

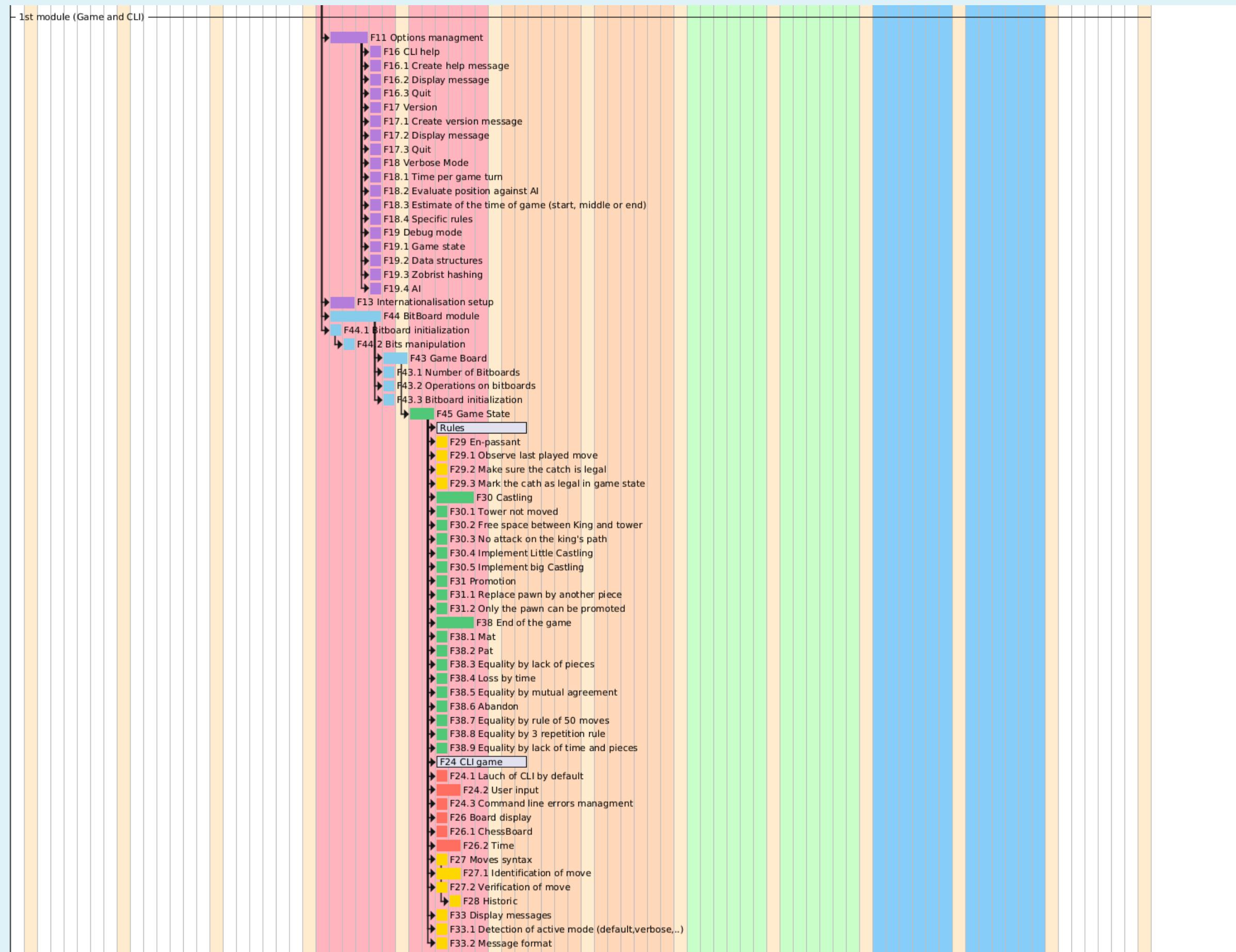
# ÉCHECS JAVA

Mathilde Chollon, Valentin  
Jonquièrre, Denis Demirci, Iwen  
Jomaa, Jonathan Landry

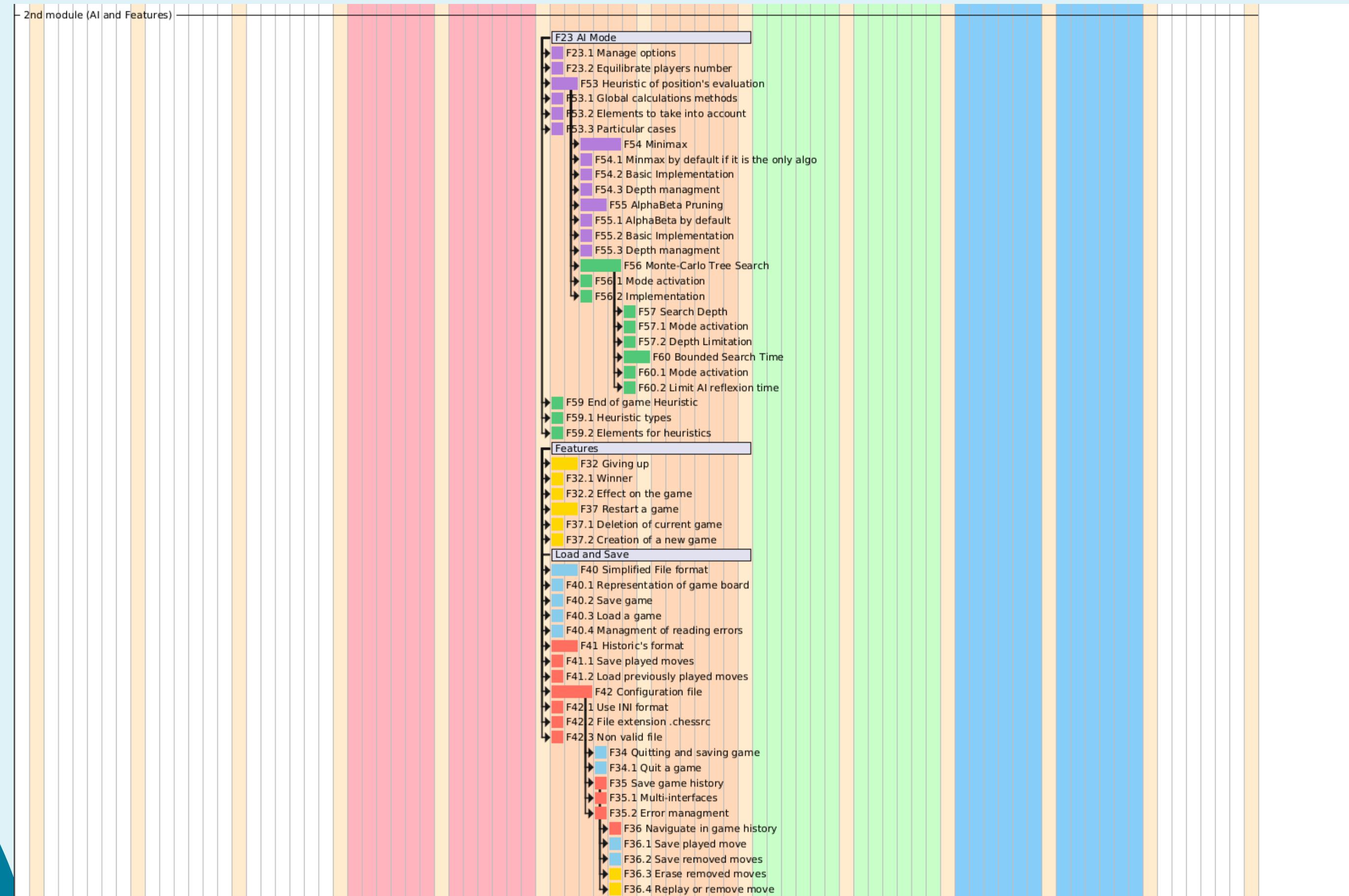
# SOMMAIRE

- 1. Agenda prévisionnel**
- 2. Architecture**
- 3. Spécifications étendues**
- 4. Contexte**
- 5. Explication du sujet**
- 6. Besoins visés**

