Étude d'une intelligence artificielle au jeu de dames

DIEME Steve

n° 24879

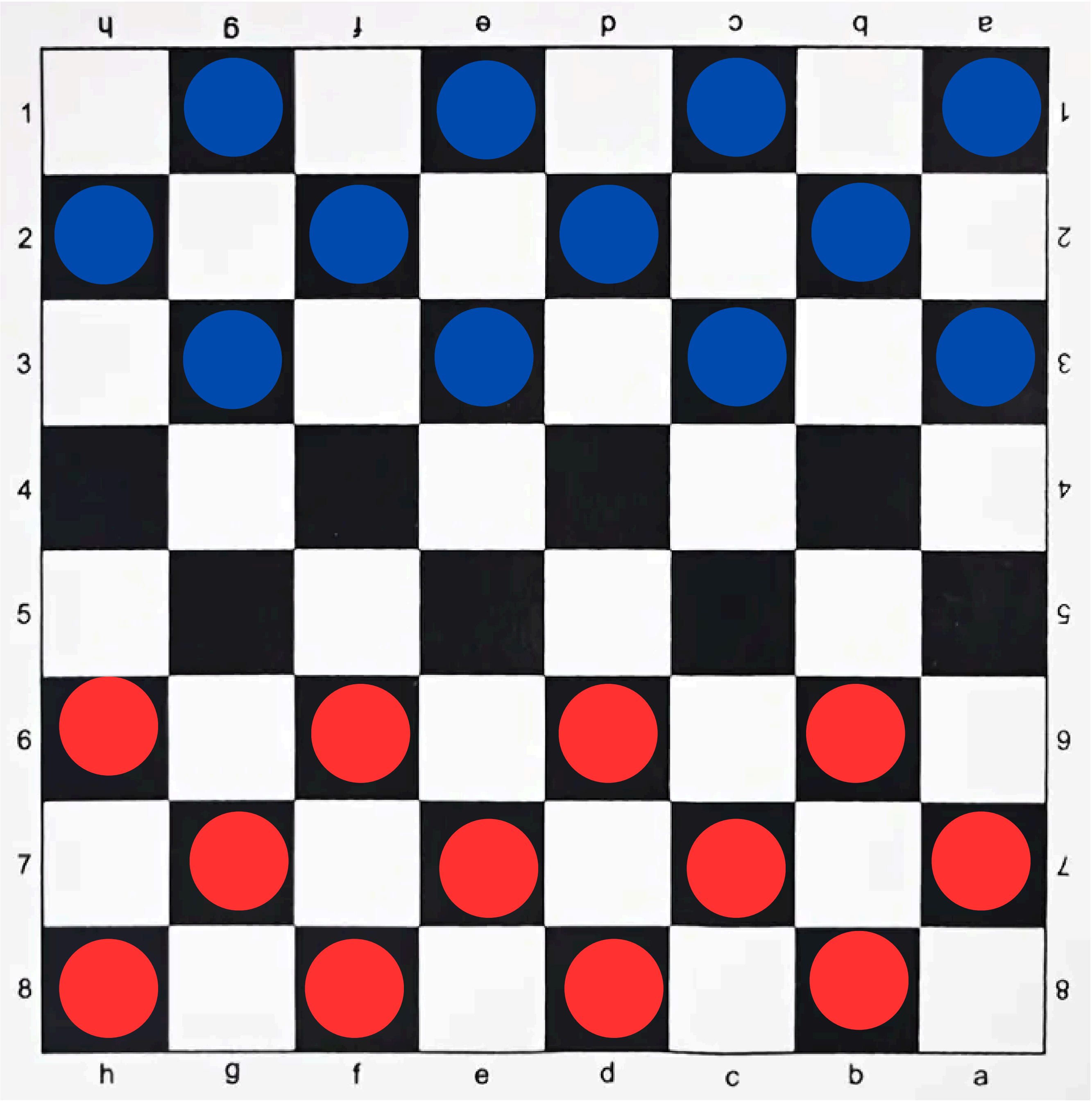
Sommaire

COMBIEN DE COUPS À L'AVANCE UNE INTELLIGENCE ARTIFICIELLE DOIT-ELLE PRÉVOIR POUR ÊTRE CAPABLE DE VAINCRE UN ÊTRE HUMAIN ?

1 CONTEXTUALISATION

Intelligence Artificielle (IA) :

Humain(HU) :



RÈGLE SPÉCIFIQUES/ HYPOTHÈSE:

- *Utilisation de la version Polonaise (en 8x8) :*

  Obligation de prendre le maximum de pions adverse lorsque cela est possible

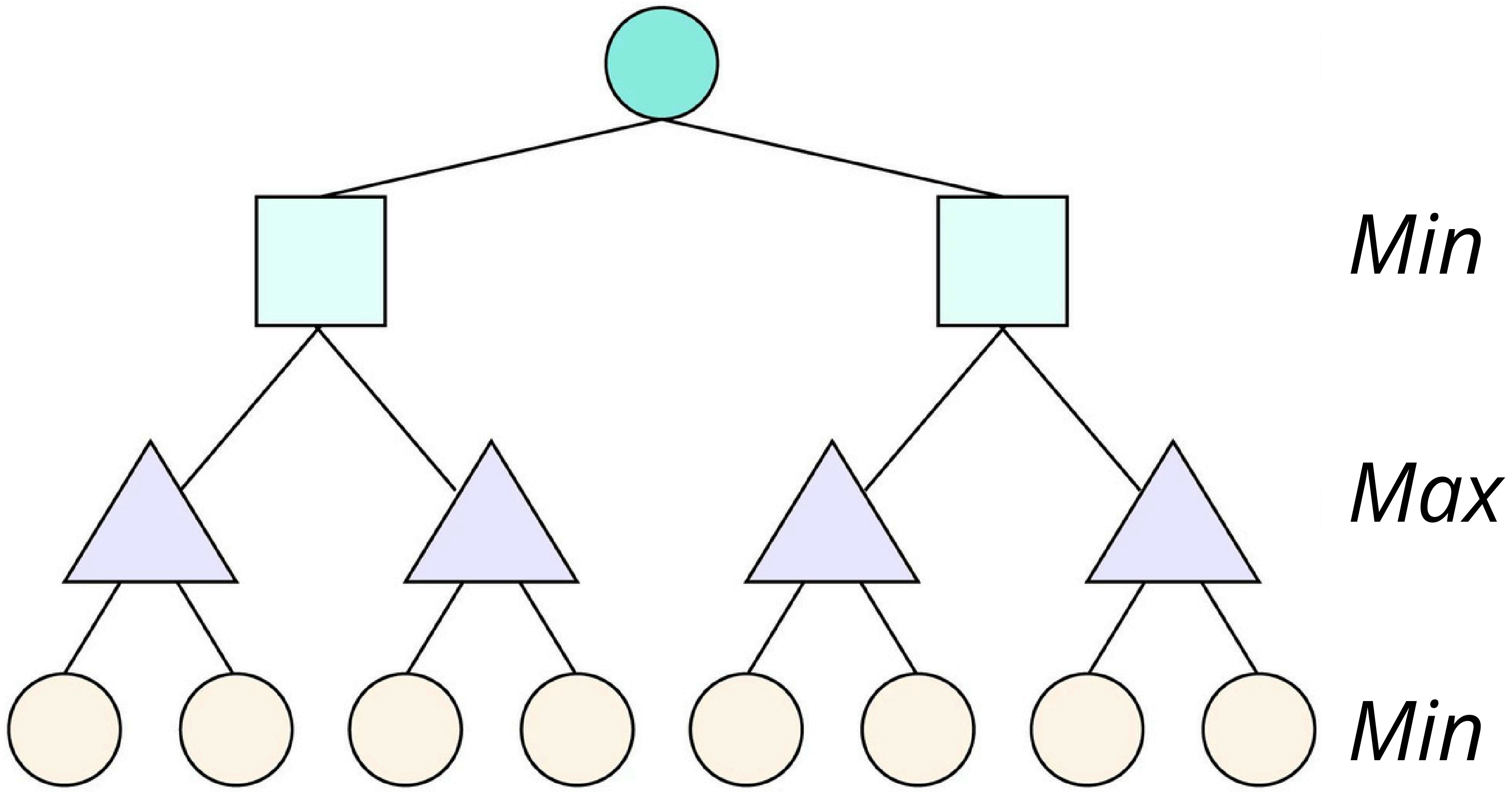
  Déplacement possible uniquement vers l'avant et prise possible dans les deux sens

- *Les dames se déplacent d'autant de cases qu'elles le souhaitent dans n'importe quel sens*

- *Partie terminée lorsque les pièces de son adversaire sont capturées ou immobilisées.*

Algorithme <u>MINMAX</u> :

