag == RookProm || tag == BishopProm || tag == QueenProm){
678         int piecePromoType = Queen;
679         switch (tag){
680           case KnightProm:
681             piecePromoType = Knight;
682             break;
683           case BishopProm:
684             piecePromoType = Bishop;
685             break;
686           case RookProm:
687             piecePromoType = Rook;
688             break;
689           default:
```

Code source : Représentation de l'échiquier et génération des coups possibles XXVIII

```
690             piecePromoType = Queen;
691             break;
692         }
693
694         if (isPieceWhite(get(startPosi))){
695           board[endPos.y][endPos.x] = White | piecePromoType;
696           zobristKey ^= piecesZobrist[Pawn][1][endPosi];
697           zobristKey ^= piecesZobrist[piecePromoType][1][endPosi];
698         }
699         else{
700           board[endPos.y][endPos.x] = Black | piecePromoType;
701           zobristKey ^= piecesZobrist[Pawn][0][endPosi];
702           zobristKey ^= piecesZobrist[piecePromoType][0][endPosi];
703         }
704         board[startPos.y][startPos.x] = None;
705       }
706
707       if (newGameState.canWhiteKingCastle != currentGameState.canWhiteKingCastle){
708         zobristKey ^= castlingRightZobrist[0];
709       }
710       if (newGameState.canWhiteQueenCastle != currentGameState.canWhiteQueenCastle){
711         zobristKey ^= castlingRightZobrist[1];
712       }
713       if (newGameState.canBlackKingCastle != currentGameState.canBlackKingCastle){
714         zobristKey ^= castlingRightZobrist[2];
715       }
716       if (newGameState.canBlackQueenCastle != currentGameState.canBlackQueenCastle){
```

Code source : Représentation de l'échiquier et génération des coups possibles XXIX

```
717         zobristKey ^= castlingRightZobrist[3];
718       }
719       zobristKey ^= doublePushFileZobrist[currentGameState.doublePushFile];
720       zobristKey ^= doublePushFileZobrist[newGameState.doublePushFile];
721       zobristKey ^= whiteToMoveZobrist;
722
723       newGameState.zobristKey = zobristKey;
724       whiteToMove = !whiteToMove;
725       gameStateHistory.push(currentGameState);
726       currentGameState = newGameState;
727
728     }
729
730     void BoardManager::unmakeMove(int move){
731       int startPosi = startPos(move);
732       int endPosi = endPos(move);
733       int tag = (move & tagMask) >> 12;
734       sf::Vector2i startPos = posIntTo2D(startPosi);
735       sf::Vector2i endPos = posIntTo2D(endPosi);
736
737       if (startPosi == endPosi){
738         return;
739       }
740       if (tag == QuietMove || tag == DoublePawnPush){
741         board[startPos.y][startPos.x] = board[endPos.y][endPos.x];
742         board[endPos.y][endPos.x] = None;
743       }
```

Code source : Représentation de l'échiquier et génération des coups possibles XXX

```
744     if (tag == Capture){
745       board[startPos.y][startPos.x] = board[endPos.y][endPos.x];
746       board[endPos.y][endPos.x] = currentGameState.capturedPiece;
747     }
748     if (tag == EPCapture){
749       board[startPos.y][startPos.x] = board[endPos.y][endPos.x];
750       board[endPos.y][endPos.x] = None;
751       if (!whiteToMove){
752         board[endPos.y + 1][endPos.x] = Black | Pawn;
753       }
754       else{
755         board[endPos.y - 1][endPos.x] = White | Pawn;
756       }
757     }
758     if (tag == KingCastle || tag == QueenCastle){
759       board[startPos.y][startPos.x] = board[endPos.y][endPos.x];
760       board[endPos.y][endPos.x] = None;
761       if (tag == KingCastle){
762         board[startPos.y][7] = board[startPos.y][5];
763         board[startPos.y][5] = None;
764       }
765       else{
766         board[startPos.y][0] = board[startPos.y][3];
767         board[startPos.y][3] = None;
768       }
769     }
770
```

Code source : Représentation de l'échiquier et génération des coups possibles XXXI