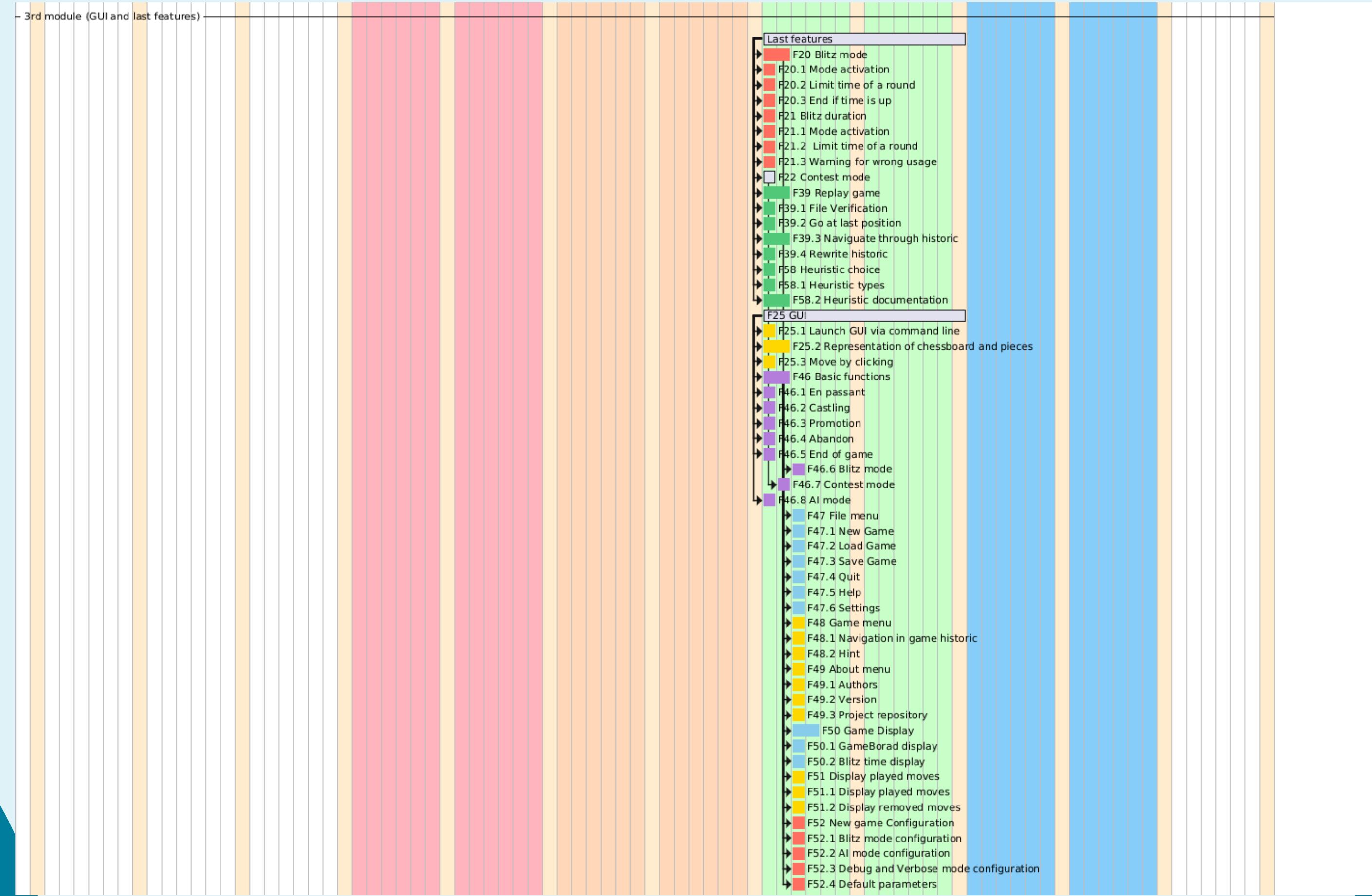
# SEMAINES 0-2



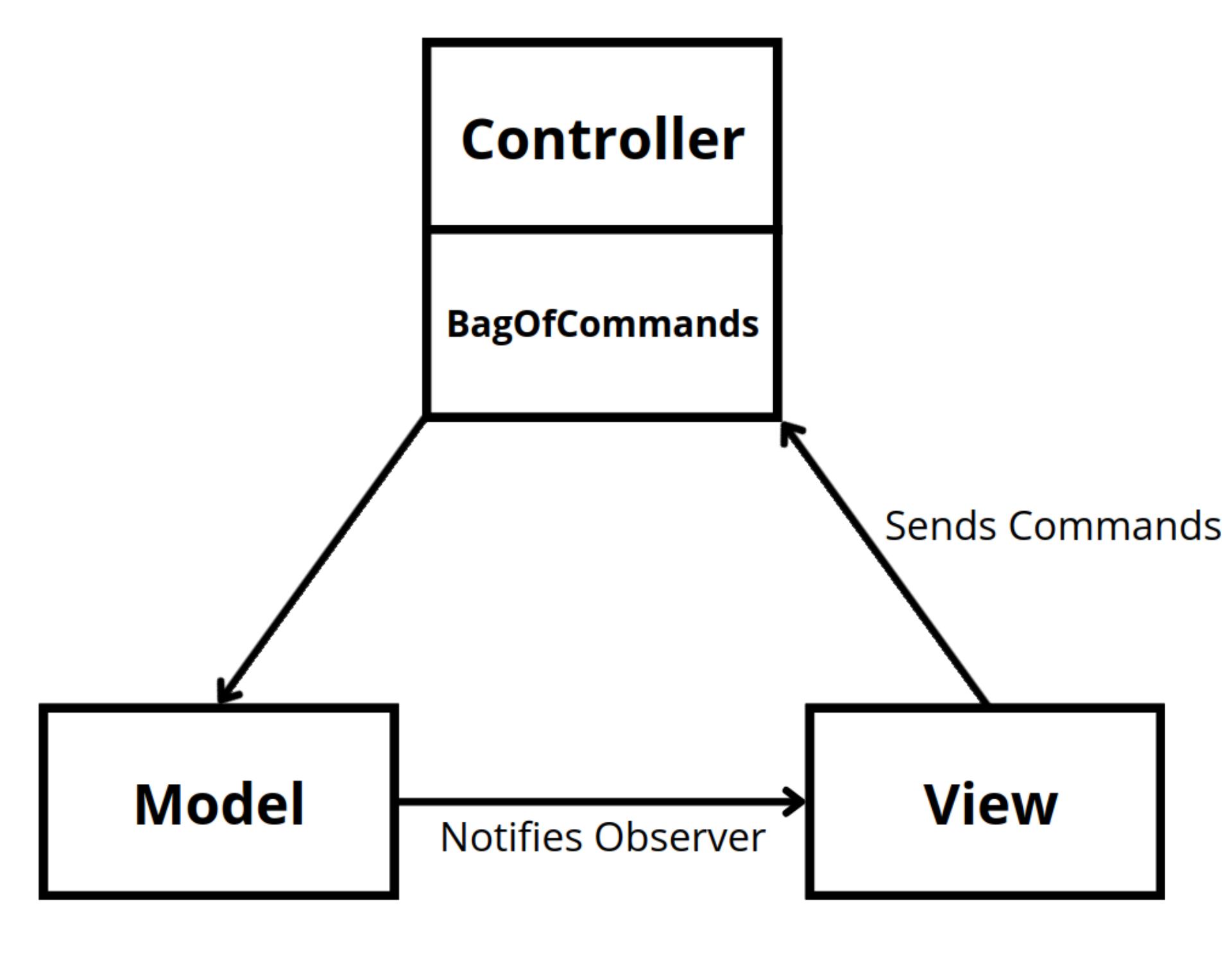
# SEMAINES 2-4



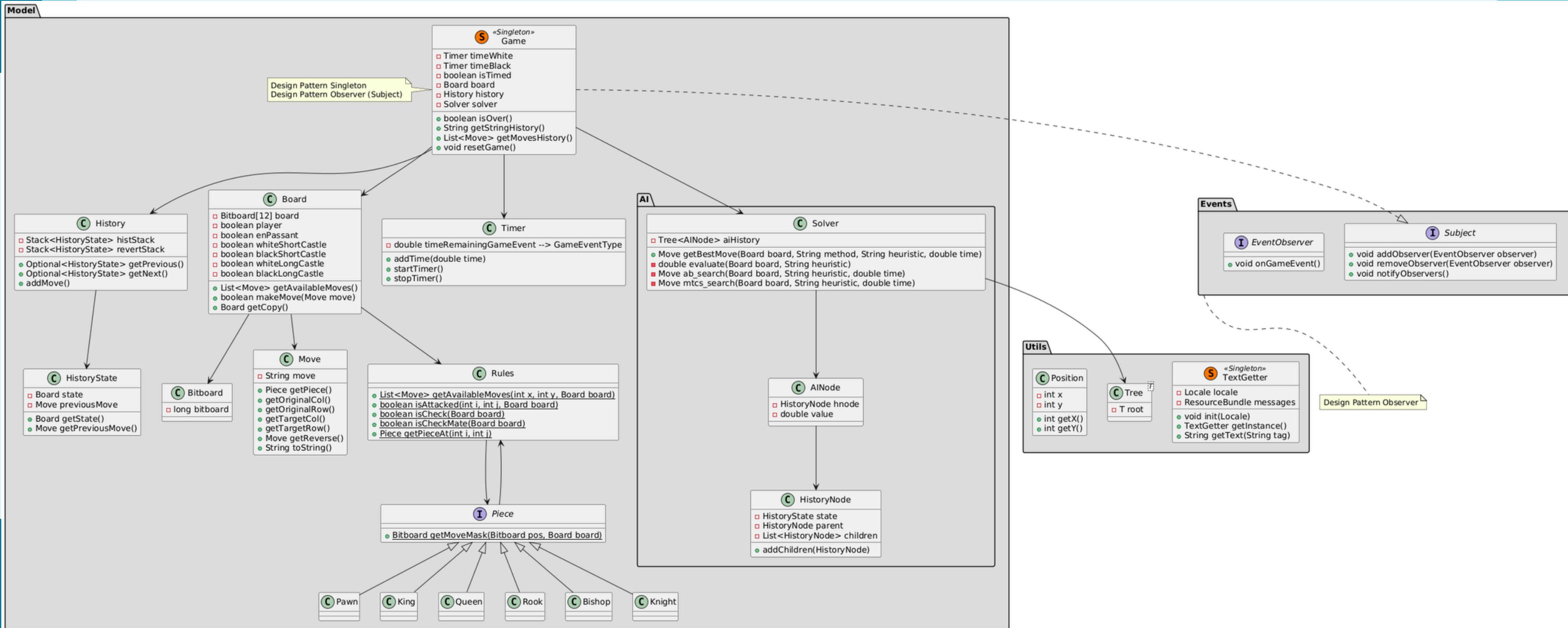
# SEMAINES 4-6



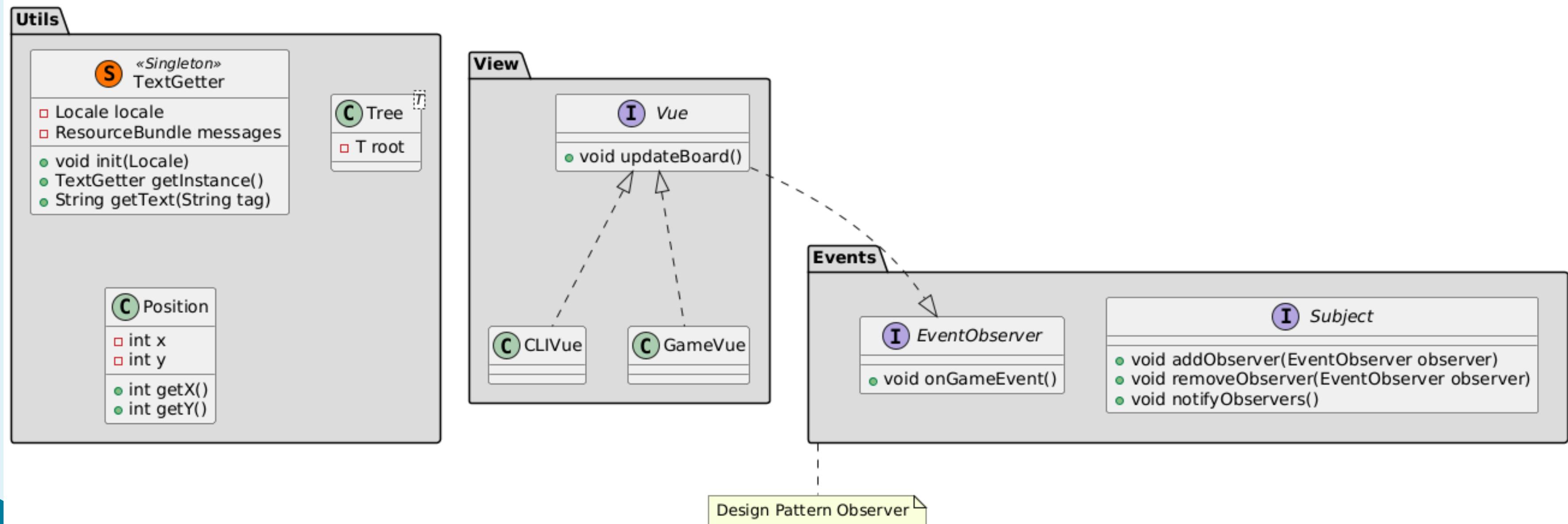
# ARCHITECTURE



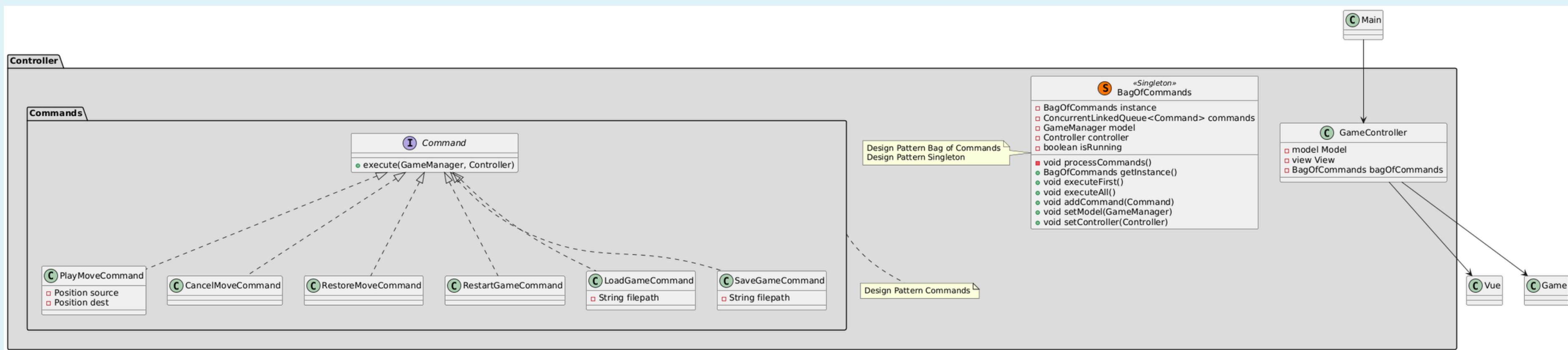
# MODEL



# VIEW



# CONTROLLER



# ARCHITECTURE

## Modules

- Model
- View
- Controller
- Events
- Utils

## Design patterns

- MVC modifié
- Observer
- Singleton
- State
- Bag of Commands
- Commands

# SPÉCIFICATIONS ÉTENDUES

Présentation de quelques besoins et de leurs sous-besoins.

Plus de précisions et de besoins dans le rapport.

# F30: ROQUE

## Gestion des petits et grands roques

- Roi et tours n'ont pas bougé

**Test:** Faire bouger le Roi ou l'une des tours et s'assurer que le roque n'est plus possible

- Espace libre entre le Roi et la tour concernée

**Test:** Check si le roque est réalisable alors qu'une ou plusieurs pièces séparent le Roi et la tour

- Le Roi n'est jamais en échec

**Test:** Valider le fait qu'aucune pièce adverse n'attaque l'une des cases sur lesquelles le Roi va se trouver

- Faire bouger les pièces pour implémenter les deux roques

**Test:** Vérifier que les pièces (Roi et tour) ont correctement été bougées sur le plateau et les règles mises à jour

# F20: MODE BLITZ

## Lancer le jeu en mode Blitz

- Activation du mode

**Test:** Lancer le jeu avec l'option, vérifier que la configuration a changé.

- Limiter le temps d'un tour

**Test:** Vérifier que le timer se remette à zéro à chaque tour

- Fin de partie si temps écoulé

**Test:** Si le temps est écoulé, vérifier que le joueur perd la partie

# F24: INTERFACE TERMINAL

- Lancement de l'interface terminal par défaut

**Test:** Lancer le jeu avec l'option, vérifier que la configuration du jeu a été modifiée.

- Saisie des commandes utilisateur

**Test:** Vérifier que les commandes de base fonctionnent comme attendu.

- Gestion des erreurs de commande

**Test:** Entrer des commandes invalides et vérifier les messages d'erreur.

# F59: HEURISTIQUES DE FIN DE PARTIE

## Création d'heuristiques adaptées aux finales

- Déetecter la fin de partie (non-présence des Dames, nb de pièces, nb de coups joués et possibles,...)

**Test:** Valider chaque élément qui constitue un facteur de fin de partie

- Facteurs à prendre en compte lors de finales (promotion de pion, nb de coups légaux, activité des pièces,...)

**Test:** Valider chaque élément qui fait partie d'une heuristique de fin de partie.

# F40: FORMAT DE FICHIER

## Sauvegarde et restauration de fichier

- Représentation du plateau de jeu : ASCII

- Sauvegarde du jeu

**Test:** Sauvegarder une partie. Vérifier que le plateau correspond à celui de la partie.

- Chargement de fichier

**Test:** Restaurer un fichier, vérifier que le plateau affiché correspond à celui du fichier

- Gestion des erreurs

**Test:** Créer un fichier avec des erreurs, vérifier le code de sortie

# F35: SAUVEGARDE DE L'HISTORIQUE

- Multi-interfaces ( terminal & graphique )

**Test:** Tester la sauvegarde en mode graphique et en ligne de commande.

- Gestion des erreurs

**Test:** Simuler une sauvegarde réussie et échouée.

- Partie en cours / finie

**Test:** Vérifier que la sauvegarde fonctionne pour une partie en cours et finie.

# F31: PROMOTION DE PION

## Gestion de promotion en une autre pièce

- Changer de pièce

**Test:** Faire arriver un pion sur la dernière rangée et check le changement (mouvements nouvelle pièce + bitboards)

- Valable seulement pour le pion

**Test:** Faire arriver une autre pièce que le pion sur la dernière rangée et s'assurer qu'il n'y a pas de changement ou de demande de changement

# F60: TEMPS DE RÉFLEXION IA

## Limiter le temps de “réflexion” de l’IA

- Activation du mode

**Test:** Lancer le jeu avec l’option, vérifier que la configuration du jeu a été modifiée.

- Limiter le temps de réflexion de l’IA

**Test:** Vérifier que l’IA ne prend pas plus de temps à jouer que le temps donné.

# F54: ALGORITHME DE RECHERCHE MINIMAX

- Activer Minimax par défaut

**Test:** Vérifier que l'option est bien active seulement si AlphaBeta-pruning n'est pas implémenté.

- Implémentation de base

**Test:** Vérifier que Minimax calcule les meilleurs mouvements pour différents scénarios.

- Gestion des niveaux de profondeur

**Test:** Vérifier les performances à différentes profondeur.

# F27 : NOTATION DES COUPS

- Reconnaissance du coup

**Test:** Jouer un coup et vérifier que le bon coup a été effectué (comparaison entre l'état final et attendu).

- Vérification du coup

**Test:** Jouer des coups légaux et des coups illégaux. Vérifier que seuls les coups légaux sont acceptés.

# BESOINS BONUS

# F39: REJOUER UNE PARTIE

## Charger un fichier, naviguer dans les coups et rejouer la partie depuis un coup

- Vérification du fichier

**Test:** Bon format de fichier (extension,...) et contrôler la notation des coups

- Se placer à la dernière position

**Test:** Vérifier si toutes les règles sont valides tout au cours de la partie, partie bien initialisée et observer la pile de l'historique

- Navigation dans l'historique

**Test:** Valider le fait que les coups sont correctement repris et rejoués

- Ecrasement de l'historique

**Test:** S'assurer que le fichier est correctement modifié et formé. Check notamment la pile de l'historique

# CONTEXTE DU PROJET

# NOS BESOINS

- Un logiciel modulaire
- Facile à maintenir et développer
- Utiliser la programmation objet

# LE LANGAGE C



- Des performances imbattables



- Gestion de la mémoire très chronophage
- Pas de programmation objet

# LE PYTHON



- Pas de typage statique
- Complètement interprété

# LE JAVA



- Langage pensé pour l'objet
- Portable et performant
- Utilisé par tous les membres de l'équipe dans le cadre d'autres projets

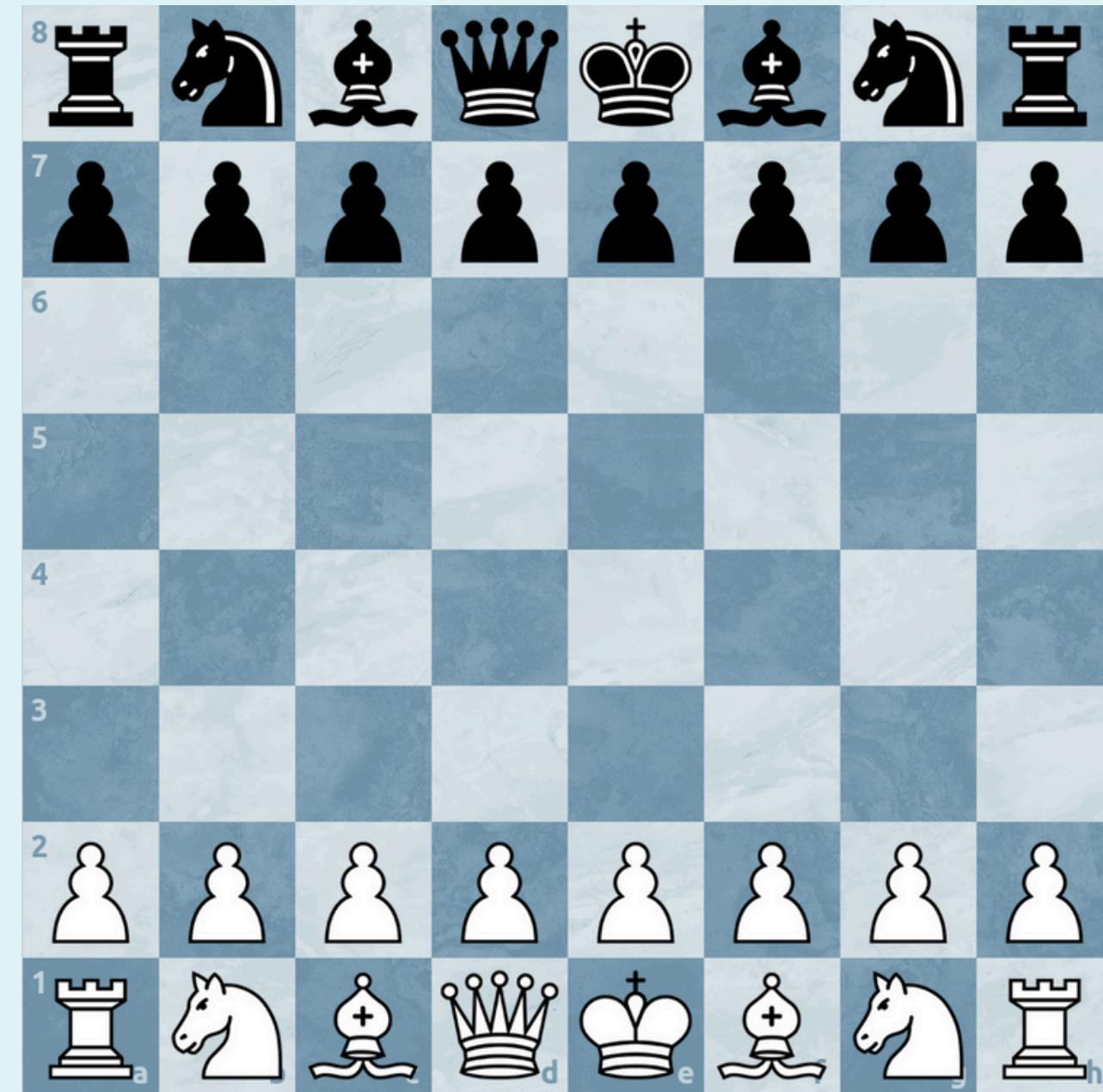
# EXISTANT

- Claude E. Shannon. Programming a computer for playing chess.
- Jakub Kral. Using monte carlo tree search to play chess.
- Zobrist hashing
- Pieter Bijl and Anh Phi Tiet, Exploring modern chess engine architectures

# **EXPLICATION DU SUJET**

# LES ECHECS

- 2 joueurs
- Plateau de 64 cases
- 16 pièces par joueur



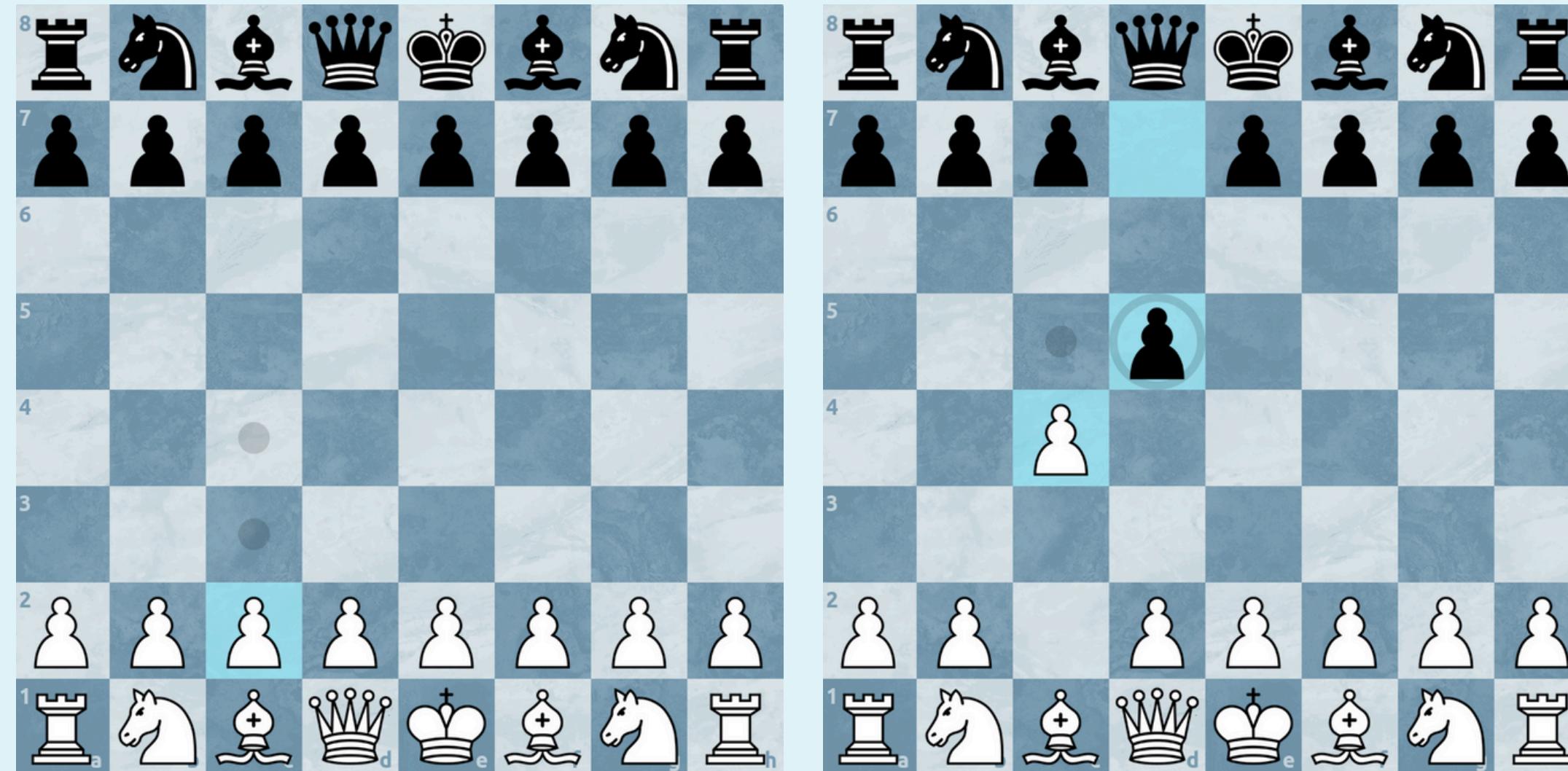
# L'OBJECTIF

- **L'objectif** des échecs est de mettre le roi adverse en échec et mat. Cela signifie que le roi est attaqué et qu'il n'y a aucun moyen légal de le défendre contre cette attaque. Le joueur qui réussit cela gagne la partie.