Pour les jeux à deux joueur consistant à minimiser la perte maximum



<u>Fonction Score:</u>

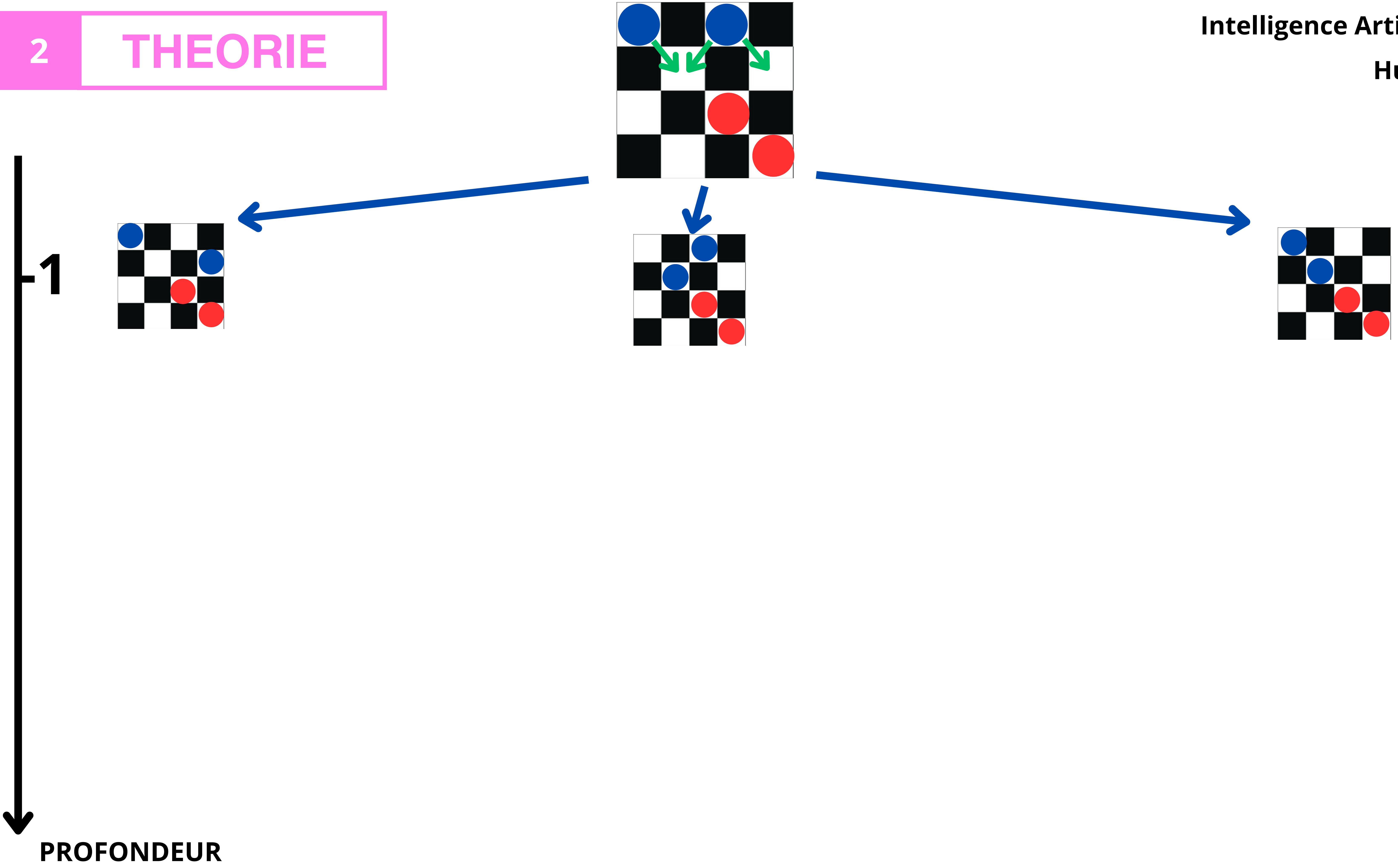
$$\textbf{Score} = \sum_{\text{Pion IA}} - \sum_{\text{Pion Humain}}$$

On suppose que le joueur Humain cherche à minimiser le score tandis que L'IA cherche à le maximiser

2 THEORIE

Intelligence Artificielle (IA) :

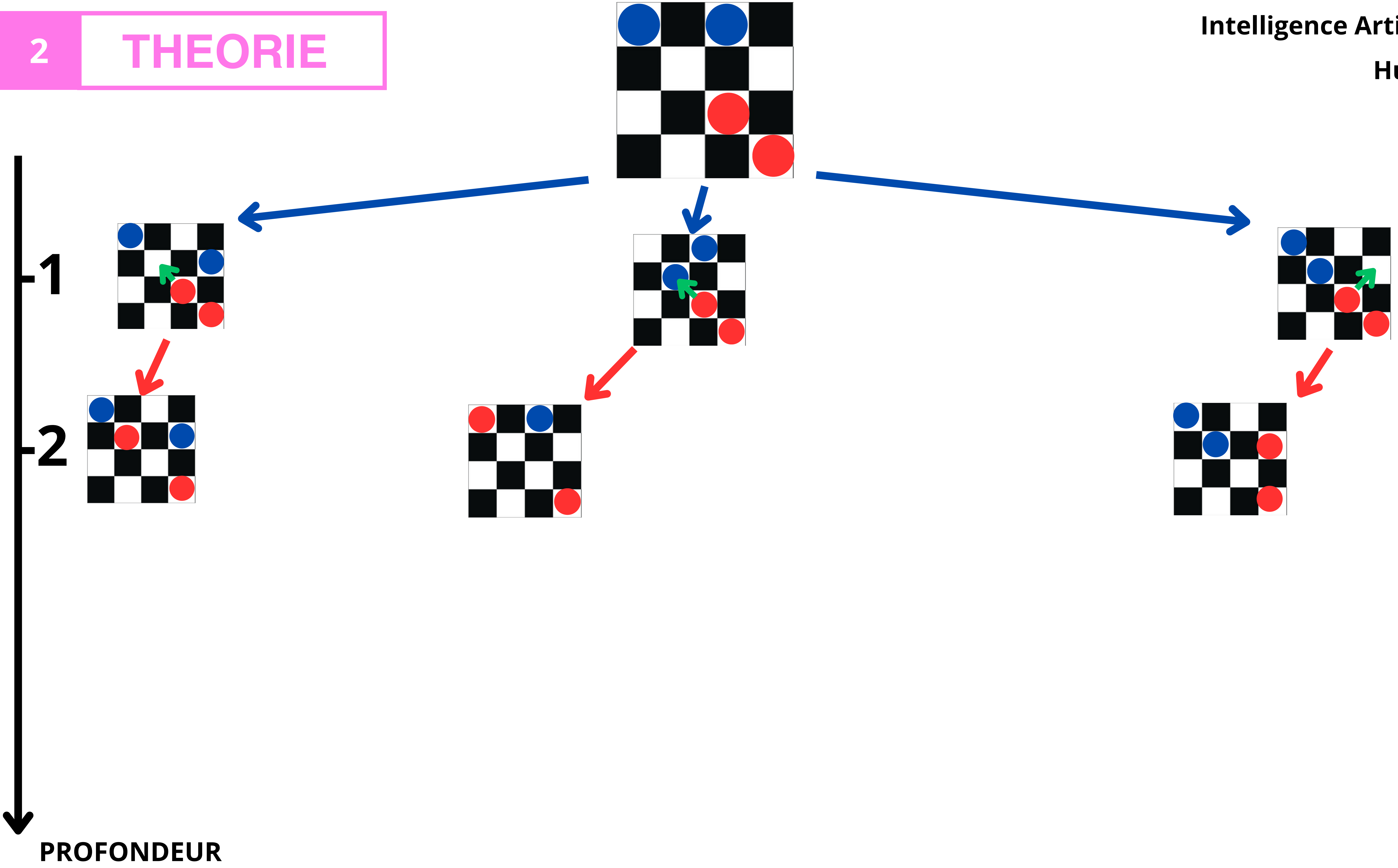
Humain(HU) :



2 THEORIE

Intelligence Artificielle (IA) :

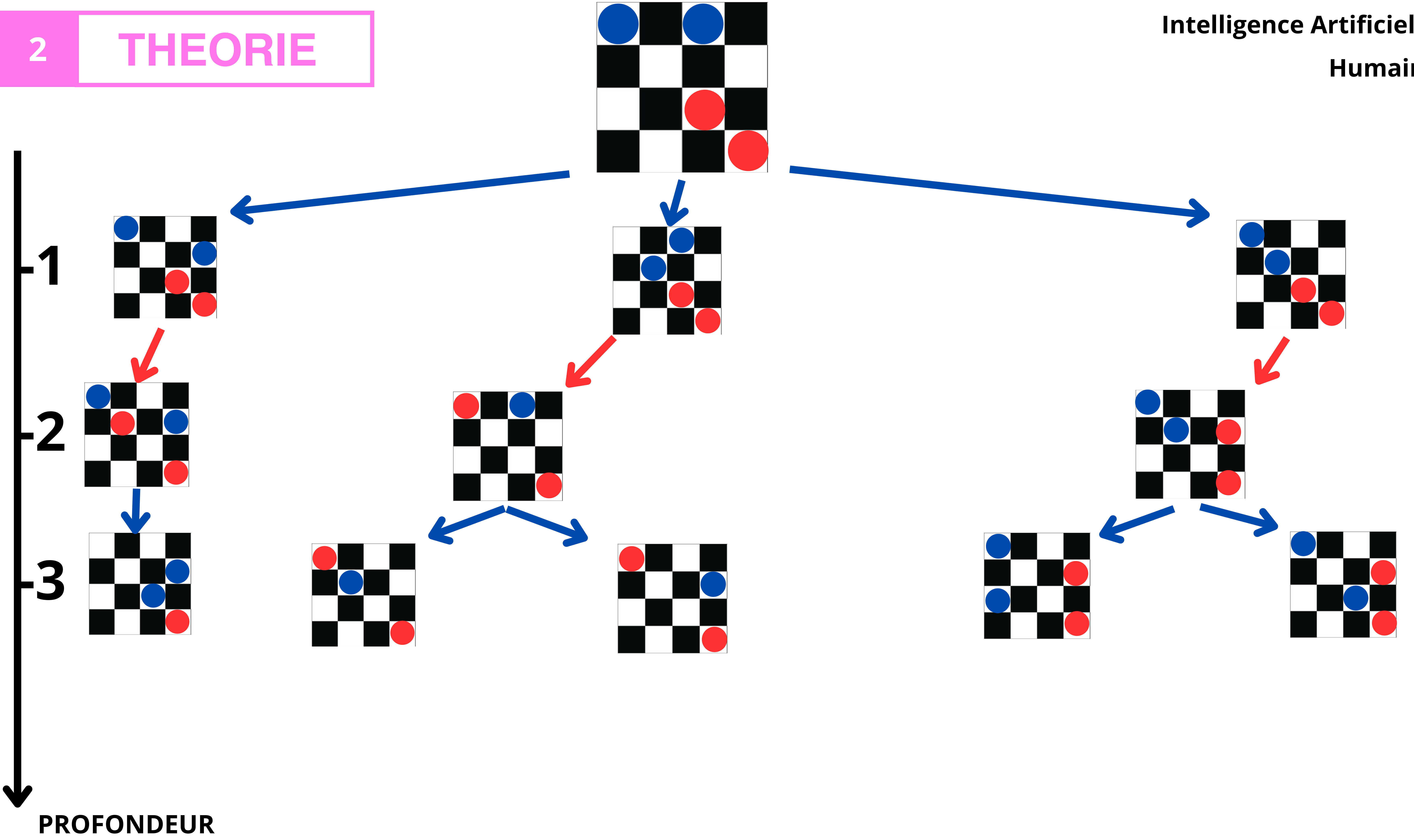
Humain(HU) :