```
771     if (tag == KnightProm || tag == RookProm || tag == BishopProm || tag == QueenProm){
772       if (!whiteToMove){
773         board[startPos.y][startPos.x] = White | Pawn;
774       }
775       else{
776         board[startPos.y][startPos.x] = Black | Pawn;
777       }
778       board[endPos.y][endPos.x] = None;
779     }
780 
781     if (tag == KnightPromCapture || tag == RookPromCapture || tag == BishopPromCapture
    ↪  || tag == QueenPromCapture){
782       if (!whiteToMove){
783         board[startPos.y][startPos.x] = White | Pawn;
784       }
785       else{
786         board[startPos.y][startPos.x] = Black | Pawn;
787       }
788       board[endPos.y][endPos.x] = currentGameState.capturedPiece;
789     }
790 
791     whiteToMove = !whiteToMove;
792     currentGameState = gameStateHistory.top();
793     zobristKey = currentGameState.zobristKey;
794     gameStateHistory.pop();
795   }
796 
```

Code source : Représentation de l'échiquier et génération des coups possibles XXXII

```
797   int BoardManager::get(int pos){
798     if (pos >= 0 && pos < 64){
799       return board[pos / 8][pos % 8];
800     }
801     else{
802       return 0;
803     }
804   }
805   int BoardManager::get(int x, int y){
806     if (x >= 0 && x < 8 && y  >= 0 && y < 8){
807       return board[y][x];
808     }
809     else{
810       return 0;
811     }
812   }
813
814   uint64_t BoardManager::computeZobrist(){
815     uint64_t newZobristKey = (uint64_t)0;
816
817     for (int i = 0; i < 64; ++i){
818       int piece = get(i);
819       if (piece > 0){
820         newZobristKey ^= piecesZobrist[pieceType(piece)][isPieceWhite(piece)][i];
821       }
822     }
823     newZobristKey ^= doublePushFileZobrist[currentGameState.doublePushFile];
```

Code source : Représentation de l'échiquier et génération des coups possibles XXXIII

```
824     if (!whiteToMove){
825       newZobristKey ^= whiteToMoveZobrist;
826     }
827
828     if (currentGameState.canWhiteKingCastle){
829       newZobristKey ^= castlingRightZobrist[0];
830     }
831     if (currentGameState.canWhiteQueenCastle){
832       newZobristKey ^= castlingRightZobrist[1];
833     }
834     if (currentGameState.canBlackKingCastle){
835       newZobristKey ^= castlingRightZobrist[2];
836     }
837     if (currentGameState.canBlackQueenCastle){
838       newZobristKey ^= castlingRightZobrist[3];
839     }
840
841     return newZobristKey;
842   }
843
844   void BoardManager::initZobrist(){
845       std::mt19937_64 gen(time(NULL));
846       std::uniform_int_distribution<uint64_t> dis(0,
      ↪  std::numeric_limits<uint64_t>::max());
847
848       for (int pType = 1; pType <= 6; ++pType){
849         for (int color = 0 ; color <= 1; ++color){
```

Code source : Représentation de l'échiquier et génération des coups possibles XXXIV

```
850             for (int squareID = 0; squareID < 64; ++squareID){
851               piecesZobrist[pType][color][squareID] = dis(gen);
852             }
853           }
854         }
855
856         for (int i = 0 ; i < 9; ++i){
857           doublePushFileZobrist[i] = dis(gen);
858         }
859         whiteToMoveZobrist = dis(gen);
860         for (int i = 0; i < 4; ++i){
861           castlingRightZobrist[i] = dis(gen);
862         }
863
864         zobristKey = computeZobrist();
865     }
```

Code source : Trouver le meilleur coup à partir d'une position I

```
1   #include "boardManager.h"
2   #include "transposition.h"
3
4   #define maxBotDepth 50
5
6   #define pawnValue 100
7   #define knightValue 280
8   #define bishopValue 320
9   #define rookValue 479
10  #define queenValue 929
11  #define kingValue 20000
12
13  enum BotType
14  {
15    NotBot = 0,
16    Random = 1,
17    Good = 2
18
19  };
20  struct MoveScore
21  {
22      int move;
23      int score;
24  };
25
26  class Bot
27  {
```

Code source : Trouver le meilleur coup à partir d'une position II