# LES DEPLACEMENTS

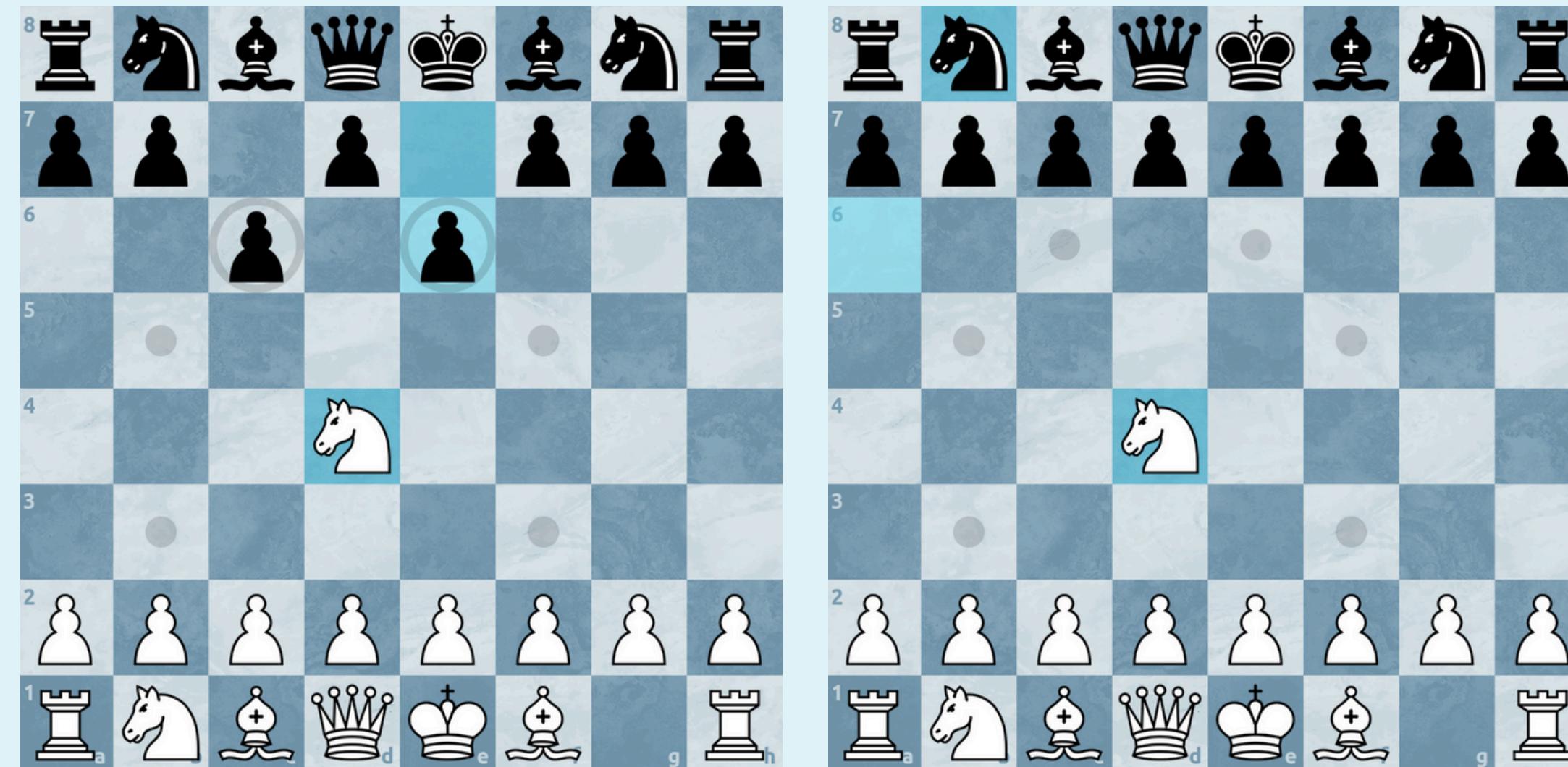
- Les **déplacements** sont unique à chaque type de pièces

## LE PION



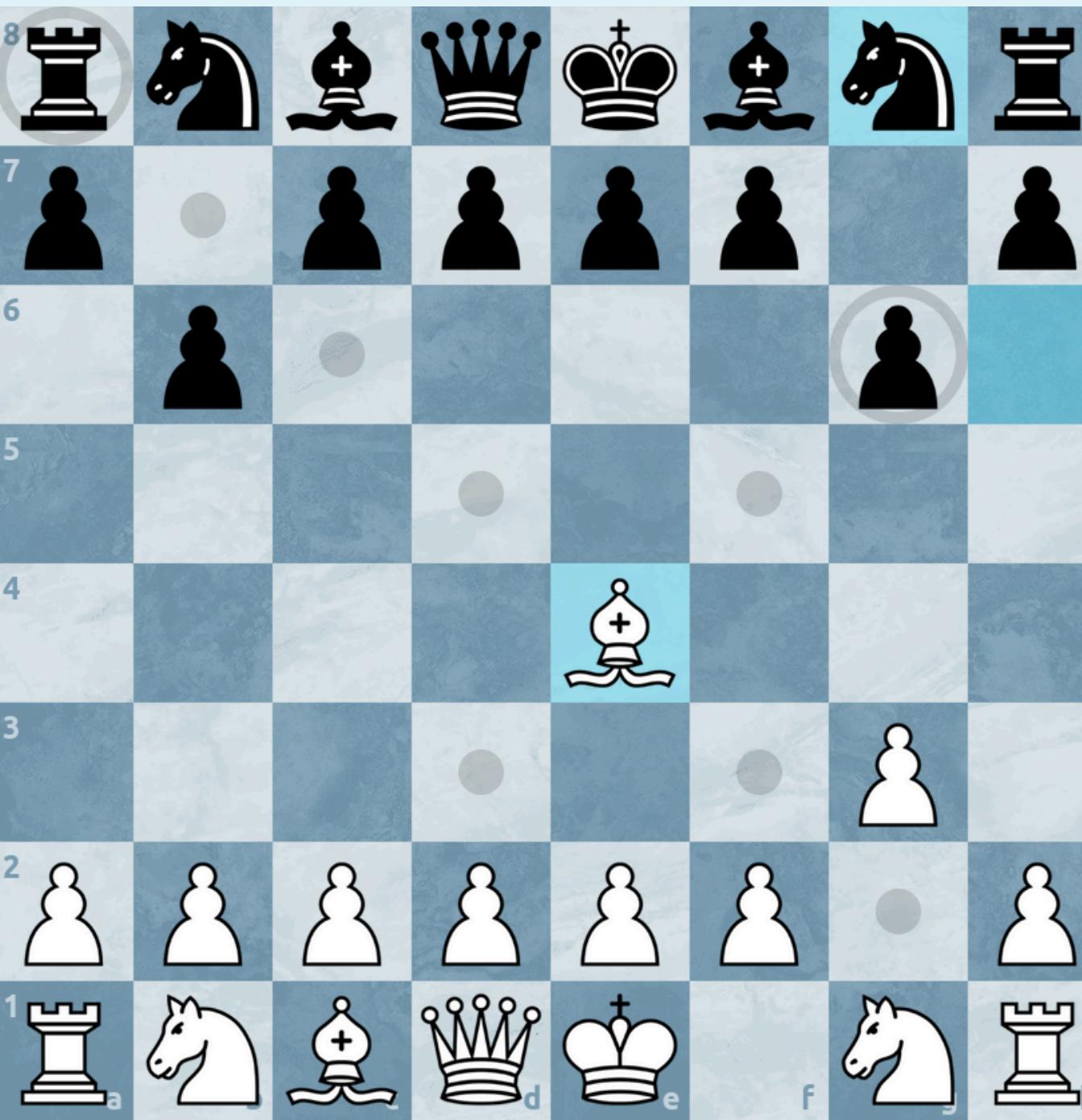
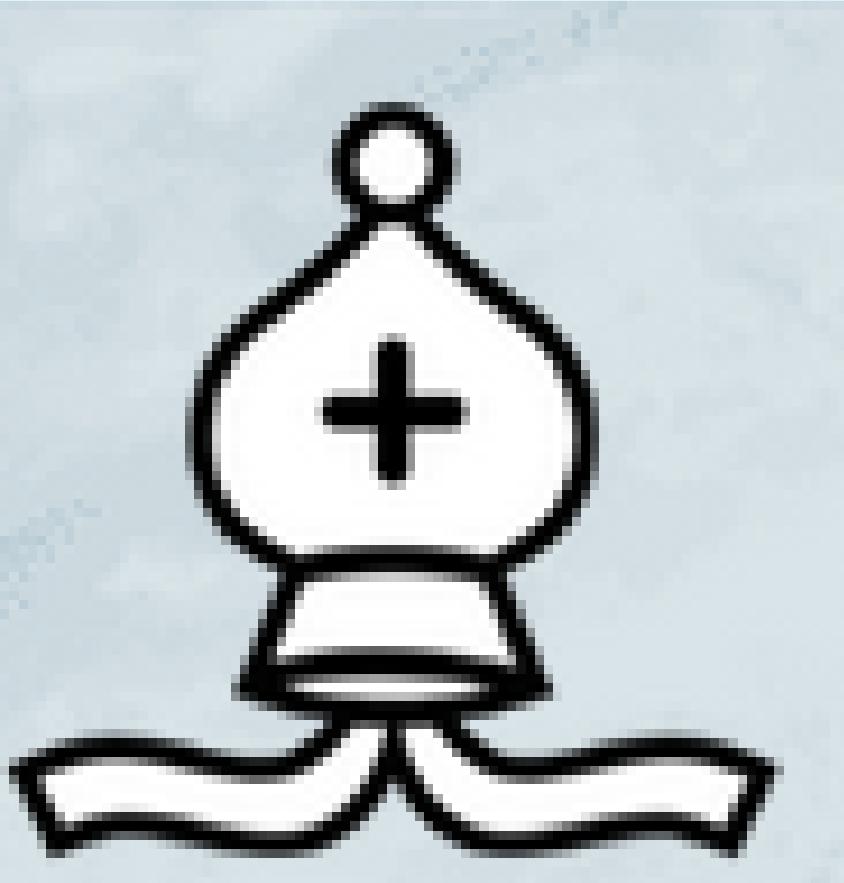
# LES DEPLACEMENTS