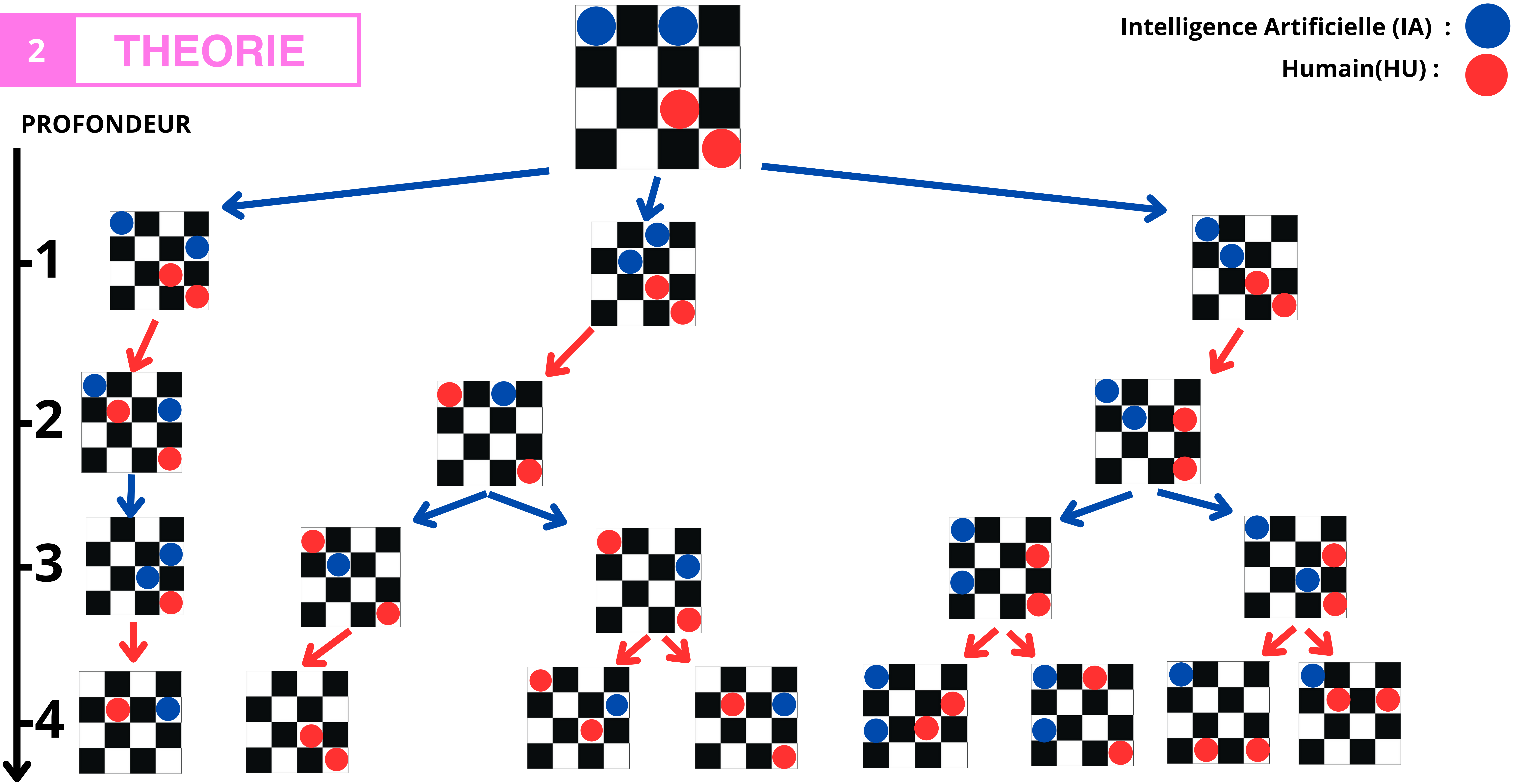
2 THEORIE

Intelligence Artificielle (IA) :

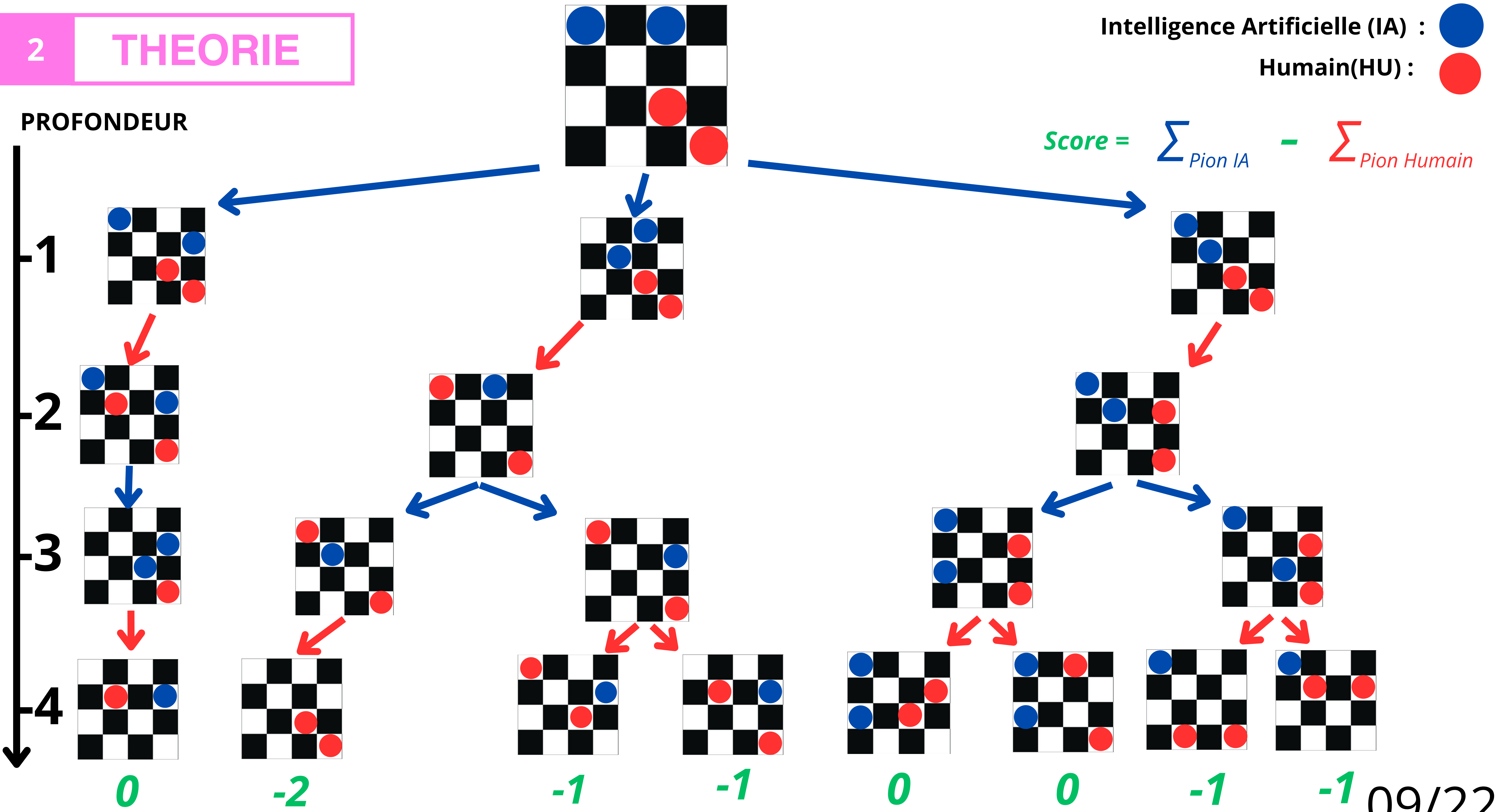
Humain(HU) :



2 THEORIE

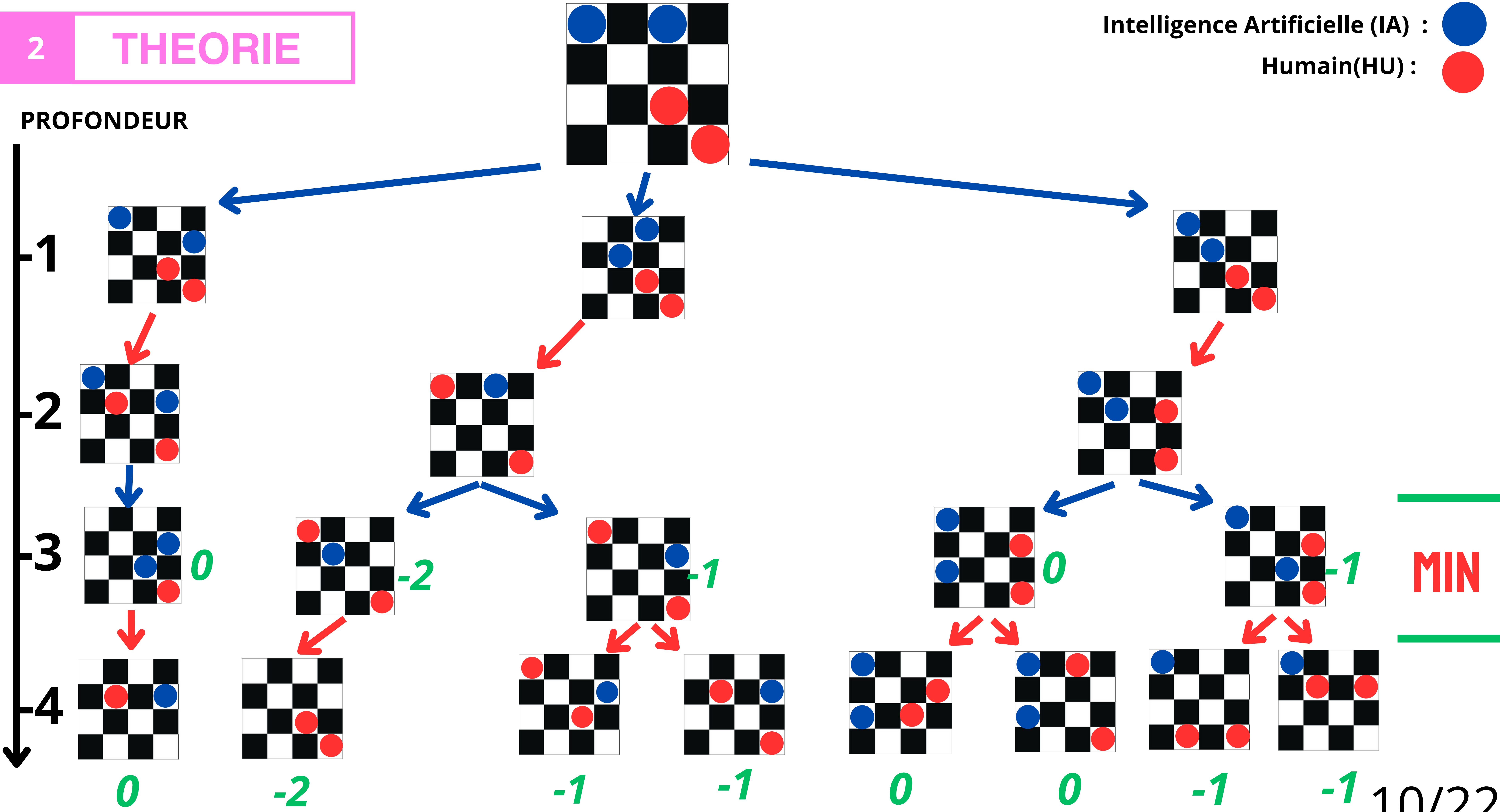


2 THEORIE



Intelligence Artificielle (IA) :
Humain(HU) :
Score = $\sum_{\text{Pion IA}} - \sum_{\text{Pion Humain}}$
PROFONDEUR
1
2
3
4
0
-2
-1
-1
0
0
-1
-1