```
28    public:
29      Bot();
30      int getBestMove(BoardManager* board);
31
32      int quietSearch(BoardManager* board, int alpha, int beta);
33      int search(BoardManager* board, char depth, int alpha, int beta);
34      int evaluate(BoardManager* board);
35      int scoreMove(BoardManager* board, int move);
36      std::vector<int> orderMoves(BoardManager* board, std::vector<int> moves, char
        ↪  depth);
37      int accessHeatMapMG(int pType,int i, bool whitePlaying);
38      int accessHeatMapEG(int pType,int i, bool whitePlaying);
39
40      int nbQMoves;
41      int nbMoves = 0;
42      char currentDepth;
43
44      int maxTime;
45      std::chrono::high_resolution_clock::time_point startTime;
46      bool reachedTime;
47
48      TranspositionTable transpositionTable;
49      int nbTranspo = 0;
50      int PVmoves[maxBotDepth];
51
52
53      int mg_pawn_table[64] = {
```

Code source : Trouver le meilleur coup à partir d'une position III

```
54              0,   0,   0,   0,   0,   0,  0,   0,
55             98, 134,  61,  95,  68, 126, 34, -11,
56             -6,   7,  26,  31,  65,  56, 25, -20,
57            -14,  13,   6,  21,  23,  12, 17, -23,
58            -27,  -2,  -5,  12,  17,   6, 10, -25,
59            -26,  -4,  -4, -10,   3,   3, 33, -12,
60            -35,  -1, -20, -23, -15,  24, 38, -22,
61              0,   0,   0,   0,   0,   0,  0,   0,
62      };
63      int eg_pawn_table[64] = {
64              0,   0,   0,   0,   0,   0,   0,   0,
65            178, 173, 158, 134, 147, 132, 165, 187,
66             94, 100,  85,  67,  56,  53,  82,  84,
67             32,  24,  13,   5,  -2,   4,  17,  17,
68             13,   9,  -3,  -7,  -7,  -8,   3,  -1,
69              4,   7,  -6,   1,   0,  -5,  -1,  -8,
70             13,   8,   8,  10,  13,   0,   2,  -7,
71              0,   0,   0,   0,   0,   0,   0,   0,
72      };
73      int mg_knight_table[64] = {
74            -167, -89, -34, -49,  61, -97, -15, -107,
75             -73, -41,  72,  36,  23,  62,   7,  -17,
76             -47,  60,  37,  65,  84, 129,  73,   44,
77              -9,  17,  19,  53,  37,  69,  18,   22,
78             -13,   4,  16,  13,  28,  19,  21,   -8,
79             -23,  -9,  12,  10,  19,  17,  25,  -16,
80             -29, -53, -12,  -3,  -1,  18, -14,  -19,
```

Code source : Trouver le meilleur coup à partir d'une position IV

```
81          -105, -21, -58, -33, -17, -28, -19,  -23,
82      };
83      int eg_knight_table[64] = {
84          -58, -38, -13, -28, -31, -27, -63, -99,
85          -25,  -8, -25,  -2,  -9, -25, -24, -52,
86          -24, -20,  10,   9,  -1,  -9, -19, -41,
87          -17,   3,  22,  22,  22,  11,   8, -18,
88          -18,  -6,  16,  25,  16,  17,   4, -18,
89          -23,  -3,  -1,  15,  10,  -3, -20, -22,
90          -42, -20, -10,  -5,  -2, -20, -23, -44,
91          -29, -51, -23, -15, -22, -18, -50, -64,
92      };
93      int mg_bishop_table[64] = {
94          -29,   4, -82, -37, -25, -42,   7,  -8,
95          -26,  16, -18, -13,  30,  59,  18, -47,
96          -16,  37,  43,  40,  35,  50,  37,  -2,
97           -4,   5,  19,  50,  37,  37,   7,  -2,
98           -6,  13,  13,  26,  34,  12,  10,   4,
99            0,  15,  15,  15,  14,  27,  18,  10,
100           4,  15,  16,   0,   7,  21,  33,   1,
101         -33,  -3, -14, -21, -13, -12, -39, -21,
102     };
103     int eg_bishop_table[64] = {
104         -14, -21, -11,  -8, -7,  -9, -17, -24,
105          -8,  -4,   7, -12, -3, -13,  -4, -14,
106           2,  -8,   0,  -1, -2,   6,   0,   4,
107          -3,   9,  12,   9, 14,  10,   3,   2,
```

Code source : Trouver le meilleur coup à partir d'une position V