## LE CAVALIER



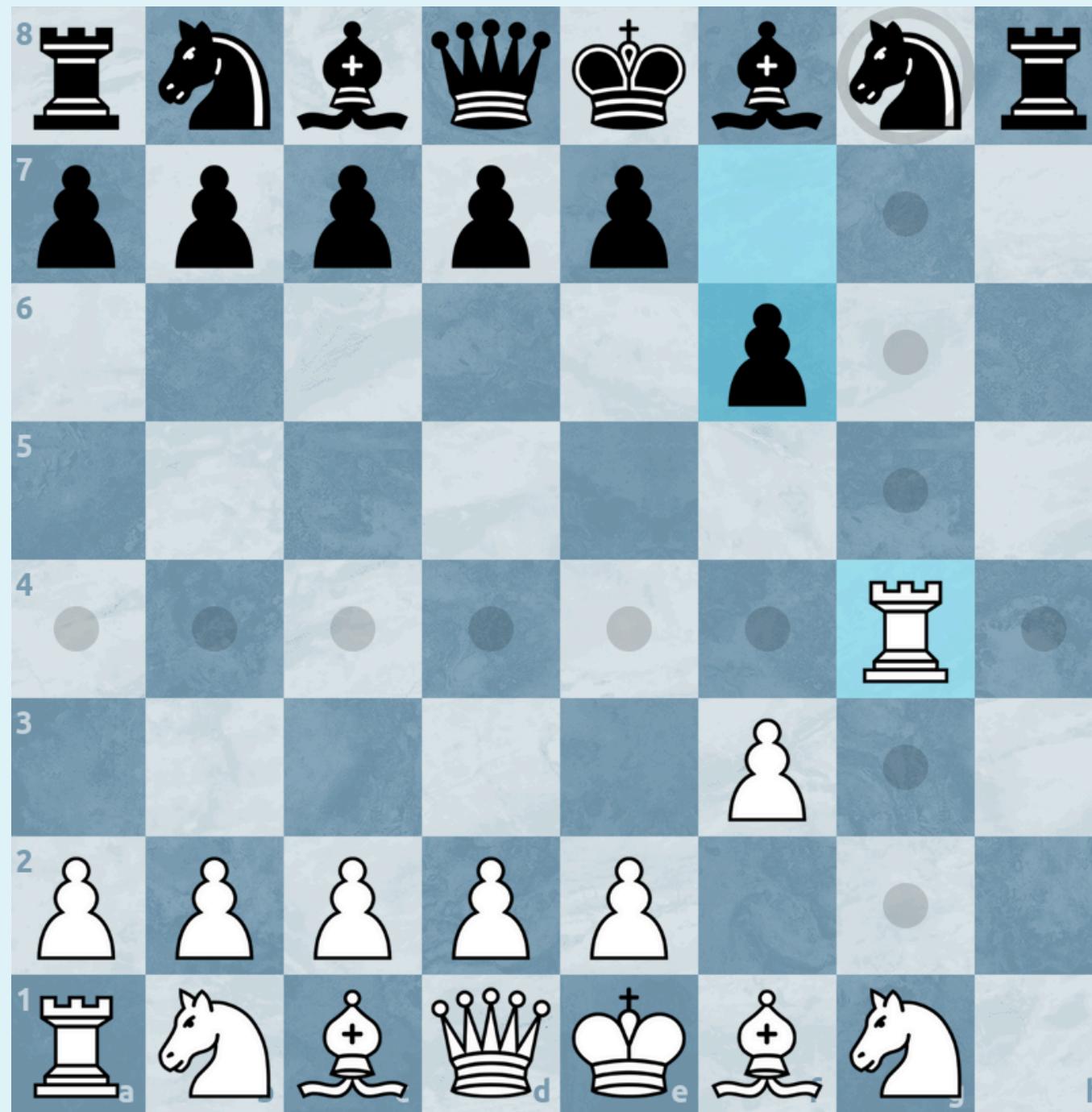
# LES DEPLACEMENTS

LE FOU

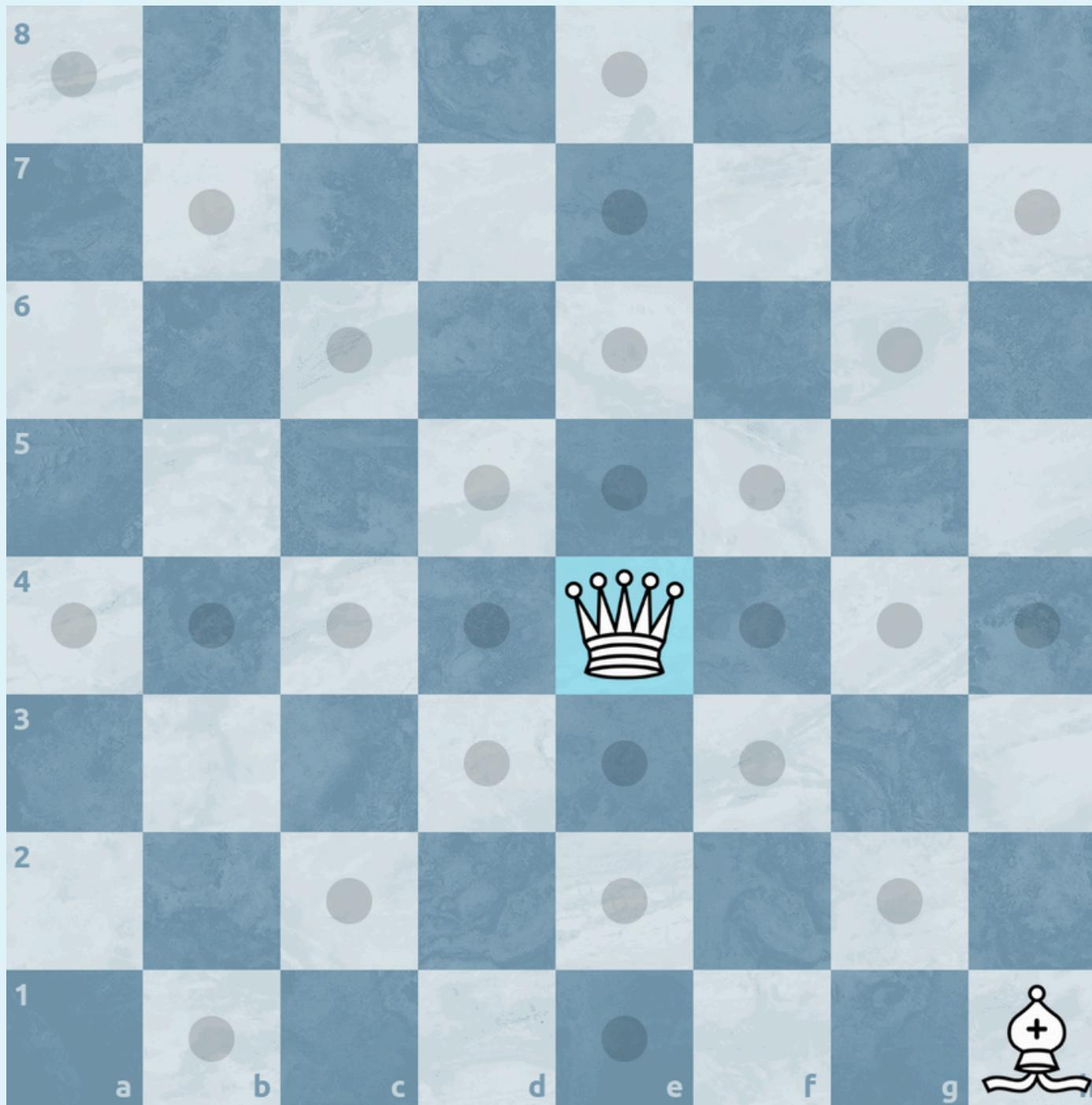


# LES DEPLACEMENTS

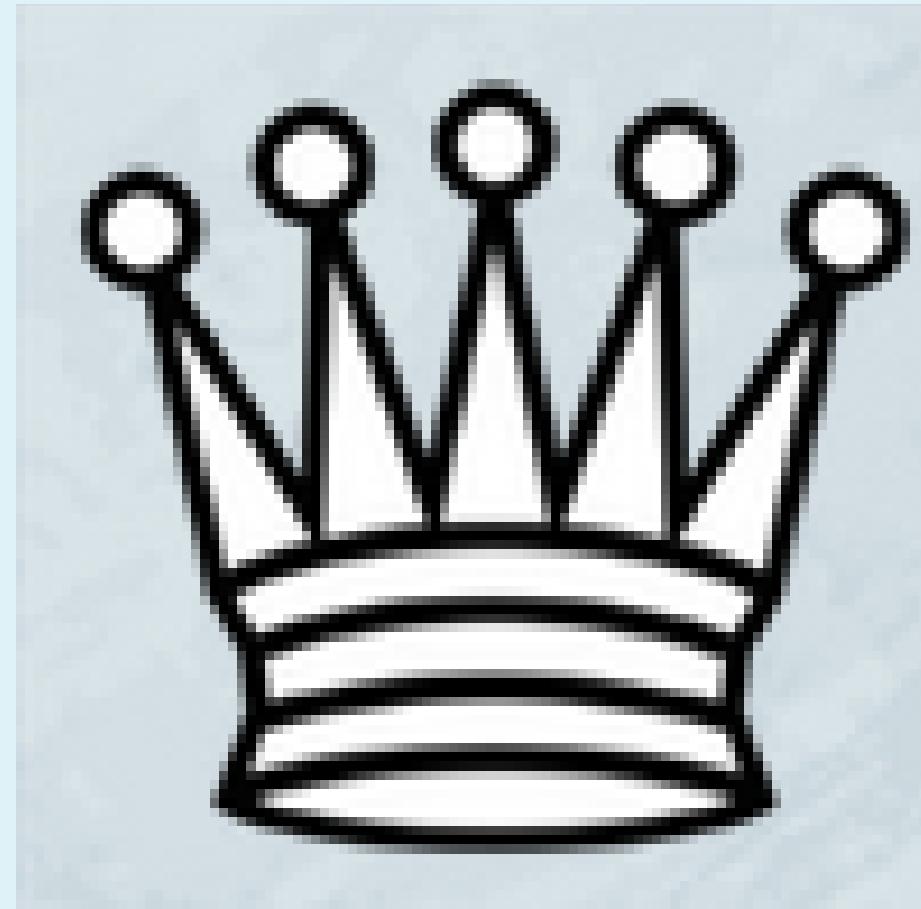
LA TOUR



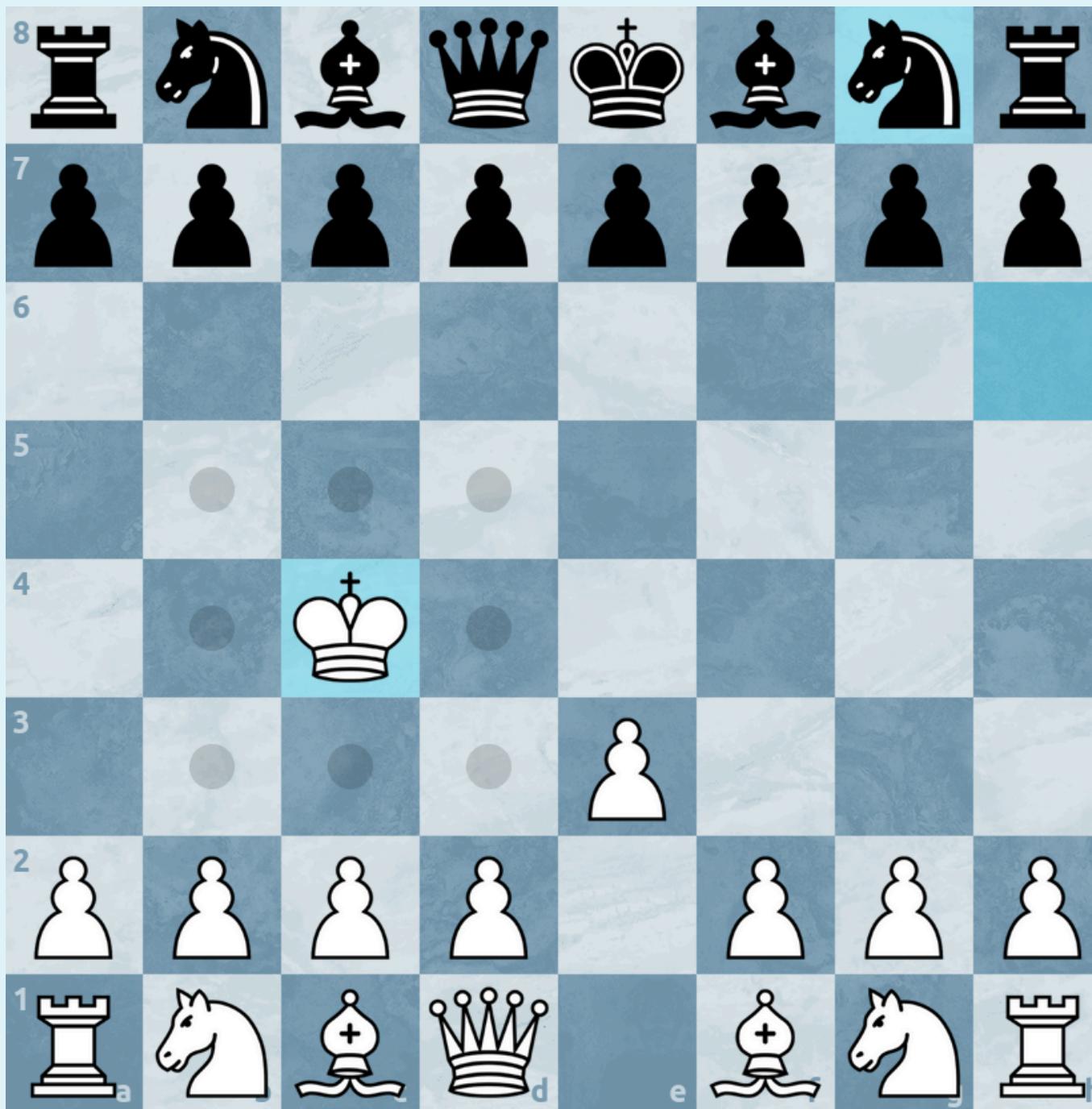
# LES DEPLACEMENTS



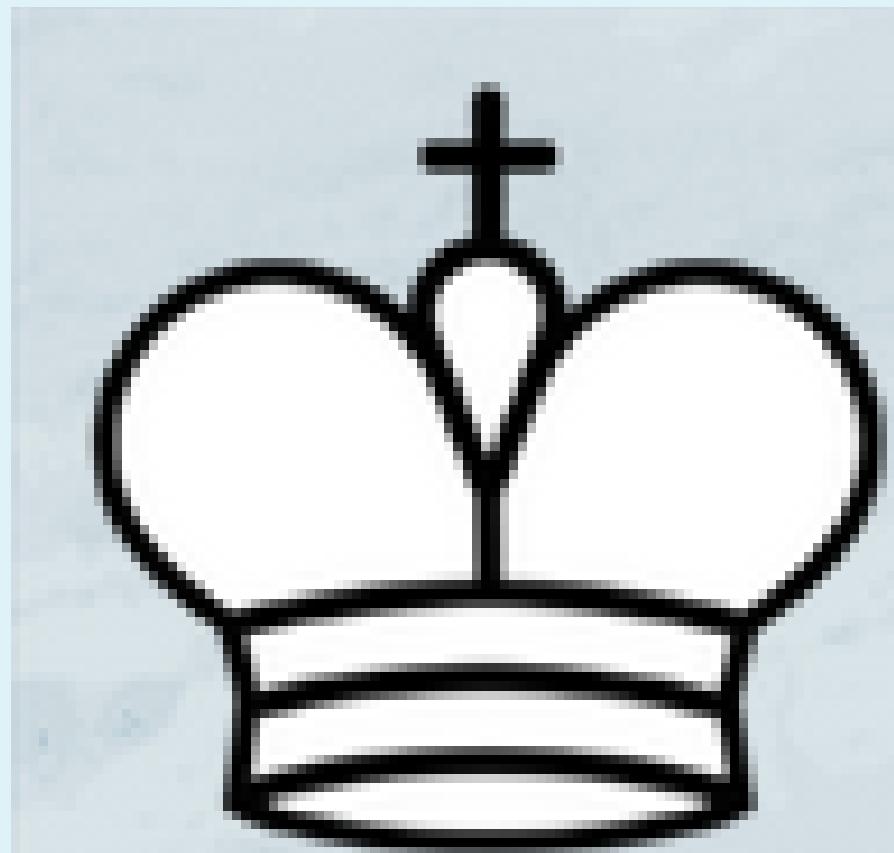
LA DAME



# LES DEPLACEMENTS

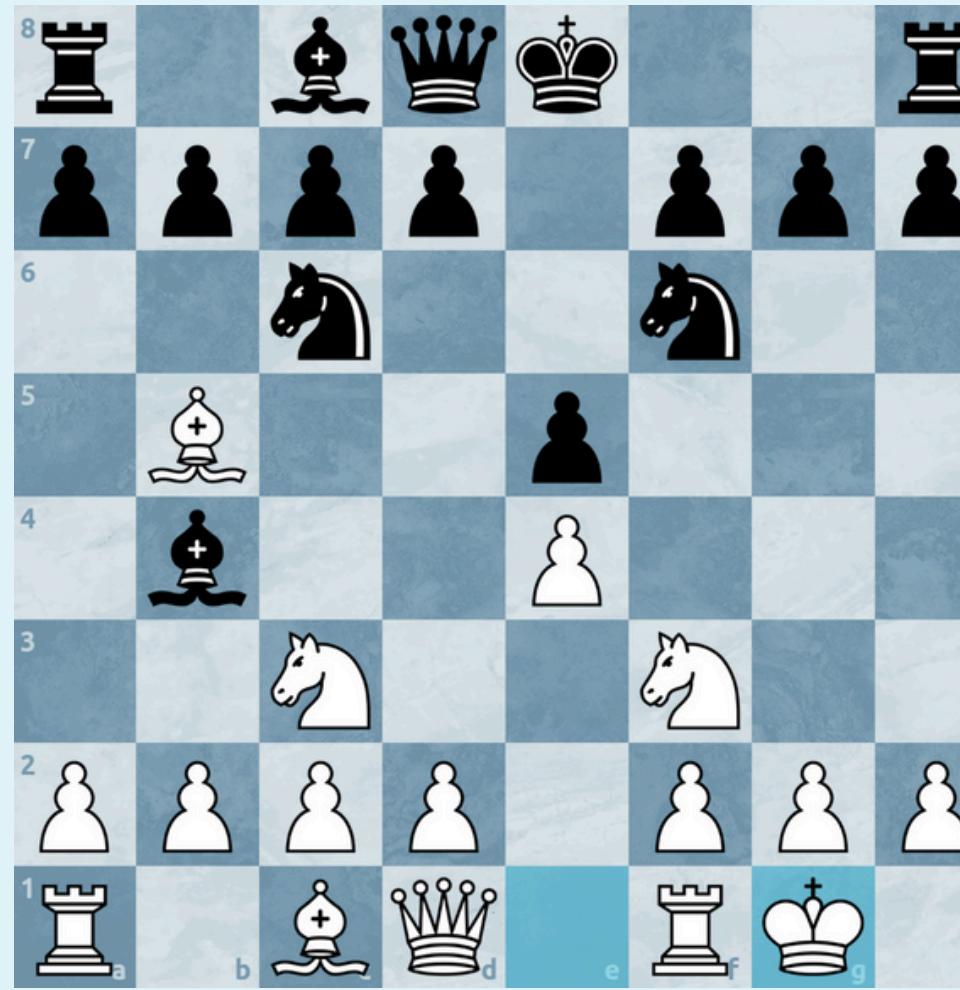
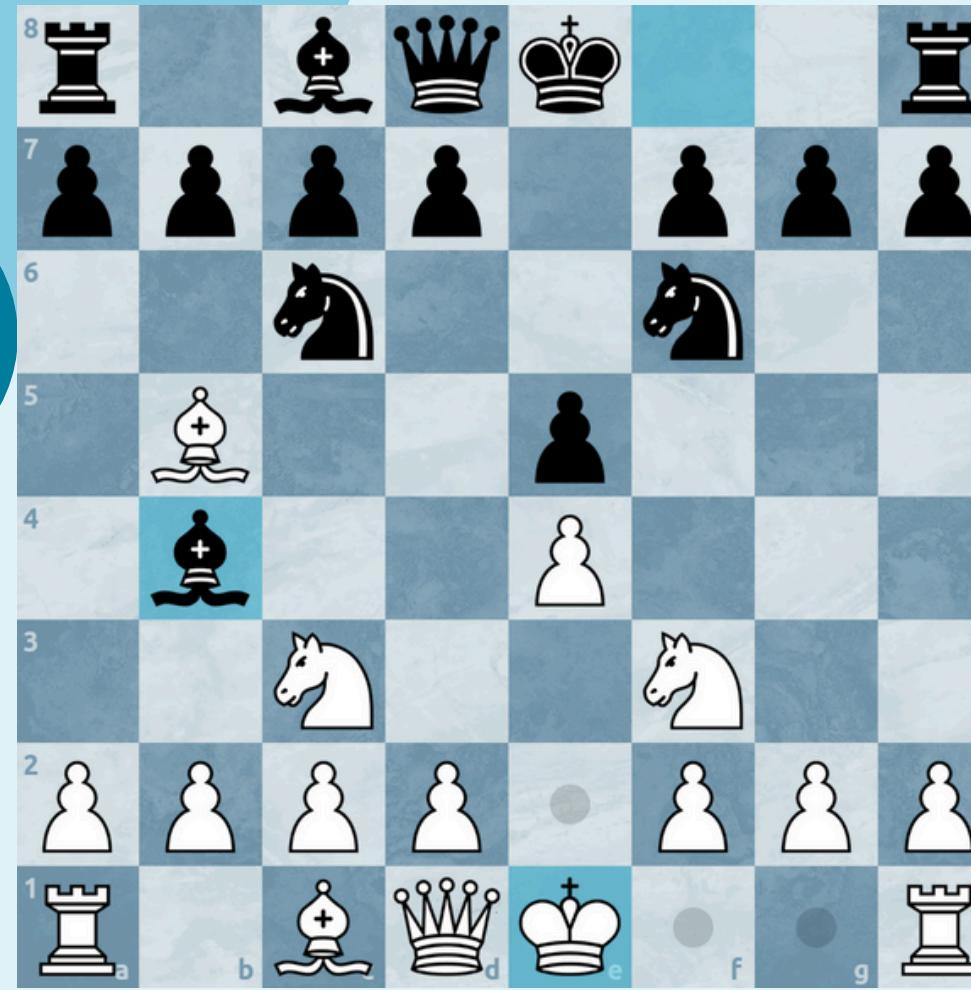