2 THEORIE

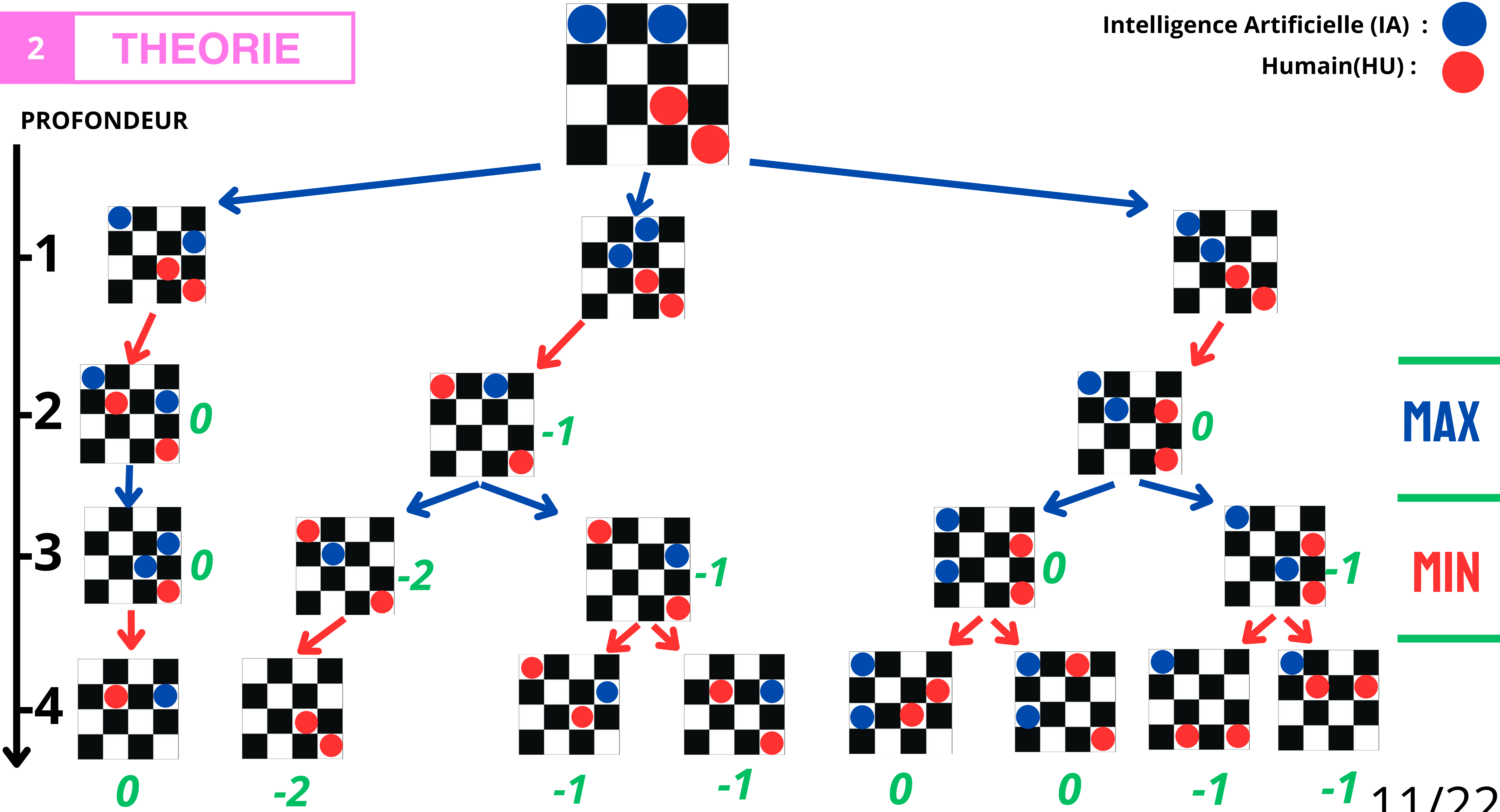


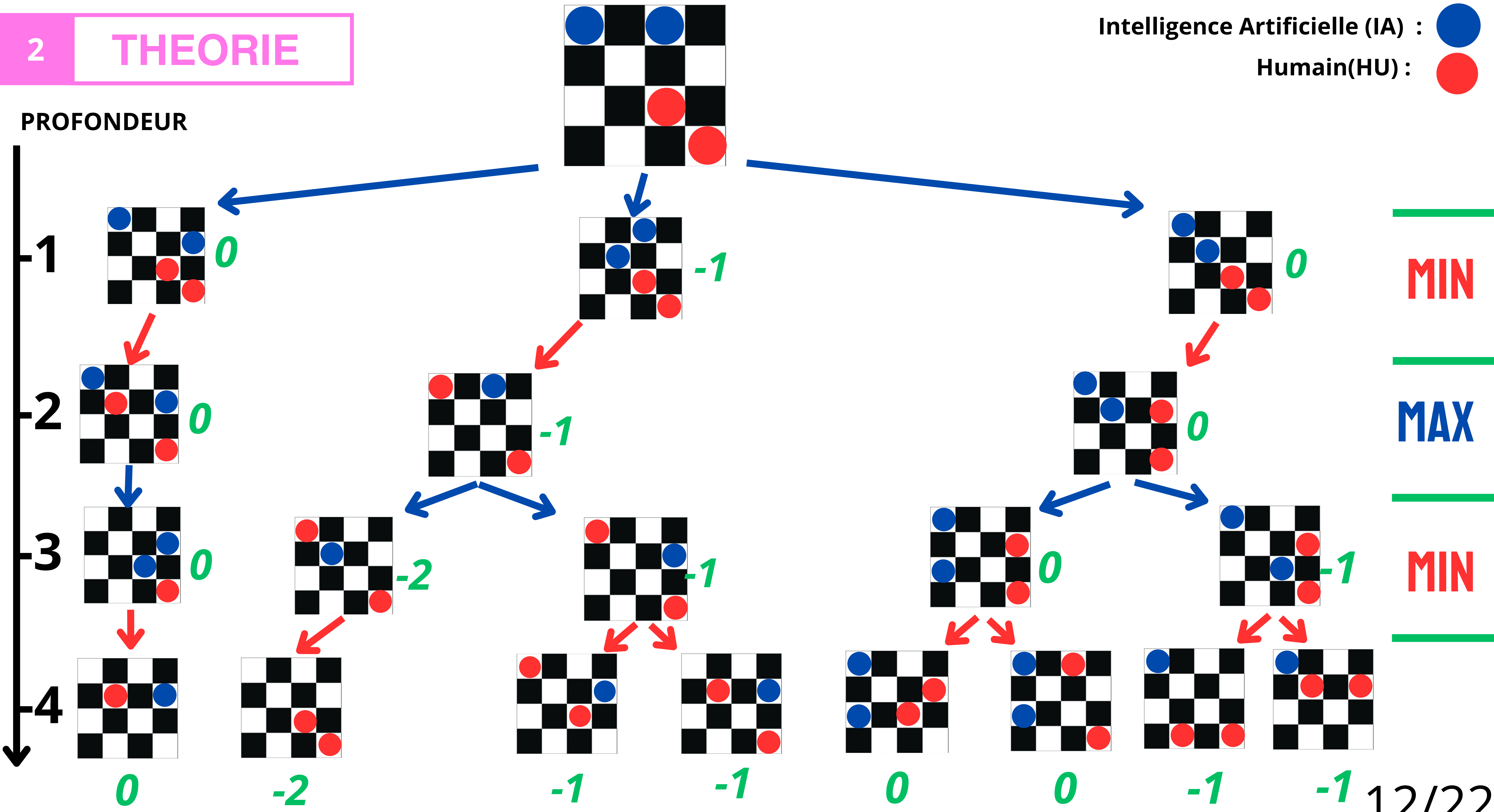
Intelligence Artificielle (IA) :
Humain(HU) :
PROFONDEUR
1
2
3
4
0
-2
-1
0
-1
MIN
0
-2
-1
-1
0
0
-1
-1

2 THEORIE

Intelligence Artificielle (IA) :