```
108          -6,   3,  13,  19,   7,  10,  -3,  -9,
109         -12,  -3,   8,  10,  13,   3,  -7, -15,
110         -14, -18,  -7,  -1,   4,  -9, -15, -27,
111         -23,  -9, -23,  -5,  -9, -16,  -5, -17,
112     };
113     int mg_rook_table[64] = {
114          32,  42,  32,  51, 63,  9,  31,  43,
115          27,  32,  58,  62, 80, 67,  26,  44,
116          -5,  19,  26,  36, 17, 45,  61,  16,
117         -24, -11,   7,  26, 24, 35,  -8, -20,
118         -36, -26, -12,  -1,  9, -7,   6, -23,
119         -45, -25, -16, -17,  3,  0,  -5, -33,
120         -44, -16, -20,  -9, -1, 11,  -6, -71,
121         -19, -13,   1,  17, 16,  7, -37, -26,
122     };
123     int eg_rook_table[64] = {
124         13, 10, 18, 15, 12,  12,   8,   5,
125         11, 13, 13, 11, -3,   3,   8,   3,
126          7,  7,  7,  5,  4,  -3,  -5,  -3,
127          4,  3, 13,  1,  2,   1,  -1,   2,
128          3,  5,  8,  4, -5,  -6,  -8, -11,
129         -4,  0, -5, -1, -7, -12,  -8, -16,
130         -6, -6,  0,  2, -9,  -9, -11,  -3,
131         -9,  2,  3, -1, -5, -13,   4, -20,
132     };
133     int mg_queen_table[64] = {
134         -28,   0,  29,  12,  59,  44,  43,  45,
```

Code source : Trouver le meilleur coup à partir d'une position VI

```
135         -24, -39,  -5,   1, -16,  57,  28,  54,
136         -13, -17,   7,   8,  29,  56,  47,  57,
137         -27, -27, -16, -16,  -1,  17,  -2,   1,
138          -9, -26,  -9, -10,  -2,  -4,   3,  -3,
139         -14,   2, -11,  -2,  -5,   2,  14,   5,
140         -35,  -8,  11,   2,   8,  15,  -3,   1,
141          -1, -18,  -9,  10, -15, -25, -31, -50,
142     };
143     int eg_queen_table[64] = {
144          -9,  22,  22,  27,  27,  19,  10,  20,
145         -17,  20,  32,  41,  58,  25,  30,   0,
146         -20,   6,   9,  49,  47,  35,  19,   9,
147           3,  22,  24,  45,  57,  40,  57,  36,
148         -18,  28,  19,  47,  31,  34,  39,  23,
149         -16, -27,  15,   6,   9,  17,  10,   5,
150         -22, -23, -30, -16, -16, -23, -36, -32,
151         -33, -28, -22, -43,  -5, -32, -20, -41,
152     };
153     int mg_king_table[64] = {
154         -65,  23,  16, -15, -56, -34,   2,  13,
155          29,  -1, -20,  -7,  -8,  -4, -38, -29,
156          -9,  24,   2, -16, -20,   6,  22, -22,
157         -17, -20, -12, -27, -30, -25, -14, -36,
158         -49,  -1, -27, -39, -46, -44, -33, -51,
159         -14, -14, -22, -46, -44, -30, -15, -27,
160           1,   7,  -8, -64, -43, -16,   9,   8,
161         -15,  36,  12, -54,   8, -28,  24,  14,
```

Code source : Trouver le meilleur coup à partir d'une position VII