LE ROI



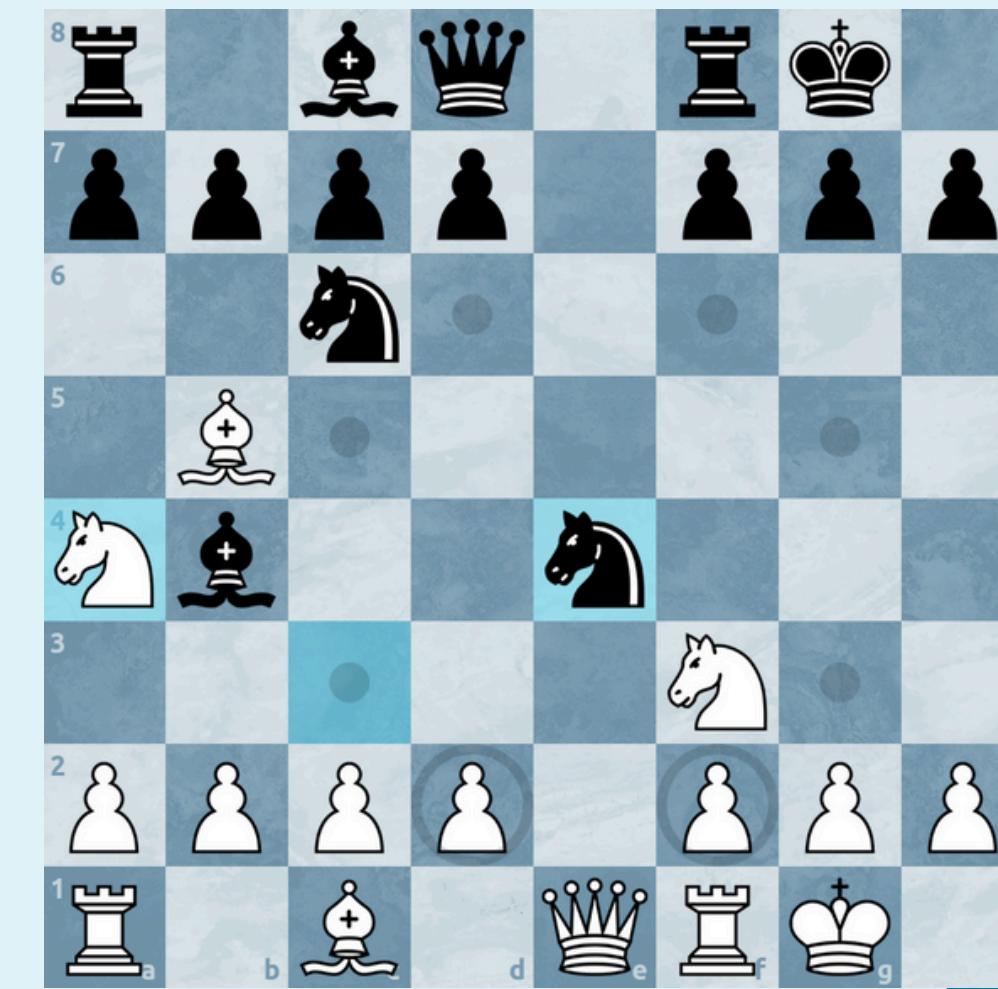
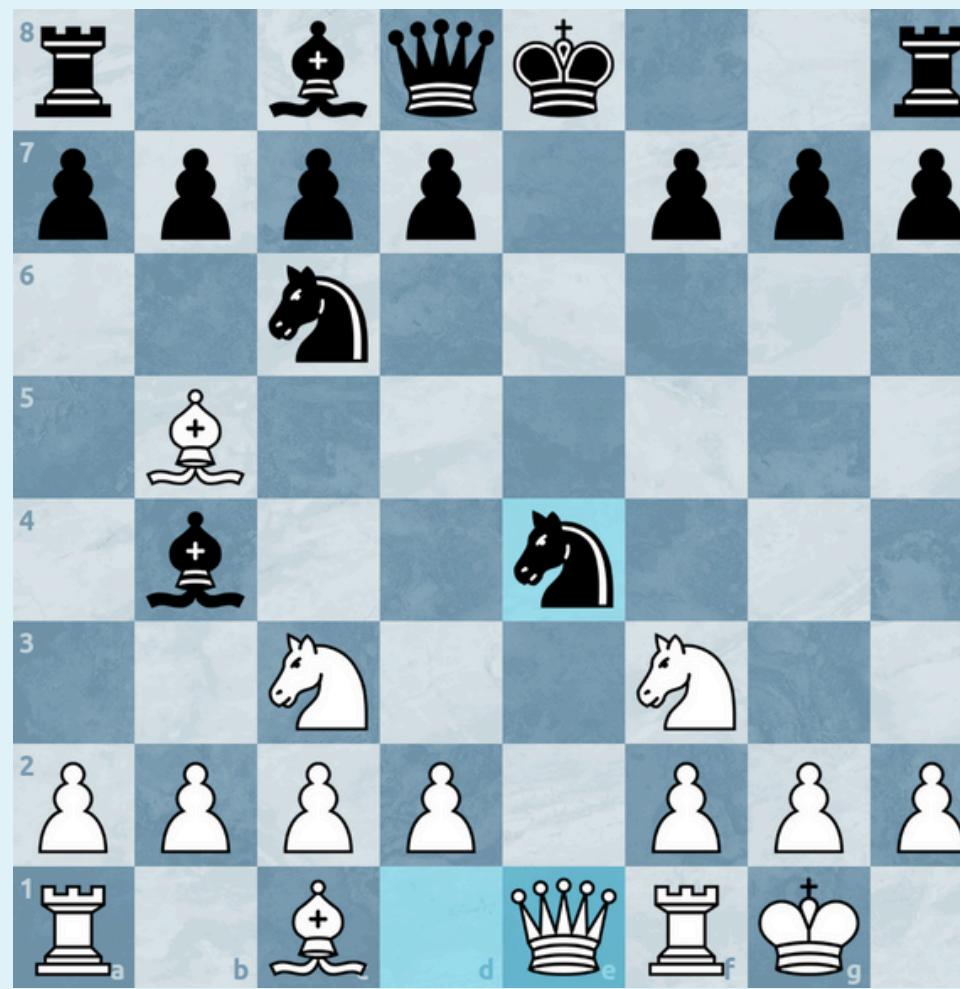
# LES RÈGLES SPÉCIFIQUES

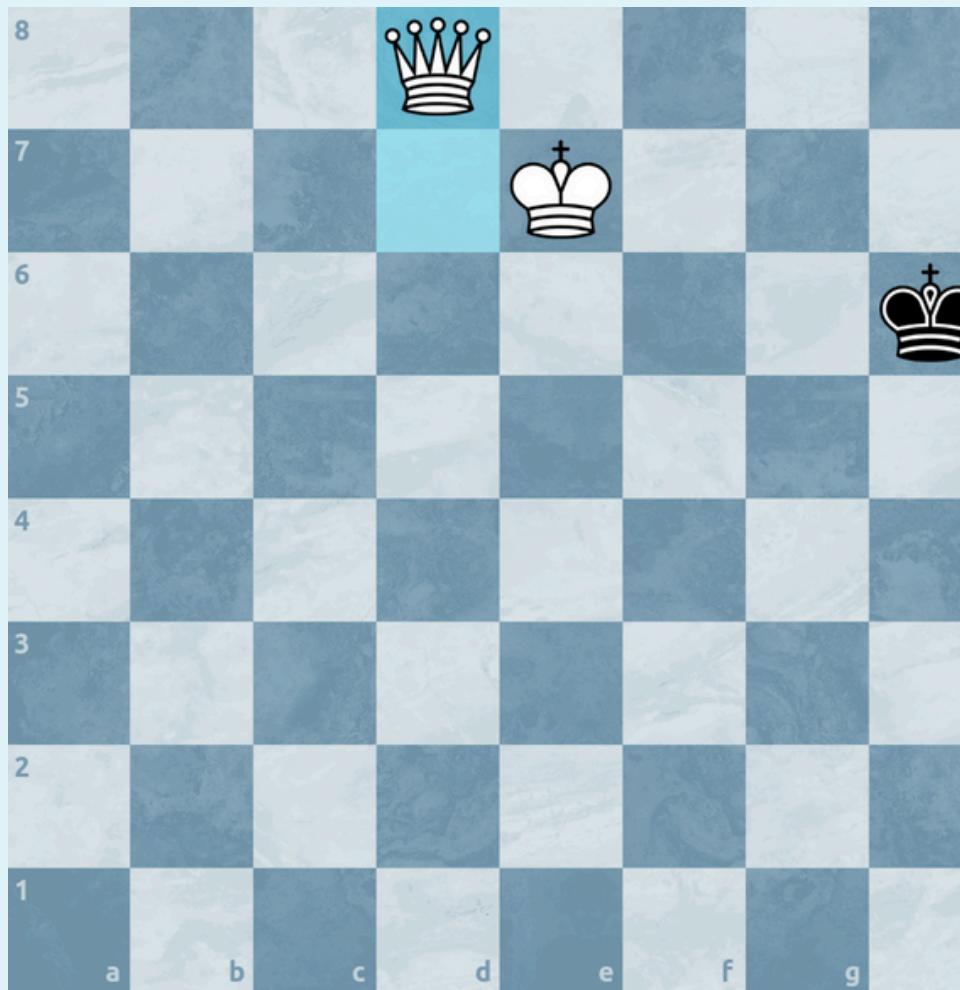
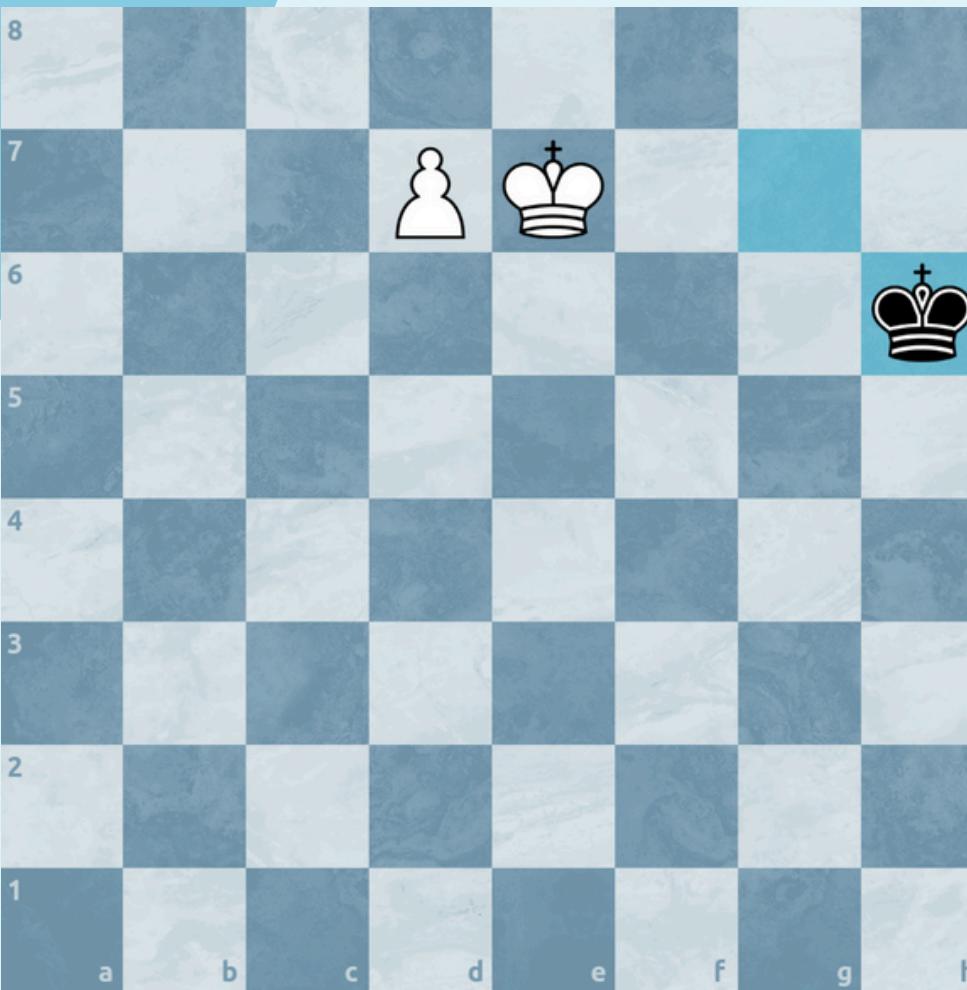
- Le roque
- le clouage
- en passant
- promotion du pion



## Le roque

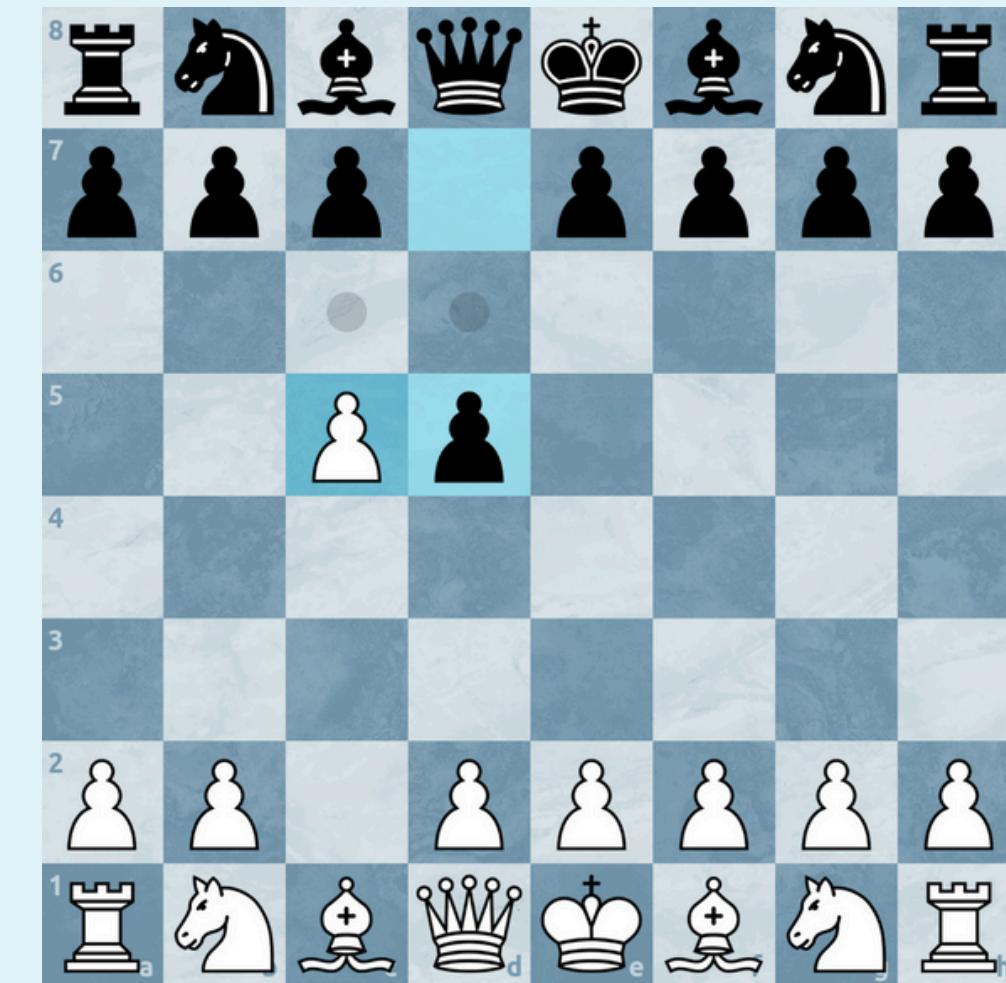
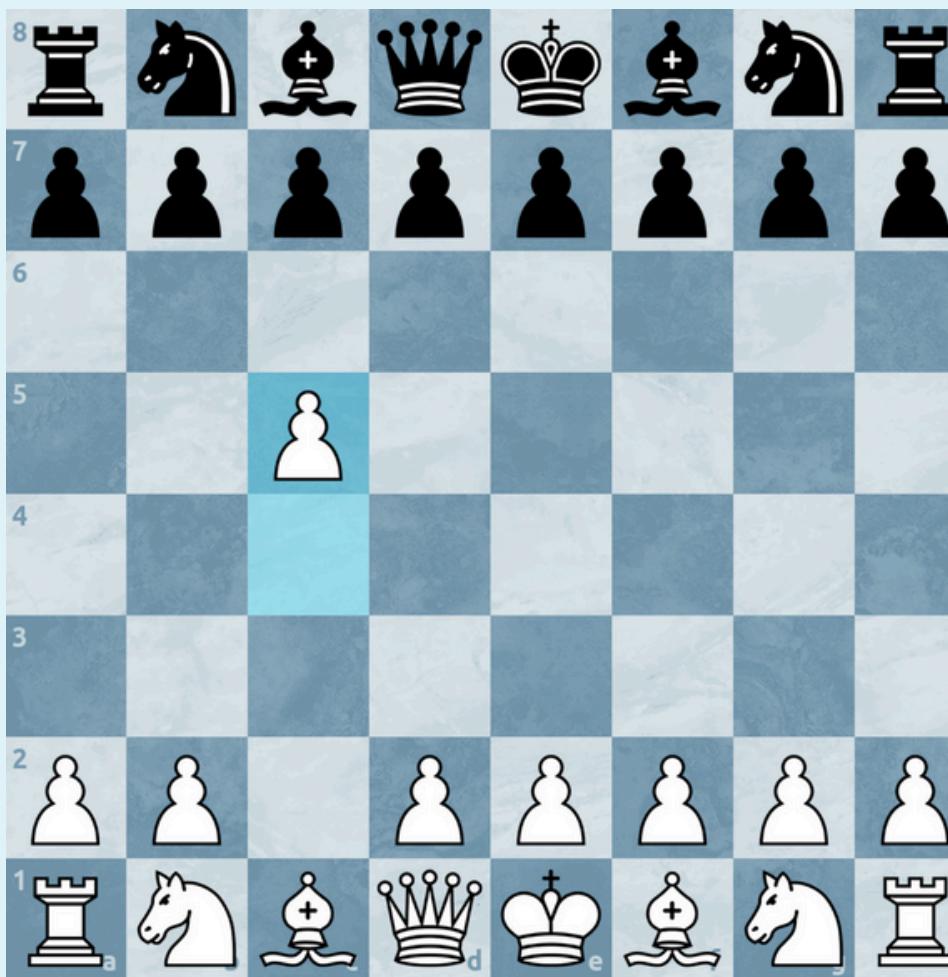
## Le clouage





## La promotion du pion

en passant



# FIN DE PARTIE

- Échec et mat
- pat ( nulle )
- la règle des 50 coups ( nulle )
- la triple répétition ( nulle )
- manque de matériel ( nulle )
- la nulle par accord mutuel
- dépassement de temps
- l'abandon