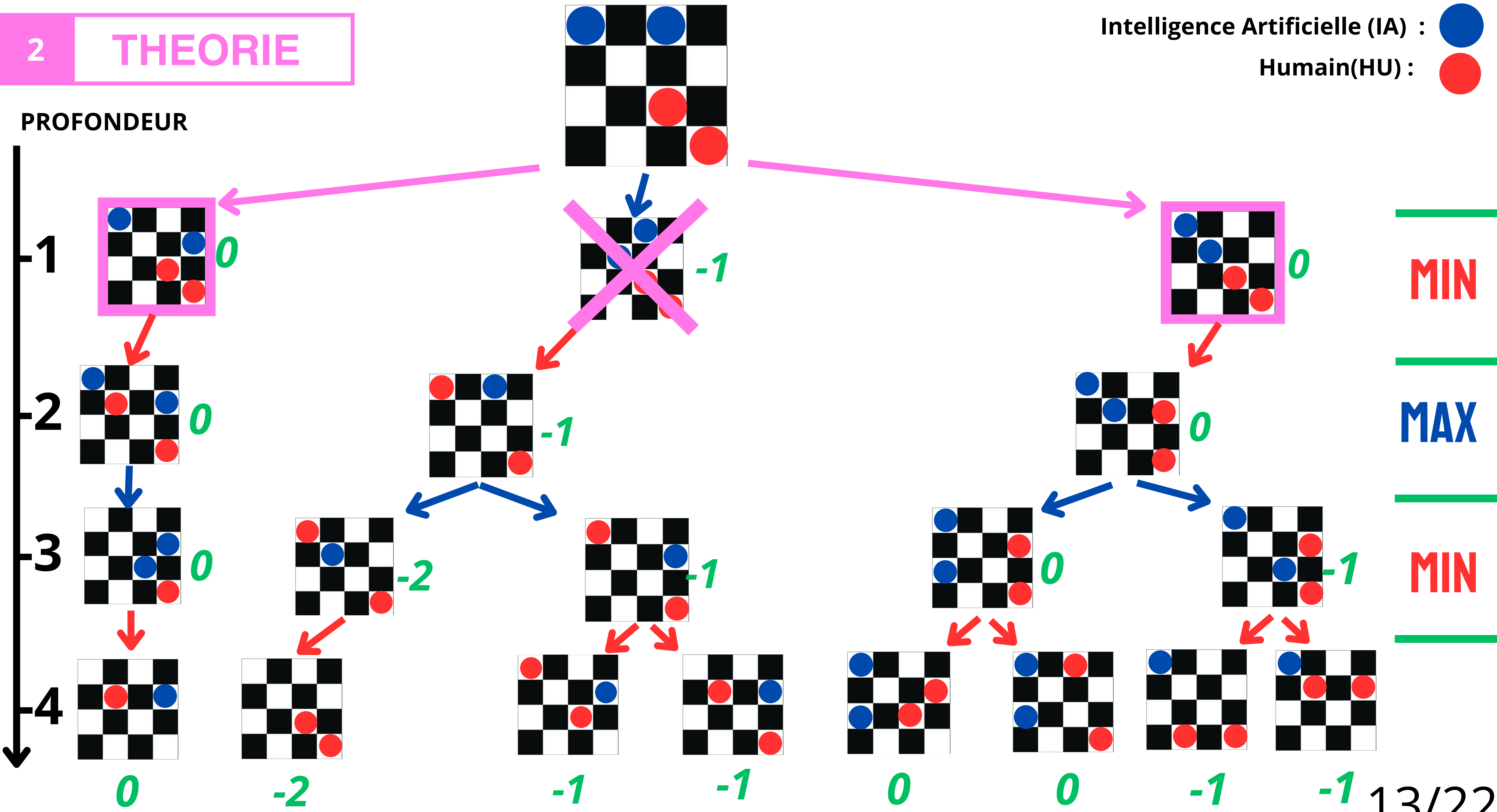
Humain(HU) :





Intelligence Artificielle (IA) :
Humain(HU) :
PROFONDEUR
1
0
-1
0
MIN
2
0
-1
0
MAX
3
0
-2
-1
0
-1
MIN
4
0
-2
-1
-1
0
0
-1
-1

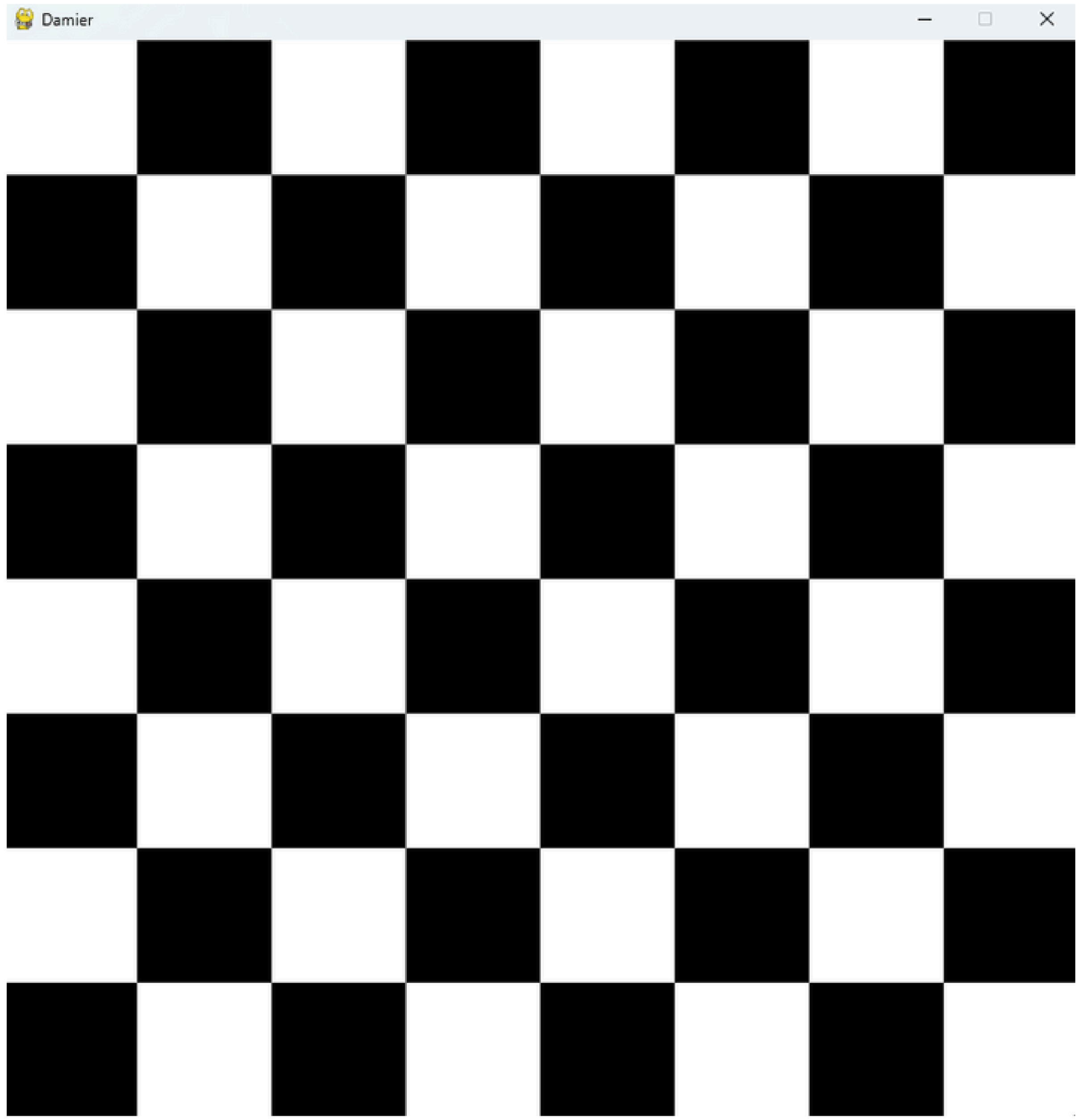
2 THEORIE



Intelligence Artificielle (IA) :
Humain(HU) :
PROFONDEUR
1
2
3
4
0
-1
0
0
-1
0
0
-2
-1
0
-1
0
-2
-1
-1
0
0
-1
-1
MIN
MAX
MIN

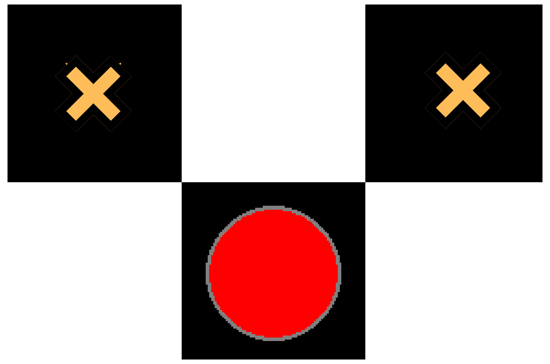
Interface

- *Utilisation de la bibliothèque sur python*
- Dessin du damier 8x8, puis ajout des pièces sur celui-ci
- Ajout des fonctionnalités essentielles au jeu de dames

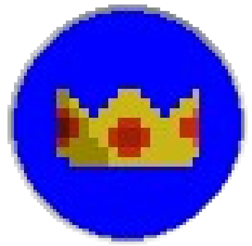


Ajout des mouvements et fonctionnalitées diverses au damier

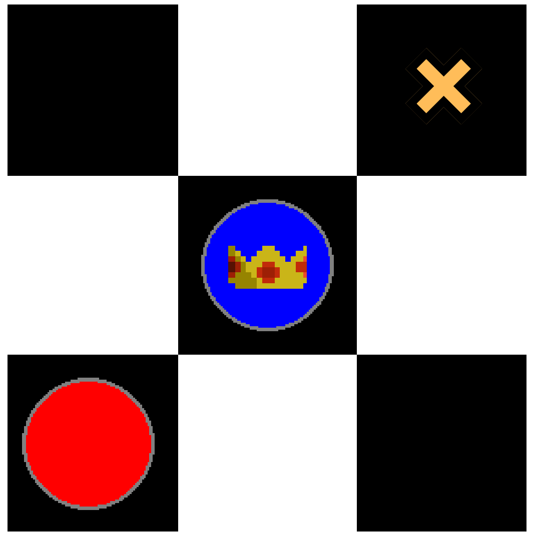
- *Mouvement possible seulement vers l'avant et affichage des possibilités de déplacement :*