```
162     };
163     int eg_king_table[64] = {
164         -74, -35, -18, -18, -11,  15,   4, -17,
165         -12,  17,  14,  17,  17,  38,  23,  11,
166          10,  17,  23,  15,  20,  45,  44,  13,
167          -8,  22,  24,  27,  26,  33,  26,   3,
168         -18,  -4,  21,  24,  27,  23,   9, -11,
169         -19,  -3,  11,  21,  23,  16,   7,  -9,
170         -27, -11,   4,  13,  14,   4,  -5, -17,
171         -53, -34, -21, -11, -28, -14, -24, -43
172     };
173
174 };
```

Code source : Trouver le meilleur coup à partir d'une position I

```
1   #include "bot.h"
2   {
3
4   int pieceValue(int ptype){
5     switch (ptype){
6       case Pawn:
7         return pawnValue;
8         break;
9       case Knight:
10        return knightValue;
11        break;
12      case Rook:
13        return rookValue;
14        break;
15      case Bishop:
16        return bishopValue;
17        break;
18      case Queen:
19        return queenValue;
20        break;
21      case King:
22        return kingValue;
23        break;
24    }
25    return 0;
26  }
27
```

Code source : Trouver le meilleur coup à partir d'une position II

```
28  int Bot::scoreMove(BoardManager* board, int move){
29    int score = 0;
30
31    int movingPieceType = pieceType(board->get(startPos(move)));
32    int capturedPieceType = pieceType(board->get(endPos(move)));
33    int moveTag = tag(move);
34
35    if (capturedPieceType > 0){
36      score += 10 * pieceValue(capturedPieceType) - pieceValue(movingPieceType);
37    }
38
39    if (moveTag >= 8) // Promotion{
40      int promPieceType = Queen;
41      switch (moveTag){
42        case KnightProm : case KnightPromCapture:
43          promPieceType = Knight;
44          break;
45        case BishopProm : case BishopPromCapture:
46          promPieceType = Bishop;
47          break;
48        case RookProm: case RookPromCapture:
49          promPieceType = Rook;
50          break;
51      }
52      score += pieceValue(promPieceType);
53    }
54    board->whiteToMove = !board->whiteToMove;
```

Code source : Trouver le meilleur coup à partir d'une position III

```
55      board->controlledSquares();
56      if (board->controlled[endPos(move) / 8][endPos(move) % 8]){
57        score -= pieceValue(movingPieceType);
58      }
59      board->whiteToMove = !board->whiteToMove;
60      return score;
61    }
62
63    bool compMove(MoveScore mscore1, MoveScore mscore2){
64      return mscore1.score > mscore2.score;
65    }
66
67    std::vector<int> Bot::orderMoves(BoardManager* board, std::vector<int> moves, char
      ↪  depth){
68      std::vector<MoveScore> movesScore;
69      for (int move : moves){
70        MoveScore mscore;
71        mscore.move = move;
72
73        if (move == PVmoves[currentDepth - depth]){
74          mscore.score = infinity;
75        }
76        else{
77          mscore.score = scoreMove(board, move);
78        }
79
80        movesScore.push_back(mscore);
```

Code source : Trouver le meilleur coup à partir d'une position IV

```
81      }
82
83      std::sort(movesScore.begin(), movesScore.end(), compMove);
84      std::vector<int> sortedMoves;
85
86      for(MoveScore mscore : movesScore){
87        sortedMoves.push_back(mscore.move);
88      }
89      return sortedMoves;
90    }
91
92
93
94    int rotate(int i, bool whitePlaying){
95      return whitePlaying ? i : (7 - (i%8)) + (i/8) * 7;
96    }
97
98    int Bot::accessHeatMapMG(int pType,int i, bool whitePlaying){
99      switch(pType){
100       case Pawn:
101         return mg_pawn_table[rotate(i, whitePlaying)];
102         break;
103       case Knight:
104         return mg_knight_table[rotate(i, whitePlaying)];
105         break;
106       case Bishop:
107         return mg_bishop_table[rotate(i, whitePlaying)];
```

Code source : Trouver le meilleur coup à partir d'une position V