# PRÉCISIONS SUR LE MANQUE DE MATERIEL

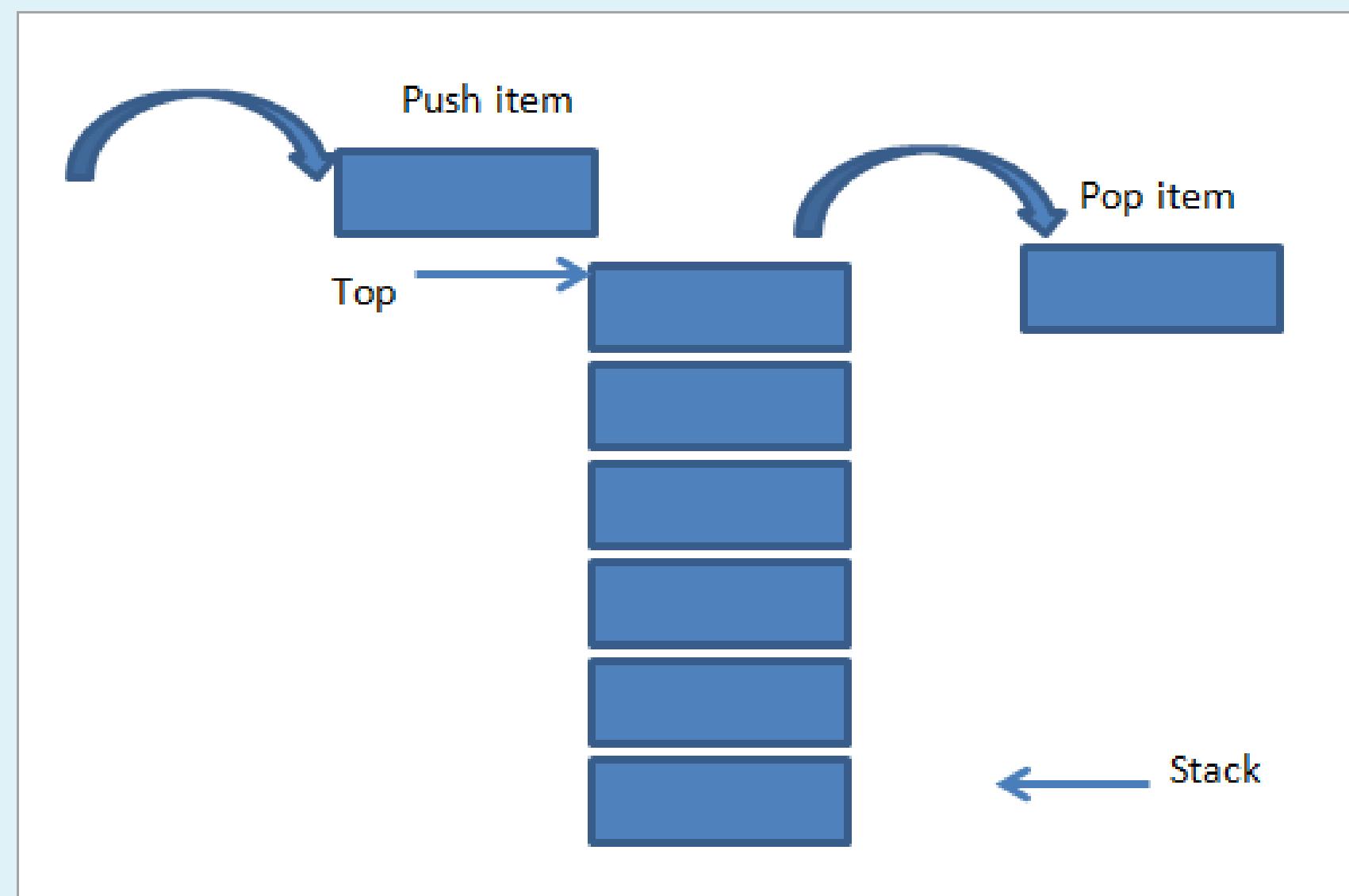
- Toutes les situations sont bien connues:
  - Roi contre roi
  - Roi et fou contre roi
  - Roi et cavalier contre roi
- Cas spécifiques:
  - Roi et fou contre Roi et fou (fous de même couleur)
  - Roi et deux cavaliers contre roi (pas de mat possible sans erreur)

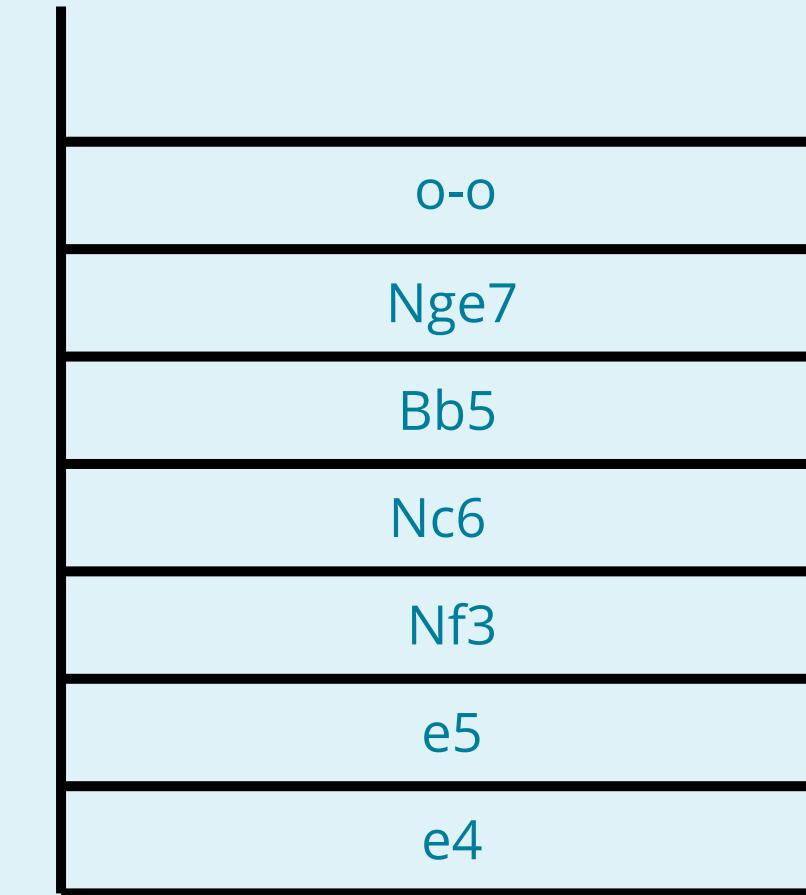
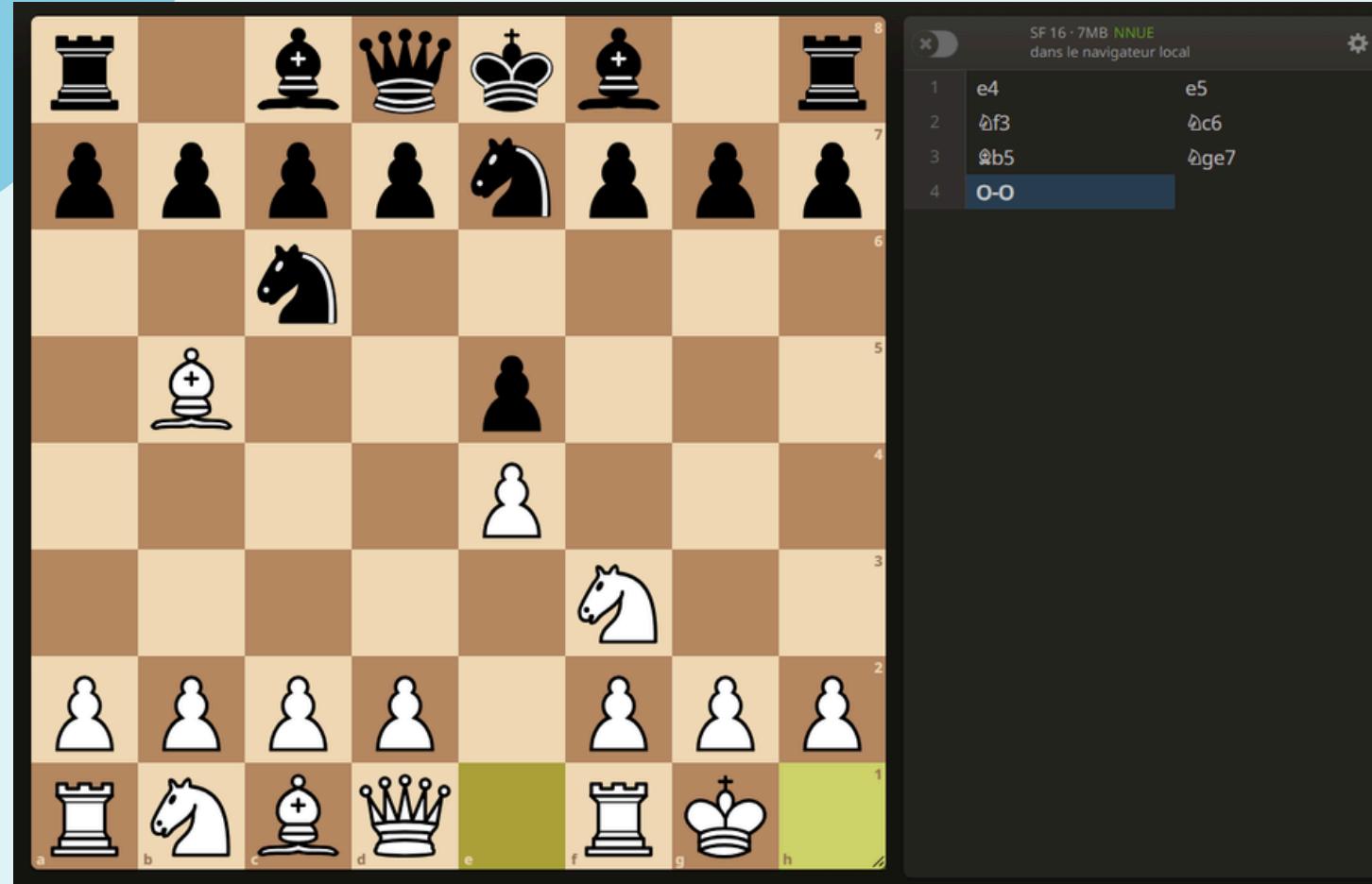
# **STRUCTURES DE DONNÉES**

Historique des coups  
Arbres VS Pile

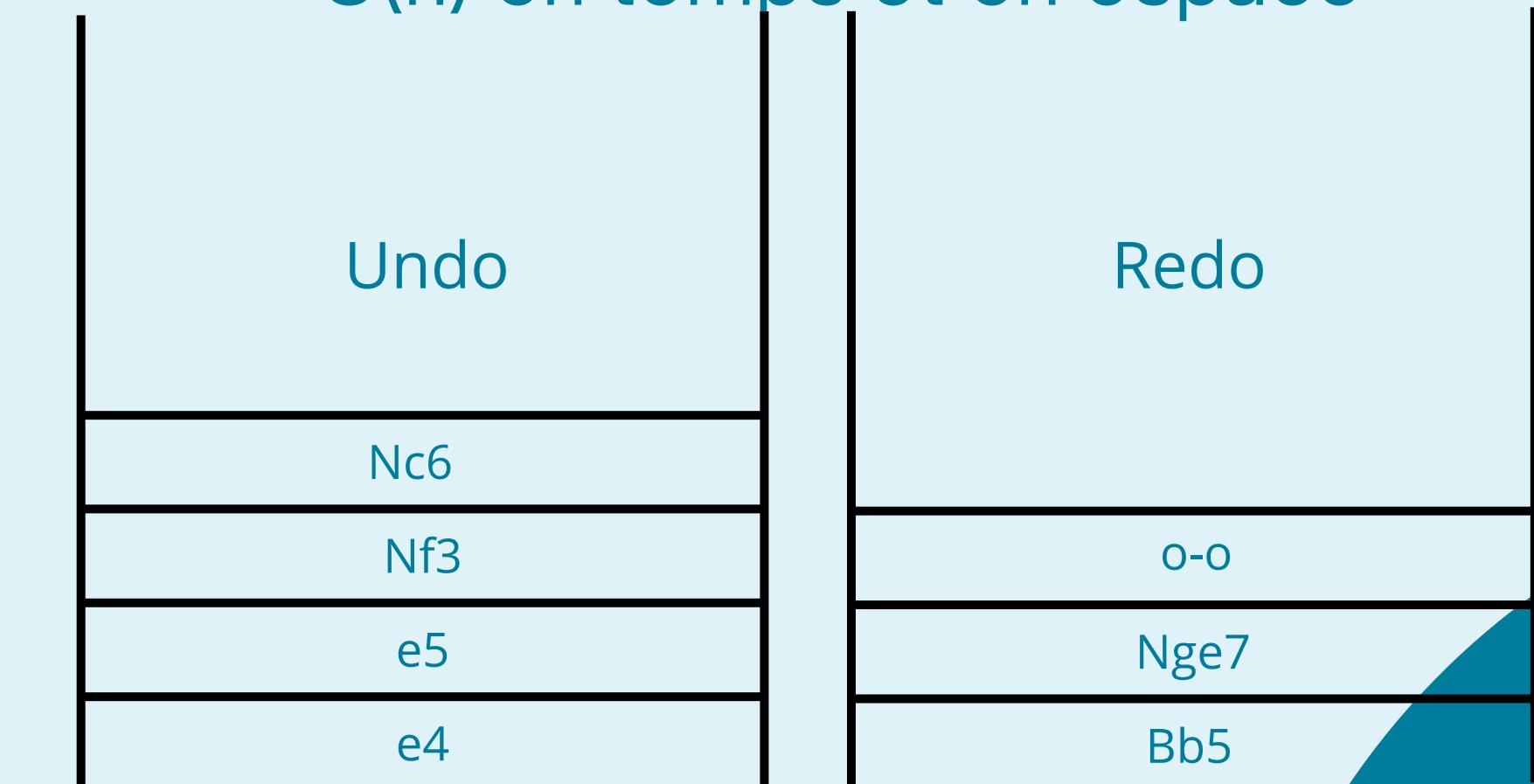
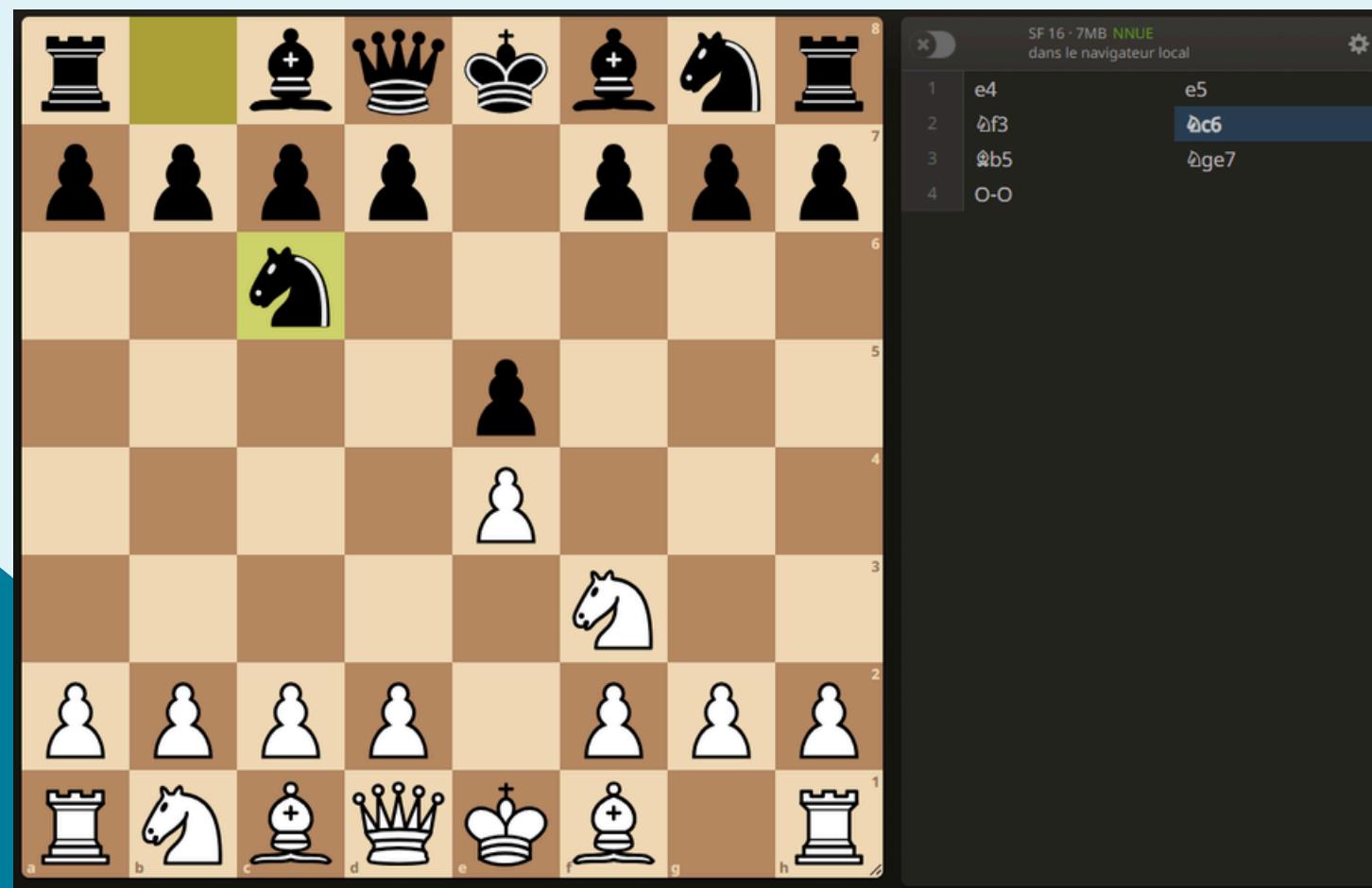
# STRUCTURES DE DONNÉES

- Pas de game review ou game analysis (pas de branchement ou de variations)
- Ecrasement de l'historique des coups