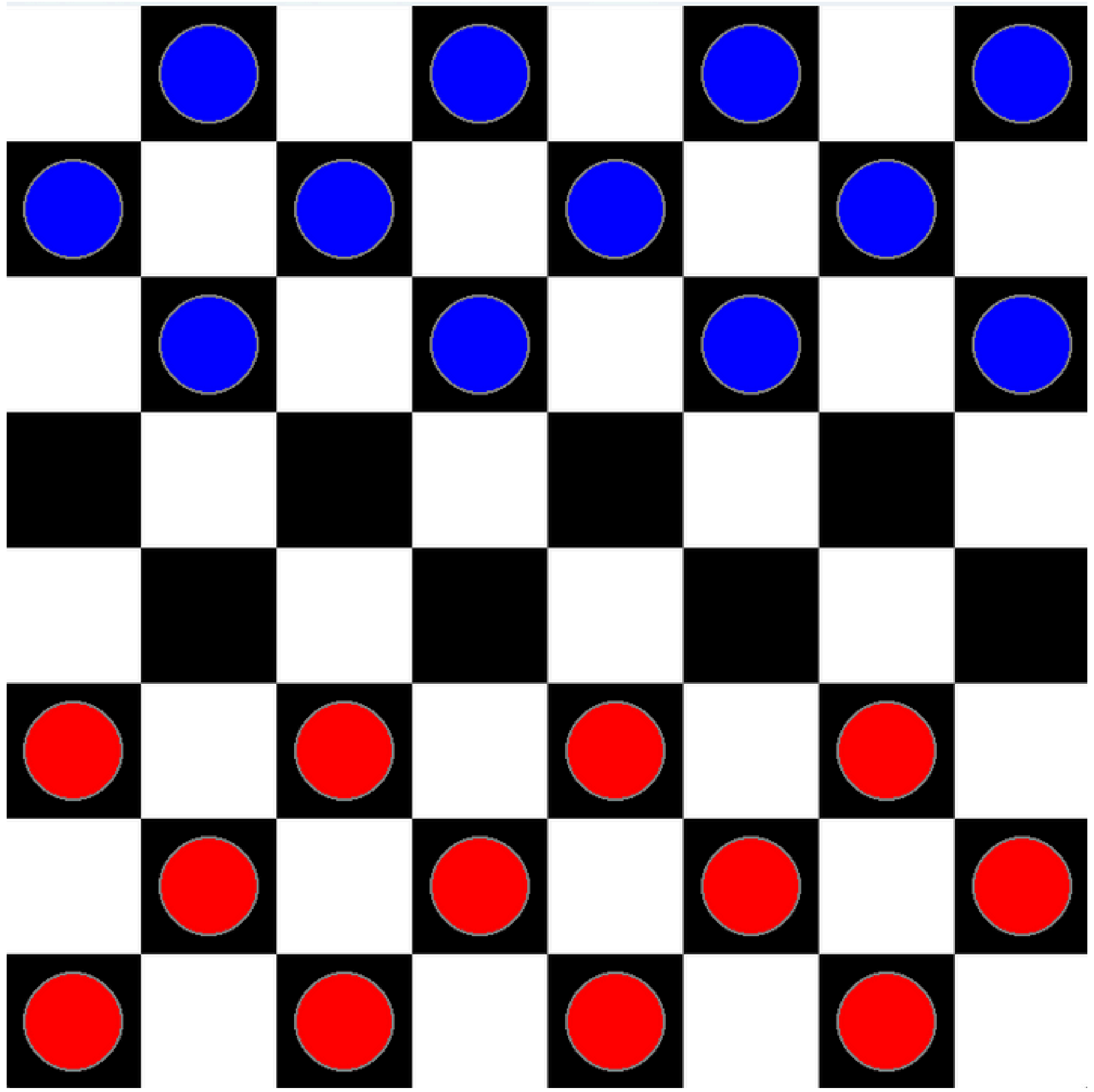


- Pieces différenciées par une couronne pour une dame :

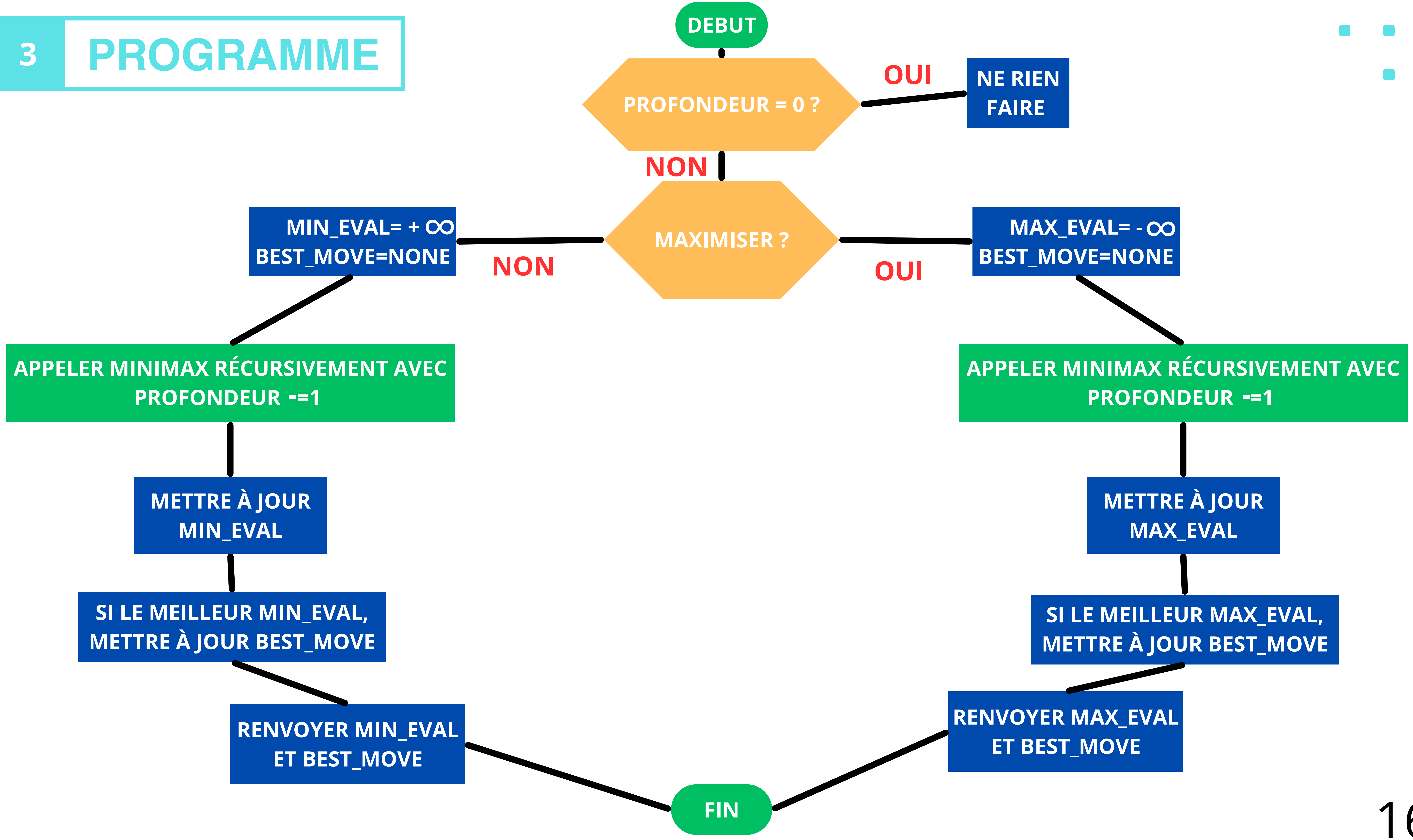


- Mise en place de l'action de prise :





3 PROGRAMME



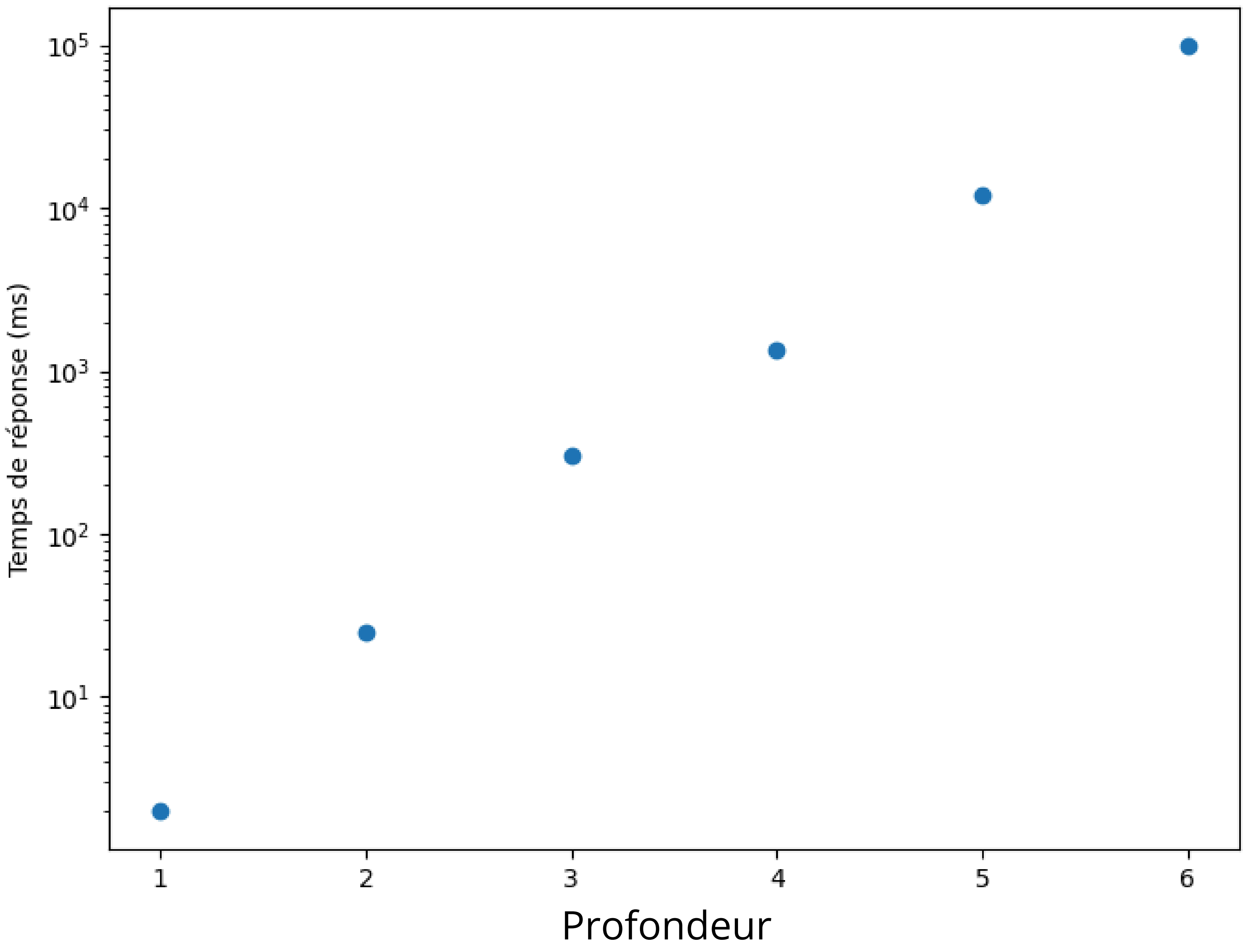
DEBUT
PROFONDEUR = 0 ?
OUI
NE RIEN FAIRE
NON
MAXIMISER ?
NON
MIN_EVAL= + ∞
BEST_MOVE=NONE
OUI
MAX_EVAL= - ∞
BEST_MOVE=NONE
APPELER MINIMAX RÉCURSIVEMENT AVEC PROFONDEUR -=1
APPELER MINIMAX RÉCURSIVEMENT AVEC PROFONDEUR -=1
METTRE À JOUR MIN_EVAL
METTRE À JOUR MAX_EVAL
SI LE MEILLEUR MIN_EVAL, METTRE À JOUR BEST_MOVE
SI LE MEILLEUR MAX_EVAL, METTRE À JOUR BEST_MOVE
RENVOYER MIN_EVAL ET BEST_MOVE
RENVOYER MAX_EVAL ET BEST_MOVE
FIN