```
108         break;
109       case Rook:
110         return mg_rook_table[rotate(i, whitePlaying)];
111         break;
112       case Queen:
113         return mg_queen_table[rotate(i, whitePlaying)];
114         break;
115       case King:
116         return mg_king_table[rotate(i, whitePlaying)];
117         break;
118     }
119     return 0;
120   }
121
122   int Bot::accessHeatMapEG(int pType,int i, bool whitePlaying){
123     switch(pType){
124       case Pawn:
125         return eg_pawn_table[rotate(i, whitePlaying)];
126         break;
127       case Knight:
128         return eg_knight_table[rotate(i, whitePlaying)];
129         break;
130       case Bishop:
131         return eg_bishop_table[rotate(i, whitePlaying)];
132         break;
133       case Rook:
134         return eg_rook_table[rotate(i, whitePlaying)];
```

Code source : Trouver le meilleur coup à partir d'une position VI

```
135         break;
136       case Queen:
137         return eg_queen_table[rotate(i, whitePlaying)];
138         break;
139       case King:
140         return eg_king_table[rotate(i, whitePlaying)];
141         break;
142     }
143     return 0;
144   }
145
146
147   int restrainKingEndGame(BoardManager* board, int myKingPos, int opponentKingPos){
148     int newScore = 0;
149     int distCenterOpp = abs((opponentKingPos % 8) - 3) + abs((opponentKingPos / 8) -
      ↪  3);
150     int distCenterFriend = abs((myKingPos % 8) - 3) + abs((myKingPos / 8) - 3);
151     int distKings = abs((myKingPos % 8) - (opponentKingPos % 8)) + abs((myKingPos / 8)
      ↪  - (opponentKingPos / 8));
152
153     newScore += distCenterOpp - distCenterFriend;
154     newScore += 2 * (14 - distKings);
155     return newScore;
156
157   }
158
159   int Bot::evaluate(BoardManager* board){
```

Code source : Trouver le meilleur coup à partir d'une position VII

```
160     int score = 0;
161     int whiteScoreValue = 0;
162     int blackScoreValue = 0;
163     int heatMapScoreMGWhite = 0;
164     int heatMapScoreMGBlack = 0;
165     int heatMapScoreEGWhite = 0;
166     int heatMapScoreEGBlack = 0;
167     int pieceNumber = 0;
168
169     for (int i = 0 ; i < 64 ; ++i){
170       int piece = board->get(i);
171       int pType = pieceType(piece);
172       if (pType != None){
173         pieceNumber += 1;
174       }
175       if (isPieceWhite(piece)){
176         whiteScoreValue += pieceValue(pType);
177         heatMapScoreMGWhite += accessHeatMapMG(pType,i, board->whiteToMove);
178         heatMapScoreEGWhite += accessHeatMapEG(pType,i, board->whiteToMove);
179       }
180       else{
181         blackScoreValue += pieceValue(pType);
182         heatMapScoreMGBlack += accessHeatMapMG(pType,i, board->whiteToMove);
183         heatMapScoreEGBlack += accessHeatMapEG(pType,i, board->whiteToMove);
184       }
185
186     }
```

Code source : Trouver le meilleur coup à partir d'une position VIII

```
187
188     float endGameWeight = 1.0 - (float(pieceNumber) / 32.0);
189
190     score += int((1.0 - endGameWeight) * ( (board->currentGameState.hasWhiteCastled) -
        ↪  (board->currentGameState.hasBlackCastled)) * (board->whiteToMove ? 1 : -1) *
        ↪  50);
191     score += int( 0.5 * ( (1.0 - endGameWeight) * (heatMapScoreMGWhite -
        ↪  heatMapScoreMGBlack) + endGameWeight * (heatMapScoreEGWhite -
        ↪  heatMapScoreEGBlack) )* (board->whiteToMove ? 1 : -1));
192     score -= board->isChecked() * 50;
193
194     if (board->whiteToMove){
195       score += int(restrainKingEndGame(board, board->currentGameState.whiteKingPos,
          ↪  board->currentGameState.blackKingPos) * 10 * endGameWeight);
196     }
197     else{
198       score += int(restrainKingEndGame(board, board->currentGameState.blackKingPos,
          ↪  board->currentGameState.whiteKingPos) * 10 * endGameWeight);
199     }
200
201     score += (whiteScoreValue - blackScoreValue) * (board->whiteToMove ? 1 : -1);
202     return score;
203   }
204
205   int Bot::quietSearch(BoardManager* board, int alpha, int beta){
206     int eval = evaluate(board);
207     if (eval >= beta){
```

Code source : Trouver le meilleur coup à partir d'une position IX

```
208         return beta;
209       }
210       if (eval > alpha){
211         alpha = eval;
212       }
213
214       std::vector<int> moves = board->generateMoves(true);
215       std::vector<int> sortedMoves = orderMoves(board, moves, 0);
216
217       for (int move : sortedMoves){
218         board->makeMove(move);
219         int eval = -quietSearch(board, -beta, -alpha);
220         board->unmakeMove(move);
221
222         nbQMoves += 1;
223
224         if (eval >= beta){
225           return beta;
226         }
227         if (eval > alpha){
228           alpha = eval;
229         }
230       }
231       return alpha;
232
233     }
234
```

Code source : Trouver le meilleur coup à partir d'une position X

```
235   int Bot::search(BoardManager* board, char depth, int alpha, int beta){
236     char nodeType = AlphaNode;
237     Transposition t = transpositionTable.get(board->zobristKey, depth, alpha, beta);
238     if (t.isValid){
239       nbTranspo += 1;
240       return t.value;
241     }
242
243     std::chrono::high_resolution_clock::time_point endTime =
        ↪  std::chrono::high_resolution_clock::now();
244     std::chrono::milliseconds duration =
        ↪  std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
245
246     if (depth == 0){
247       nbMoves += 1;
248       return quietSearch(board, alpha, beta);
249     }
250     if (duration.count() > maxTime){
251       reachedTime = true;
252       return 0;
253     }
254
255     std::vector<int> moves = board->generateMoves(false);
256     std::vector<int> sortedMoves = orderMoves(board, moves, depth);
257     if (sortedMoves.size() == 0){
258       if (board->isChecked()){
259         return -3*kingValue;
```

Code source : Trouver le meilleur coup à partir d'une position XI

```
260         }
261         return 0;
262       }
263
264       int bestPositionMove = 0;
265
266       for (int move : sortedMoves){
267
268         board->makeMove(move);
269         int eval = -search(board, depth - 1, -beta, -alpha);
270         board->unmakeMove(move);
271
272
273         if (eval >= beta){
274           transpositionTable.set(board->zobristKey, depth, beta, BetaNode, move);
275           return beta;
276         }
277         if (eval > alpha){
278           nodeType = ExactNode;
279           bestPositionMove = move;
280           alpha = eval;
281         }
282
283       }
284       transpositionTable.set(board->zobristKey, depth, alpha, nodeType,
          ↪  bestPositionMove);
285       return alpha;
```

Code source : Trouver le meilleur coup à partir d'une position XII

```
286
287     }
288
289
290
291     int Bot::getBestMove(BoardManager* board){
292       nbMoves = 0;
293       nbTranspo = 0;
294       reachedTime = false;
295       nbQMoves = 0;
296       startTime = std::chrono::high_resolution_clock::now();
297       int eval = 0;
298       int finalEval = 0;
299
300       for (char i = 1; i <= maxBotDepth; ++i){
301         currentDepth = i;
302         eval = search(board, i, -infinity, infinity);
303         if (!reachedTime){
304           finalEval = eval;
305           std::stack<int> moves;
306           for (int j = 0; j < 1; ++j){
307             Transposition t = transpositionTable.get(board->zobristKey, 0, 0, 0);
308             if (t.isValid){
309               PVmoves[j] = t.bestMove;
310               board->makeMove(t.bestMove);
311               moves.push(t.bestMove);
312             }
```

Code source : Trouver le meilleur coup à partir d'une position XIII