4 ETUDES

PROFONDEUR	TEMPS (ms)	DIFFICULTE DU JOUEUR HUMAIN
1	2-3	Aucune, l'IA ne cherche pas à gagner
2	25	Facile
3	200-300	Moyen
4	1200-1500 (1,2-1,5 secondes)	Moyen+
5	12000-18000(12-18 secondes)	Difficile
6	97000-99000 (~1,40min)	Difficile
7	830000(~13min)	-

Complexitée Exponentielle:

$$\mathcal{O}(c^p)$$



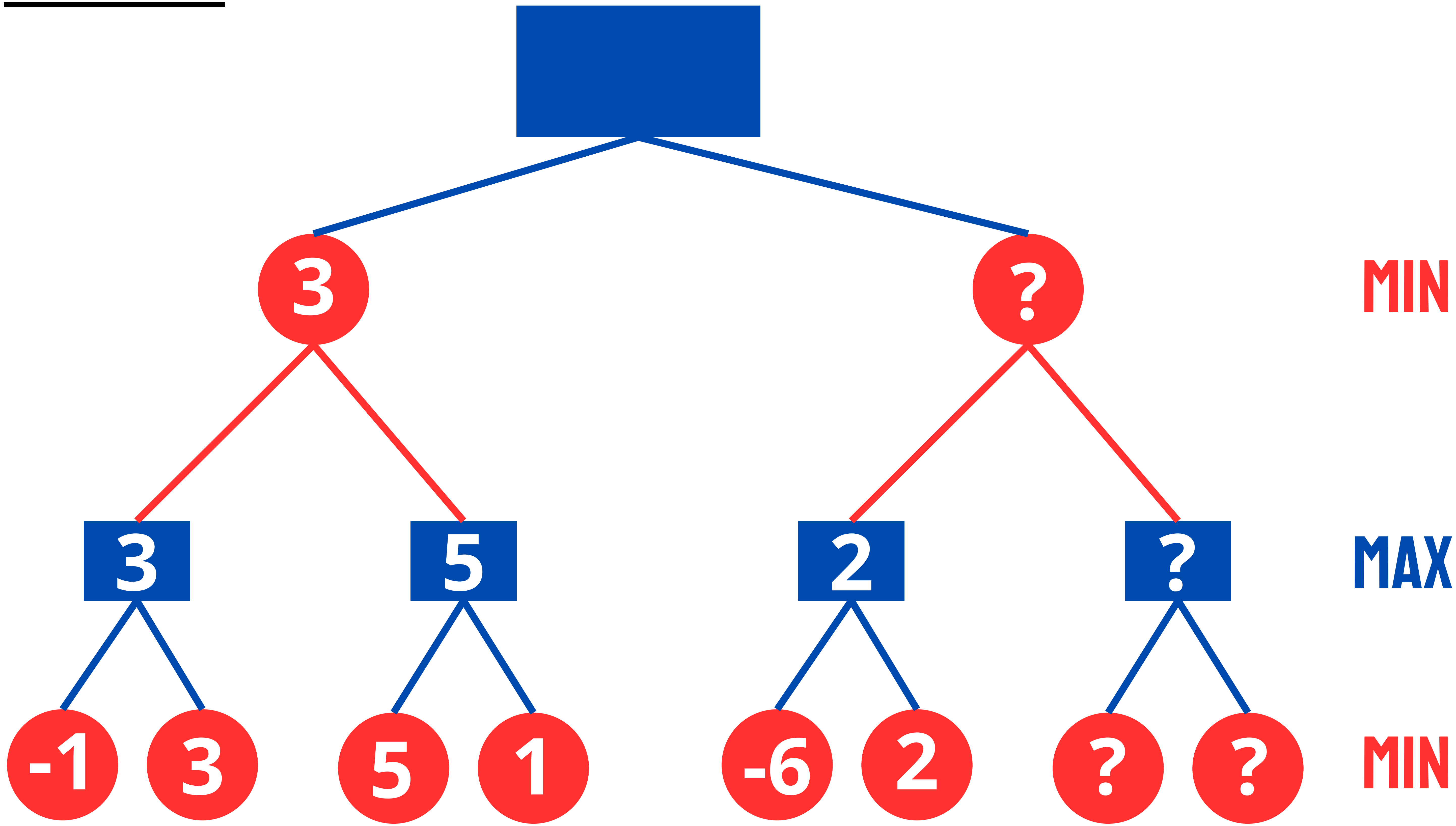
c : nombre total de coups possibles

p : profondeur

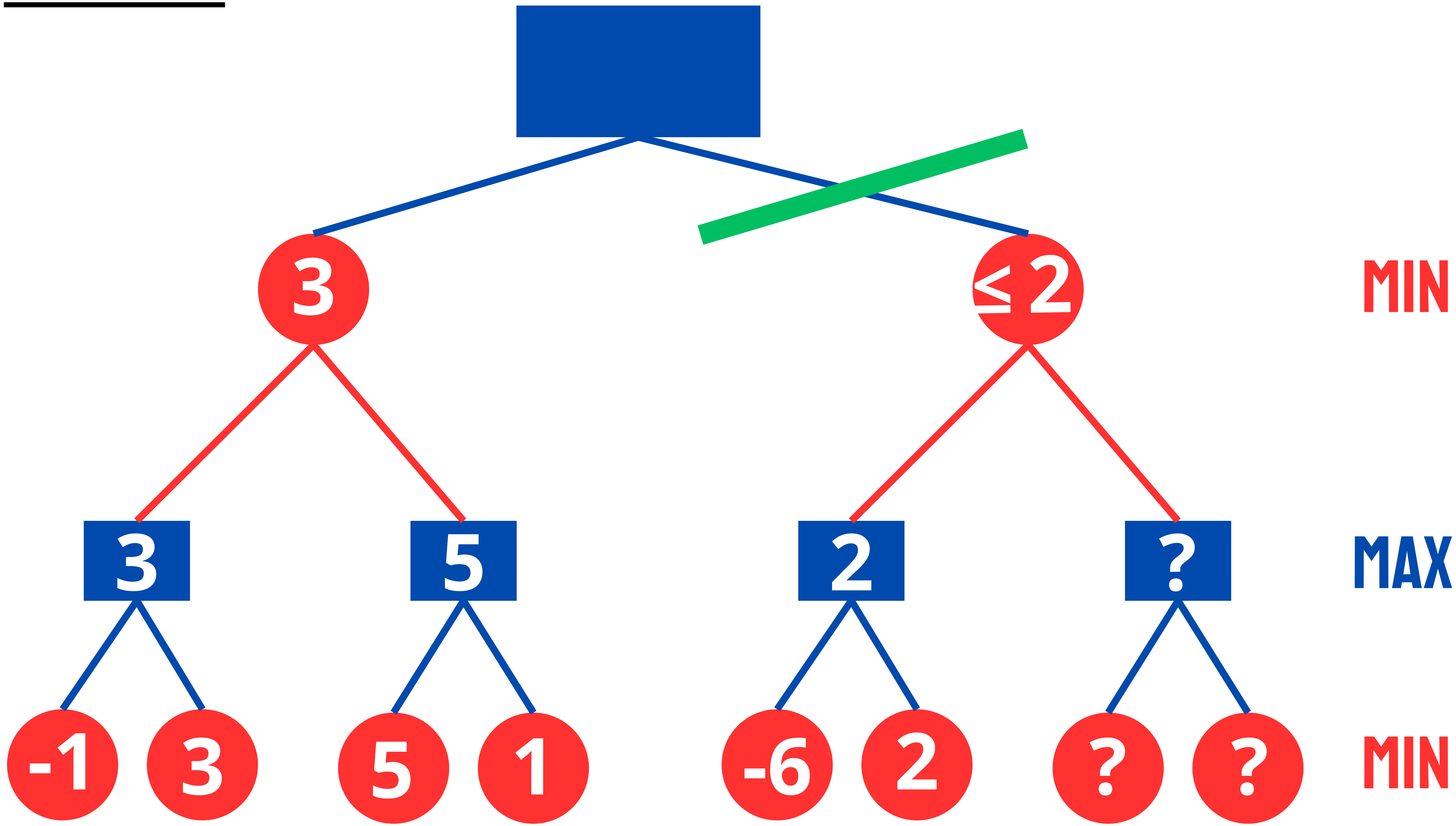
5 CONCLUSION

PROFONDEUR	TEMPS (ms)	DIFFICULTE
1	2-3	Aucune, l'IA ne cherche pas à gagner
2	25	Facile
3	200-300	Moyen
4	1200-1500 (1,2-1,5 secondes)	Moyen+
5	12000-18000(12-18 secondes)	Difficile
6	97000-99000 (~1,40min)	Difficile
7	830000(~13min)	-