```
313             else{
314               break;
315             }
316           }
317           while (!moves.empty()){
318             board->unmakeMove(moves.top());
319             moves.pop();
320           }
321         }
322         else{
323           break;
324         }
325
326       }
327
328       printf("info Profondeur : %d\n", currentDepth - 1);
329       printf("info Nombre positions evaluees : %d\n", nbMoves);
330       printf("info Nombre transpositions rencontrees : %d\n", nbTranspo);
331       printf("info Nombre d'entrees dans la table de transposition : %d\n",
          ↪  transpositionTable.count);
332       printf("info Nombre de positions silencieuses evaluees : %d\n", nbQMoves);
333       printf("info Mouvement --> %s  : eval =  %d\n",
          ↪  standardNotation(PVmoves[0]).c_str(), finalEval);
334       printf("info ----------\n");
335
336       transpositionTable.clear();
```

Code source : Trouver le meilleur coup à partir d'une position XIV

```
337         return PVmoves[0];
338     }
```

Code source : Gérer les transpositions I

```
1   #define tableSize 35000
2
3   enum NodeType{
4     ExactNode = 0,
5     AlphaNode = 1,
6     BetaNode = 2
7   };
8   struct Transposition{
9     uint64_t key;
10    char depth;
11    int value;
12    int bestMove;
13    char nodeType;
14    bool isValid = false;
15  };
16
17  class TranspositionTable{
18  public:
19    TranspositionTable();
20    Transposition get(uint64_t key, char depth, int alpha, int beta);
21    void set(uint64_t key, char depth, int value, char nodeType, int bestMove);
22    void clear();
23
24    int count;
25    Transposition table[tableSize];
26
27  };
```

Code source : Gérer les transpositions I

```
1   #include "transposition.h"
2
3   TranspositionTable::TranspositionTable(){
4     clear();
5   }
6
7   Transposition TranspositionTable::get(uint64_t key, char depth, int alpha, int beta){
8     int index = key % tableSize;
9     Transposition t = table[index];
10    if (t.key == key){
11      if (t.depth >= depth){
12        if (t.nodeType == ExactNode){
13          return t;
14        }
15        if (t.nodeType == AlphaNode && t.value <= alpha){
16          return t;
17        }
18        if (t.nodeType == BetaNode && t.value >= beta){
19          return t;
20        }
21      }
22    }
23    t.isValid = false;
24    return t;
25  }
26
27  void TranspositionTable::set(uint64_t key, char depth, int value, char nodeType, int
    ↪  bestMove){
28    Transposition t;
29    t.isValid = true;
```

Code source : Gérer les transpositions II

```
30      t.key = key;
31      t.depth = depth;
32      t.value = value;
33      t.nodeType = nodeType;
34      t.bestMove = bestMove;
35      table[key % tableSize] = t;
36      count += 1;
37
38    }
39
40    void TranspositionTable::clear(){
41      for (int i = 0 ; i < tableSize; ++i){
42        table[i].isValid = false;
43      }
44      count = 0;
45    }
```

- Dann Corbit and Swaminathan Natarajan, *Strategig test suite*, https://sites.google.com/site/strategictestsuite/about, 2010.
- Omid David-Tabibi and Nathan S. Netanyahu, *Verified null-move pruning*, https://arxiv.org/pdf/0808.1125.pdf, 2008.
- Ronald Friederich, *Pesto's evaluation function*, https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function, 2021.
- Tomasz Michniewski, *Simplified evaluation function*, https://www.chessprogramming.org/Simplified_Evaluation_Function, 2021.
- CLAUDE E. SHANNON, *Programming a computer for playing chess*, https://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf, 1949.


Albert L. Zobrist, *A new hashing method with applicaion for game playing*, `https://research.cs.wisc.edu/techreports/1970/TR88.pdf`, 1970.