Élagage alpha-bêta



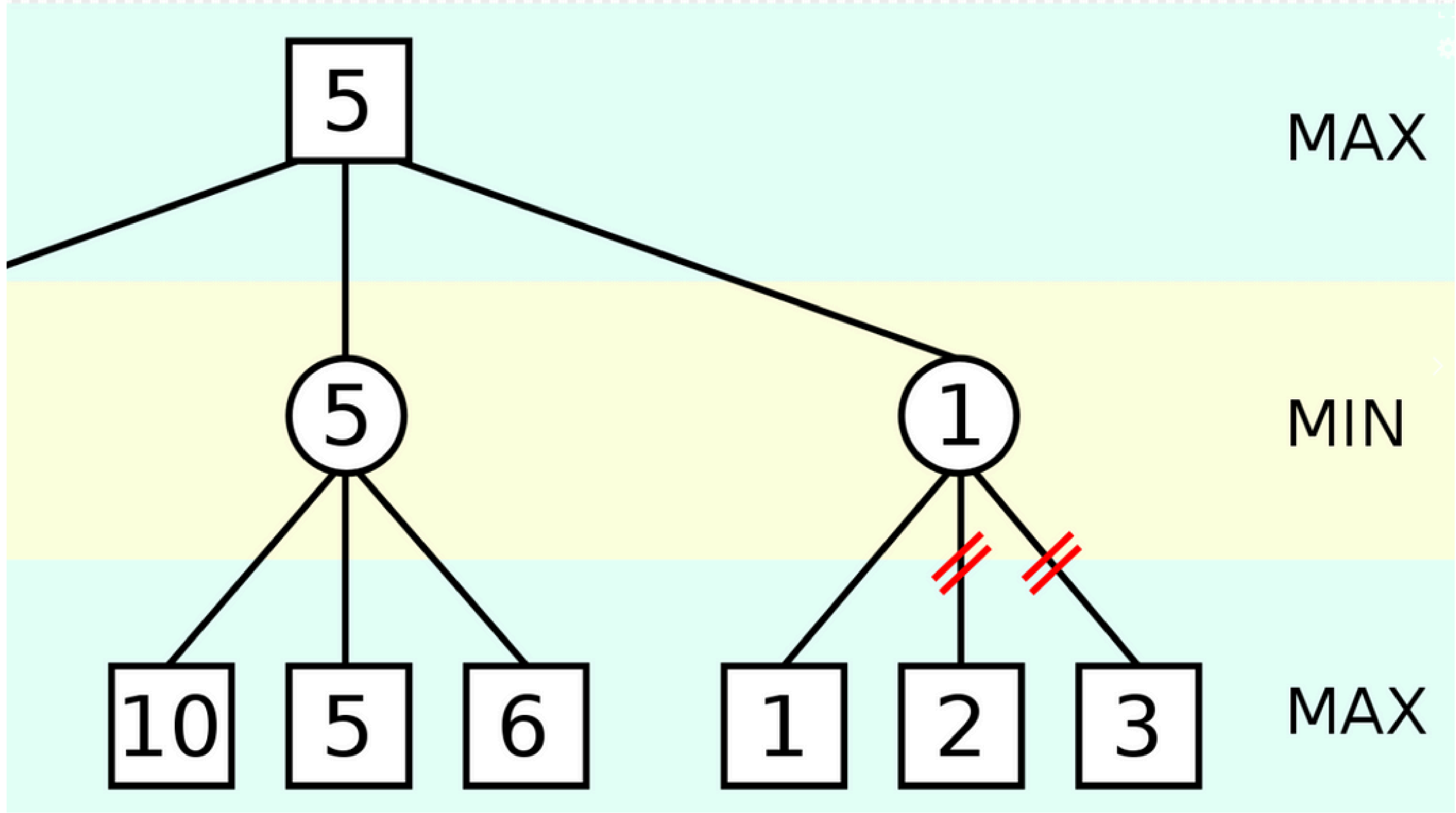
Élagage alpha-bêta



5 CONCLUSION

COMBIEN DE COUPS À L'AVANCE UNE INTELLIGENCE ARTIFICIELLE DOIT-ELLE PRÉVOIR POUR ÊTRE CAPABLE DE VAINCRE UN ÊTRE HUMAIN ?

2	25	Facile
3	200-300	Moyen
4	1200-1500 (1,2-1,5 secondes)	Moyen+
5	12000-18000(12-18 secondes)	Difficile



ÉLAGAGE ALPHA-BÊTA

Annexe-Jeu

```
def main():
    run = True
    clock = pygame.time.Clock()
    game = Game(WIN)

    while run:
        clock.tick(FPS)
        if game.turn == BLUE:
            start_ticks = pygame.time.get_ticks()
            value, new_board = minimax(game.get_board(), 7, BLUE, game)
            game.ai_move(new_board)
            end_ticks = pygame.time.get_ticks()
            ai_move_time = end_ticks - start_ticks
            print(f"AI move time: {ai_move_time} ms")
        if game.winner() != None:
            print(game.winner())
            run = False

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            if event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                row, col = get_row_col_from_mouse(pos)
                game.select(row, col)

        game.update()

    pygame.quit()

main()
```

CALCUL DU TEMPS DE L'IA

Annexe- Damier-1

```
import pygame
from .constants import BLACK, ROWS, RED, SQUARE_SIZE, COLS, WHITE, BLUE
from .piece import Piece

class Board:
    def __init__(self):
        self.board = []
        self.red_left = self.white_left = 12
        self.red_kings = self.white_kings = 0
        self.create_board()

    def draw_squares(self, win):
        win.fill(BLACK)
        for row in range(ROWS):
            for col in range(row % 2, COLS, 2):
                pygame.draw.rect(win, WHITE, (row*SQUARE_SIZE, col *SQUARE_SIZE, SQUARE_SIZE, SQUARE_SIZE))

    def evaluate(self):
        return self.white_left - self.red_left + (self.white_kings * 0.5 - self.red_kings * 0.5)

    def get_all_pieces(self, color):
        pieces = []
        for row in self.board:
            for piece in row:
                if piece != 0 and piece.color == color:
                    pieces.append(piece)
        return pieces

    def move(self, piece, row, col):
        self.board[piece.row][piece.col], self.board[row][col] = self.board[row][col], self.board[piece.row][piece.col]
        piece.move(row, col)

        if row == ROWS - 1 or row == 0:
            piece.make_king()
            if piece.color == BLUE:
                self.white_kings += 1
            else:
                self.red_kings += 1

    def get_piece(self, row, col):
        return self.board[row][col]
```

Annexe- Damier-2

```
    def draw(self, win):
        self.draw_squares(win)
        for row in range(ROWS):
            for col in range(COLS):
                piece = self.board[row][col]
                if piece != 0:
                    piece.draw(win)

    def remove(self, pieces):
        for piece in pieces:
            self.board[piece.row][piece.col] = 0
            if piece != 0:
                if piece.color == RED:
                    self.red_left -= 1
                else:
                    self.white_left -= 1

    def winner(self):
        if self.red_left <= 0:
            return BLUE
        elif self.white_left <= 0:
            return RED

        return None

    def get_valid_moves(self, piece):
        moves = {}
        left = piece.col - 1
        right = piece.col + 1
        row = piece.row

        if piece.color == RED or piece.king:
            moves.update(self._traverse_left(row -1, max(row-3, -1), -1, piece.color, left))
            moves.update(self._traverse_right(row -1, max(row-3, -1), -1, piece.color, right))
        if piece.color == BLUE or piece.king:
            moves.update(self._traverse_left(row +1, min(row+3, ROWS), 1, piece.color, left))
            moves.update(self._traverse_right(row +1, min(row+3, ROWS), 1, piece.color, right))

        return moves
```

Annexe- Mouvement

```python
    def _traverse_left(self, start, stop, step, color, left, skipped=[]):
        moves = {}
        last = []
        for r in range(start, stop, step):
            if left < 0:
                break

            current = self.board[r][left]
            if current == 0:
                if skipped and not last:
                    break
                elif skipped:
                    moves[(r, left)] = last + skipped
                else:
                    moves[(r, left)] = last

                if last:
                    if step == -1:
                        row = max(r-3, 0)
                    else:
                        row = min(r+3, ROWS)
                    moves.update(self._traverse_left(r+step, row, step, color, left-1,skipped=last))
                    moves.update(self._traverse_right(r+step, row, step, color, left+1,skipped=last))
                break
            elif current.color == color:
                break
            else:
                last = [current]

            left -= 1

        return moves
```

Annexe- Algo

```python
def simulate_move(piece, move, board, game, skip):
    board.move(piece, move[0], move[1])
    if skip:
        board.remove(skip)

    return board


def get_all_moves(board, color, game):
    moves = []

    for piece in board.get_all_pieces(color):
        valid_moves = board.get_valid_moves(piece)
        for move, skip in valid_moves.items():
            #draw_moves(game, board, piece)
            temp_board = deepcopy(board)
            temp_piece = temp_board.get_piece(piece.row, piece.col)
            new_board = simulate_move(temp_piece, move, temp_board, game, skip)
            moves.append(new_board)

    return moves


def draw_moves(game, board, piece):
    valid_moves = board.get_valid_moves(piece)
    board.draw(game.win)
    pygame.draw.circle(game.win, (0,255,0), (piece.x, piece.y), 50, 5)
    game.draw_valid_moves(valid_moves.keys())
    pygame.display.update()
    pygame.time.delay(10)
```

Annexe- MinMax

```python
def minimax(position, depth, max_player, game):
    if depth == 0 or position.winner() != None:
        return position.evaluate(), position

    if max_player:
        pygame.init()
        maxEval = float('-inf')
        best_move = None
        for move in get_all_moves(position, BLUE, game):
            evaluation = minimax(move, depth-1, False, game)[0]
            maxEval = max(maxEval, evaluation)
            if maxEval == evaluation:
                best_move = move

        return maxEval, best_move
    else:
        minEval = float('inf')
        best_move = None
        for move in get_all_moves(position, RED, game):...

        return minEval, best_move
```

Tracé

```python
import matplotlib.pyplot as plt

depth = [1, 2, 3, 4, 5, 6]
temps_de_reponse = [2, 25, 300, 1350, 12000, 99000]


plt.figure(figsize=(8, 6))
plt.plot(depth, temps_de_reponse, marker='o', linestyle='-')

plt.yscale('log')
plt.xlabel('Nombres de coups à prévoir')
plt.ylabel('Temps de réponse (ms)')
plt.show()
```

```python
def minimax(position, depth, max_player, game):
    if depth == 0 or position.winner() != None:
        return position.evaluate(), position

    if max_player:
        pygame.init()
        maxEval = float('-inf')
        best_move = None
        for move in get_all_moves(position, BLUE, game):
            evaluation = minimax(move, depth-1, False, game)[0]
            maxEval = max(maxEval, evaluation)
            if maxEval == evaluation:
                best_move = move

        return maxEval, best_move
    else:
        minEval = float('inf')
        best_move = None
        for move in get_all_moves(position, RED, game):...

        return minEval, best_move
```