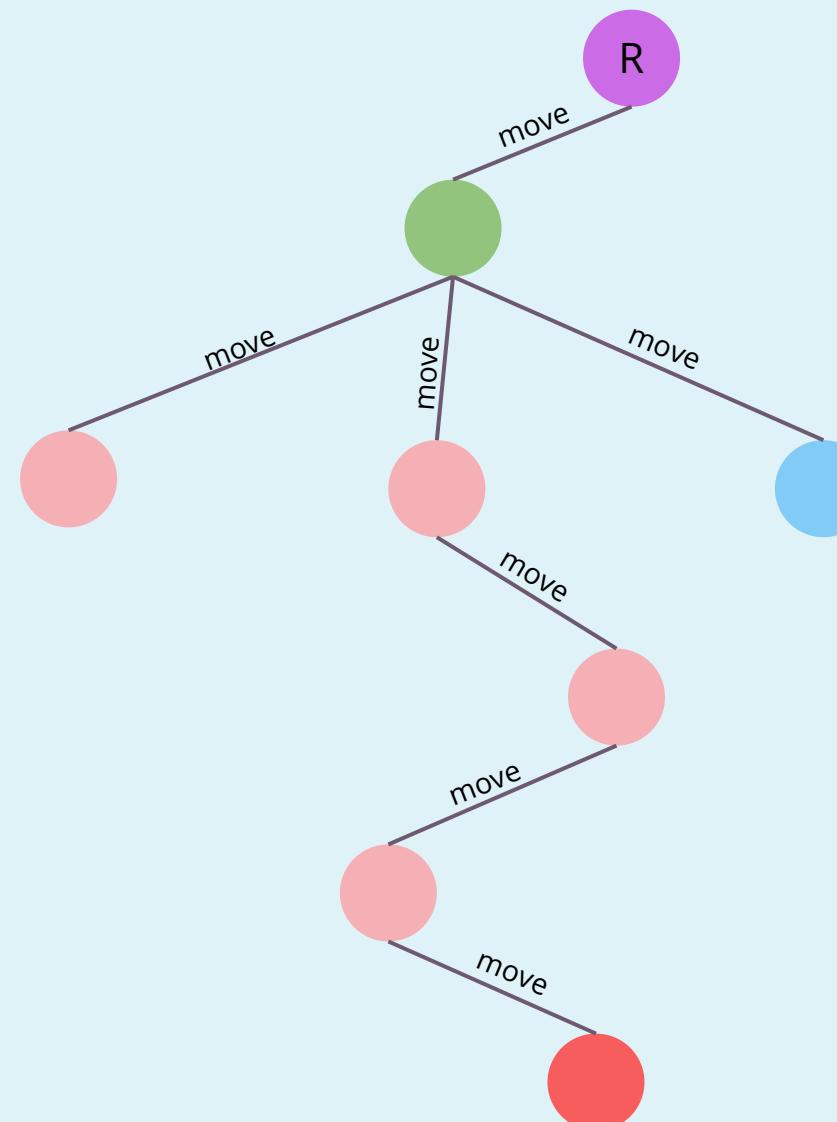


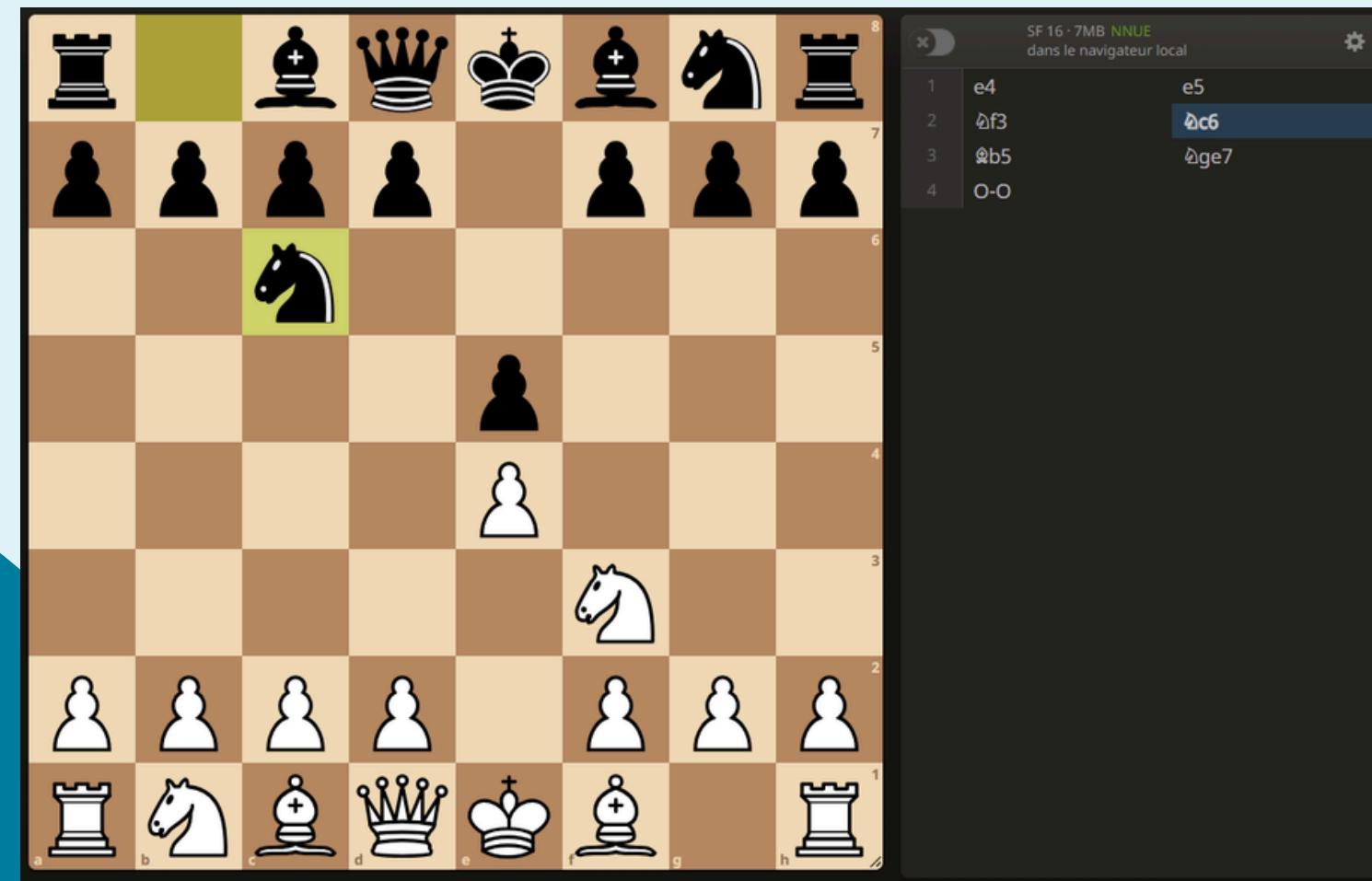
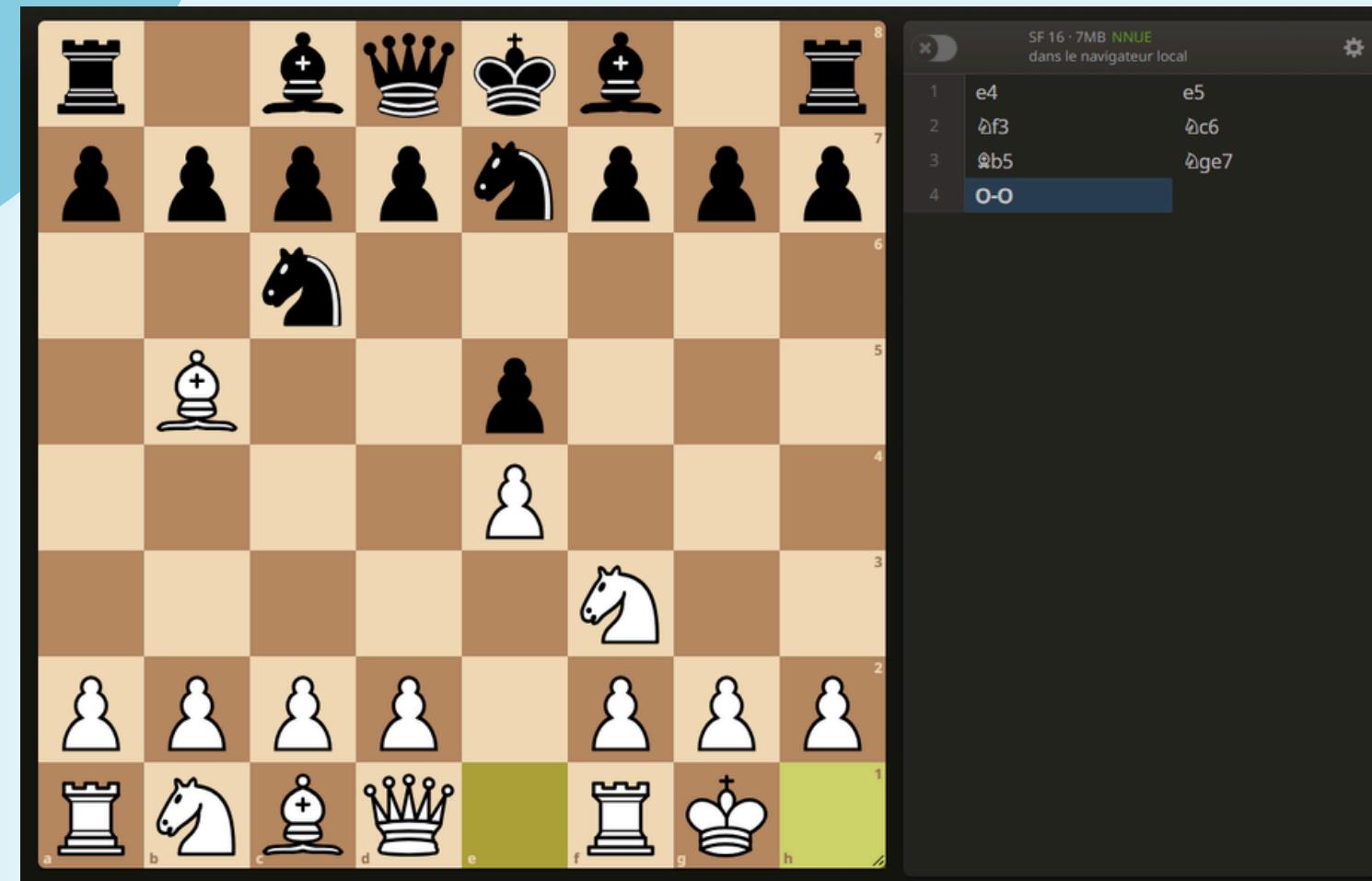
O( $n$ ) en temps et en espace



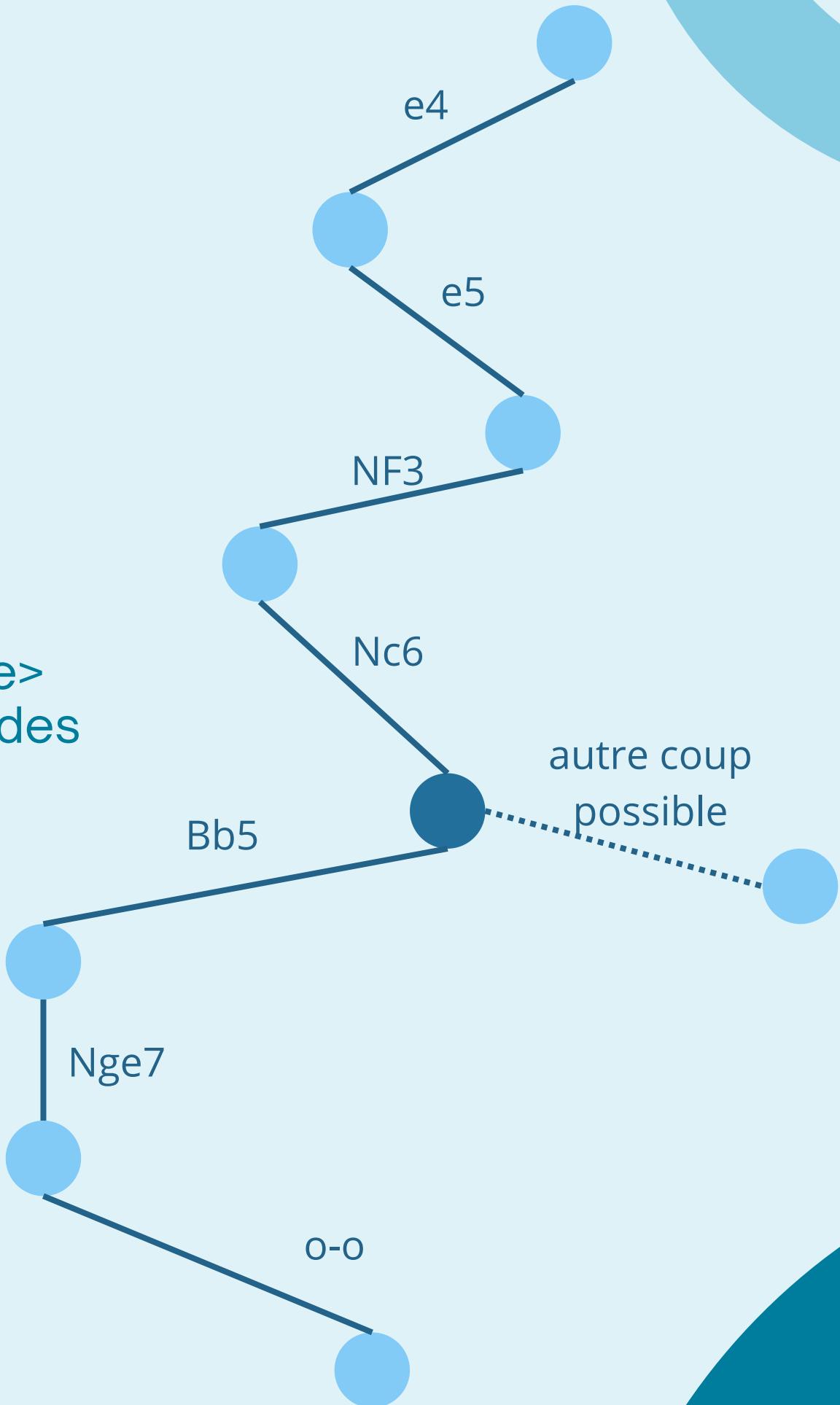
# STRUCTURES DE DONNÉES

- Option de game review ou game analysis (branchement et variations)
  - Possibilité de revenir à une position déjà jouée (même si on a joué par dessus)





En utilisant:  
Map<Integer, Node>  
On peut revenir à des  
positions en O(1)



# STRUCTURES DE DONNÉES

## Complexité en espace:

- En passant: 1 bit (flag) + 4 bits (quelle case des 16 est une cible d'en passant) => 5 bits
- Temps à la pendule : 54 bits (27 bits pour 86 400 000ms soit 24h)
- Droits aux roques : 4 bits
- Règle des 50 coups : 6 bits (entre 0 et 50)
- Nombre de “FullMoves” : 10 bits (1000 coups)
- 12 bitmaps pour les pièces

====>  $O(847\text{bits}^n)$  en espace

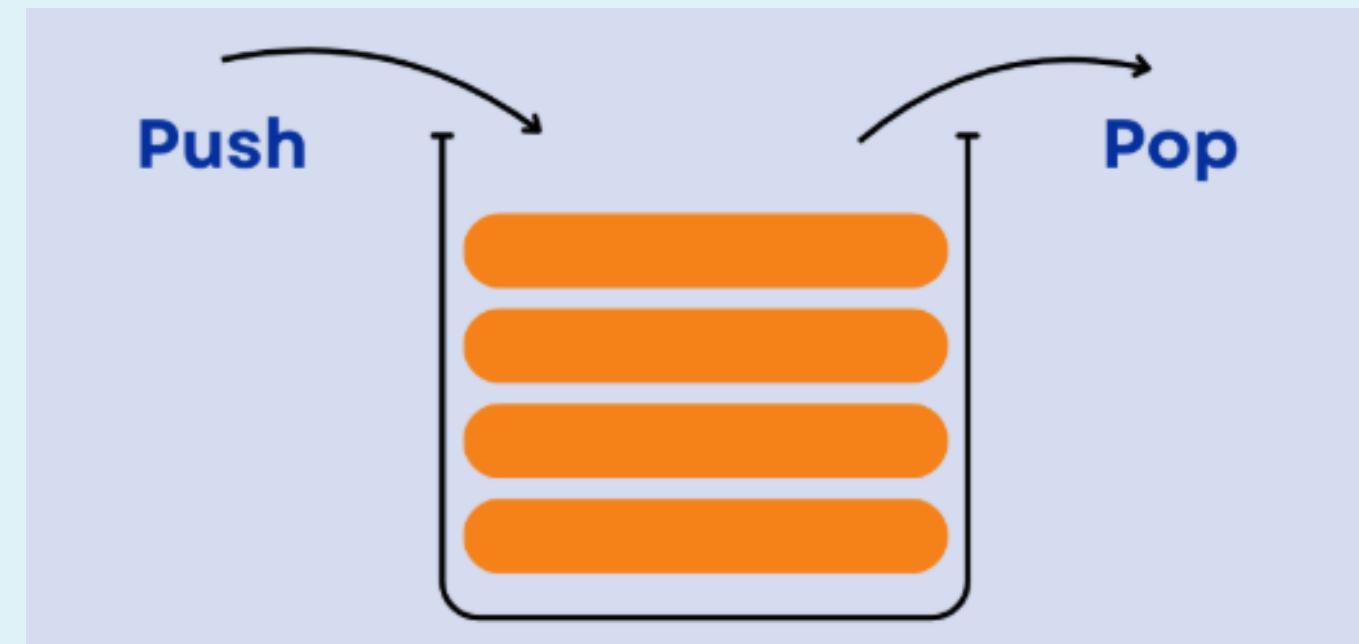
## Ordre d'idée:

- 500 coups --> 52KB

# STRUCTURES DE DONNÉES

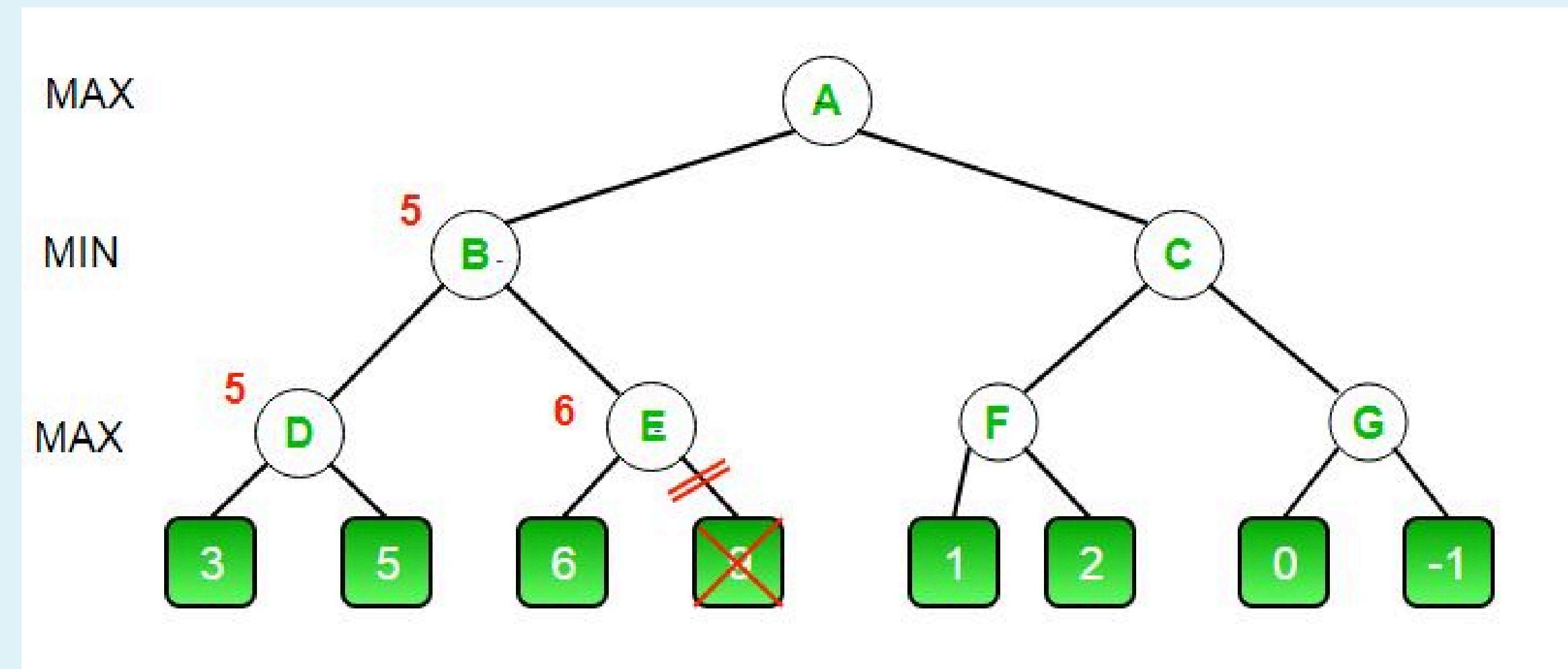
DÉCISION GLOBALE --> PILE

- Meilleur choix : arbre (utilisé dans des chess engines) mais compliqué à mettre en place
- Pile plus adaptée au problème posé



# STRUCTURES DE DONNÉES

Arbres pour les algorithmes d'IA (MinMax et Alpha-Beta)



# ALGORITHMES D'IA

- MinMax
- Alpha-Beta
- Monte Carlo Tree Search

# HEURISTIQUES

- Facteur de branchement aux échecs: ~35  
====> profondeur limitée

Idées de base:

- Matériel
- Contrôle du centre
- Sécurité du roi

Optimisations pour les ordinateurs modernes

- Forward pruning
- Reduction

# ÉCHIQUIER SOUS FORME DE BITMAPS (12)

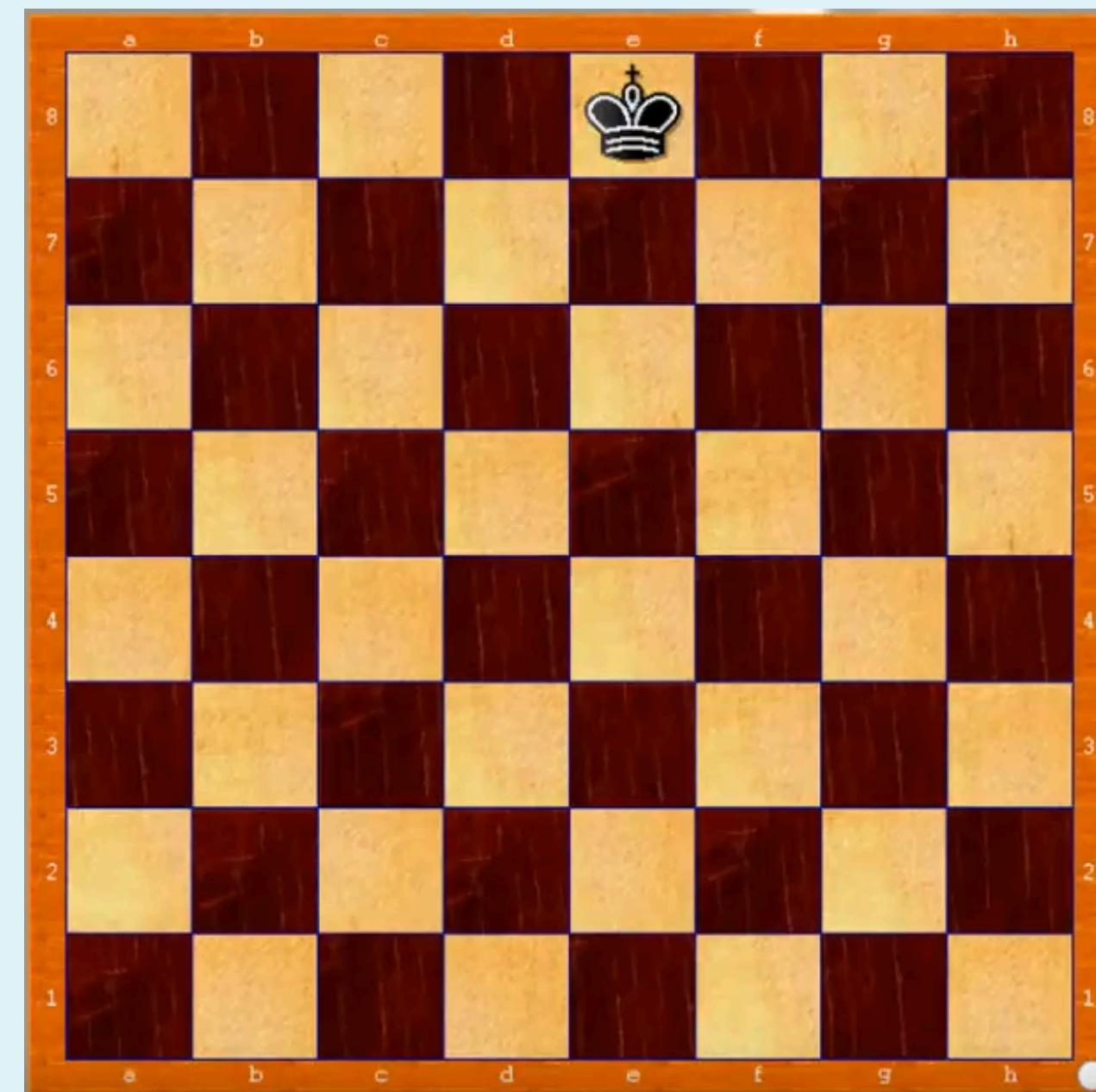
Bitboard: bien plus rapide que des tableaux,  
listes ou vecteurs.  
=> Stocker 12 entiers sur 64 bits (type primitif  
long en Java) pour chaque position.

# ÉCHIQUIER SOUS FORME DE BITMAPS (12)

On donne des indices à chaque case de l'échiquier:

8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	8	9	10	11	12	13	14	15
1	0	1	2	3	4	5	6	7
	a	b	c	d	e	f	g	h

# ÉCHIQUIER SOUS FORME DE BITMAPS (12)



# ÉCHIQUIER SOUS FORME DE BITMAPS (12)

# **ÉCHIQUIER SOUS FORME DE BITMAPS (12)**

Finalement, le bitboard nous apporte un gain de performance significatif en temps et en espace, rendant cette approche optimale.

# STRUCTURES DE DONNÉES

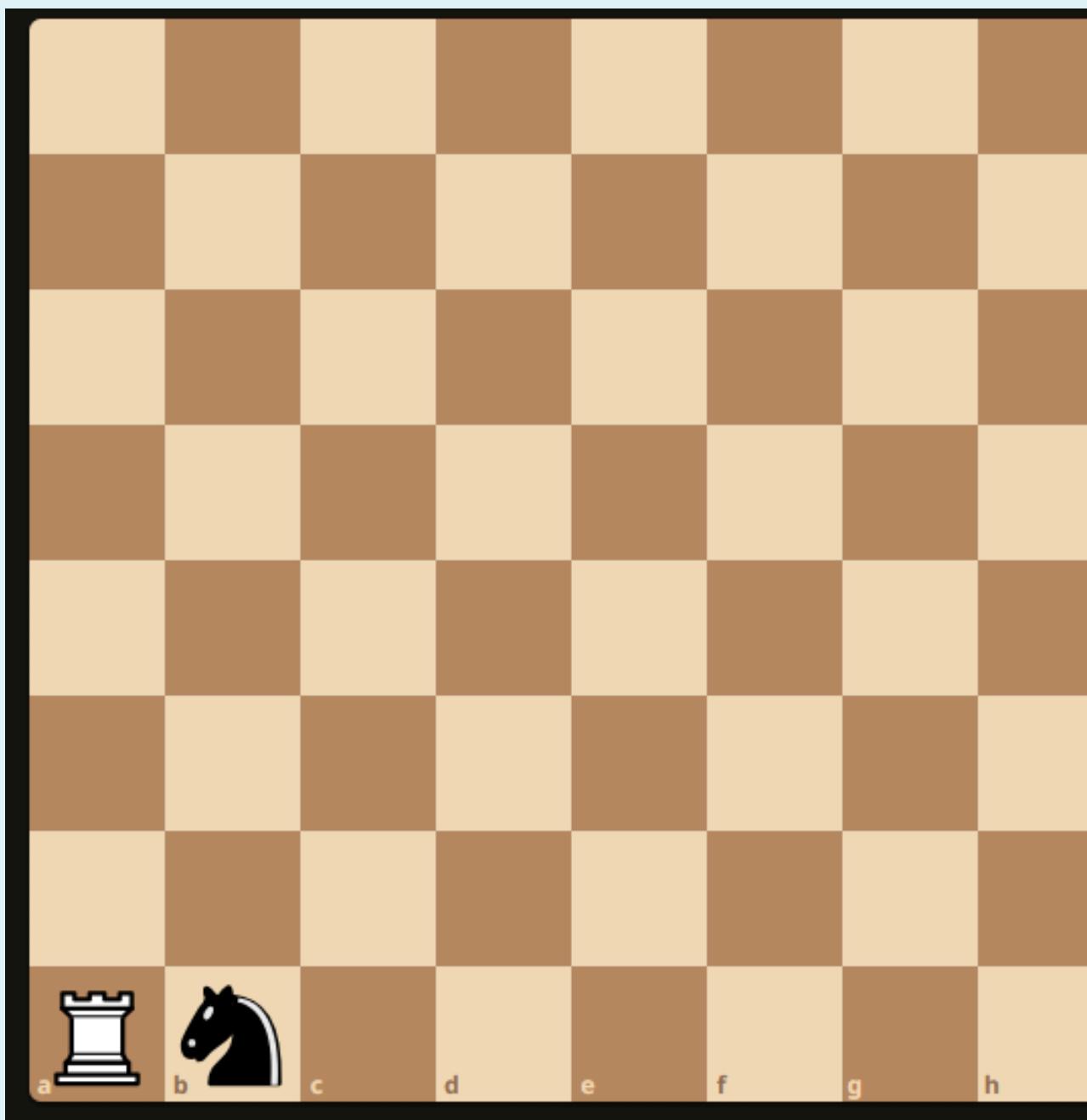
## Zobrist Hashing

5 Tables représentées en Arrays

- ZobristBoardTable[64][12]
- ZobristEnPassantTable[64]
- ZobristCastlingRightsTable[16]
- ZobristFiftyMoveRuleTable[51]
- ZobristFullMoveTable[1025]

# STRUCTURES DE DONNÉES

## Zobrist Hashing



long boardHash = zobrist[0][3] XOR zobrist[1][7]

chaque zobrist[i][j] est aléatoire

- Pion blanc -> 0
- Cavalier blanc -> 1
- Fou blanc -> 2
- Tour blanche -> 3
- Dame blanche -> 4
- Roi blanc -> 5
- Pion noir -> 6
- Cavalier noir -> 7
- Fou noir -> 8
- Tour noire -> 9
- Dame noire -> 10
- Roi noir -> 11

# STRUCTURES DE DONNÉES

## Zobrist Hashing

### HashMap pour la Threefold repetition rule

- On hash sur l'échiquier, on récupère ce hash et on le hash sur en passant, puis on récupère ce hash, etc.
- Chaque hash final est une clé et sa valeur associée représente le nombre de fois où une position a été atteinte
- S'il existe un hash H tel que  $H \geq 3$ , alors c'est une égalité

Peut aussi être utilisé pour l'IA -> se souvenir d'une position pour gagner du temps de calcul

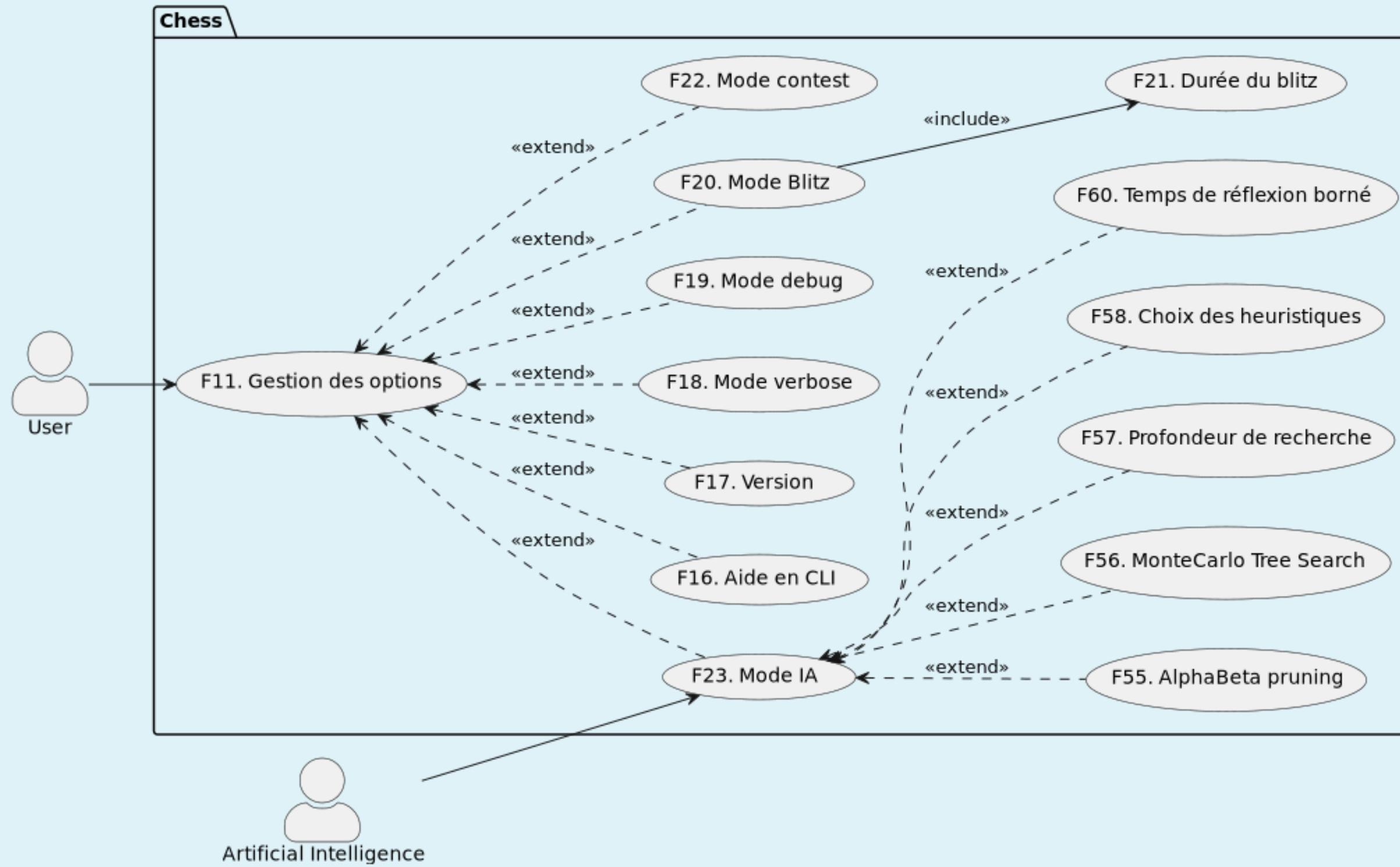
# BESOINS VISÉS

Nous avons pour objectif de remplir tous les besoins.

Classés selon leur priorité.

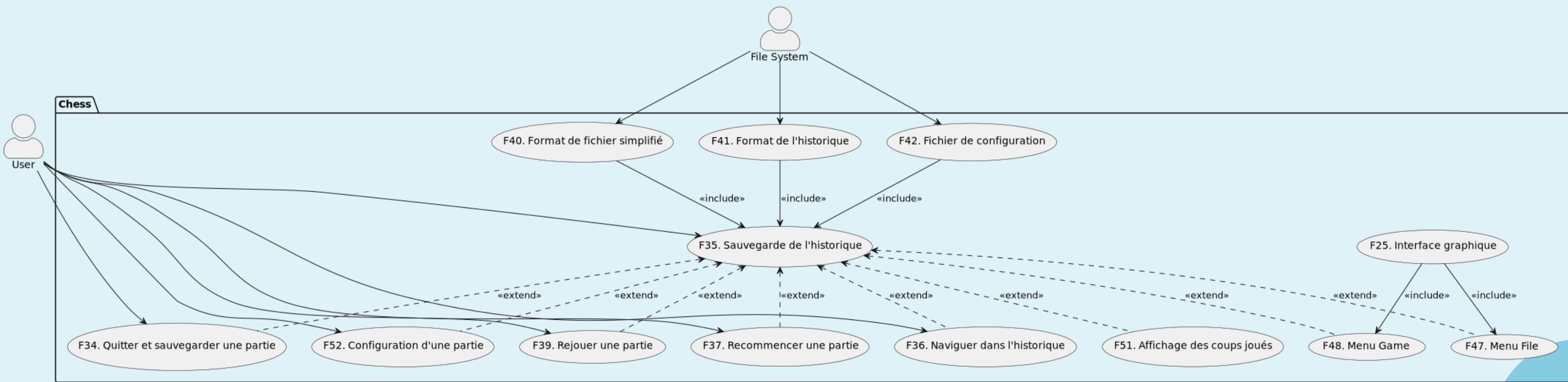
# DÉPENDANCES

## Gestion des options



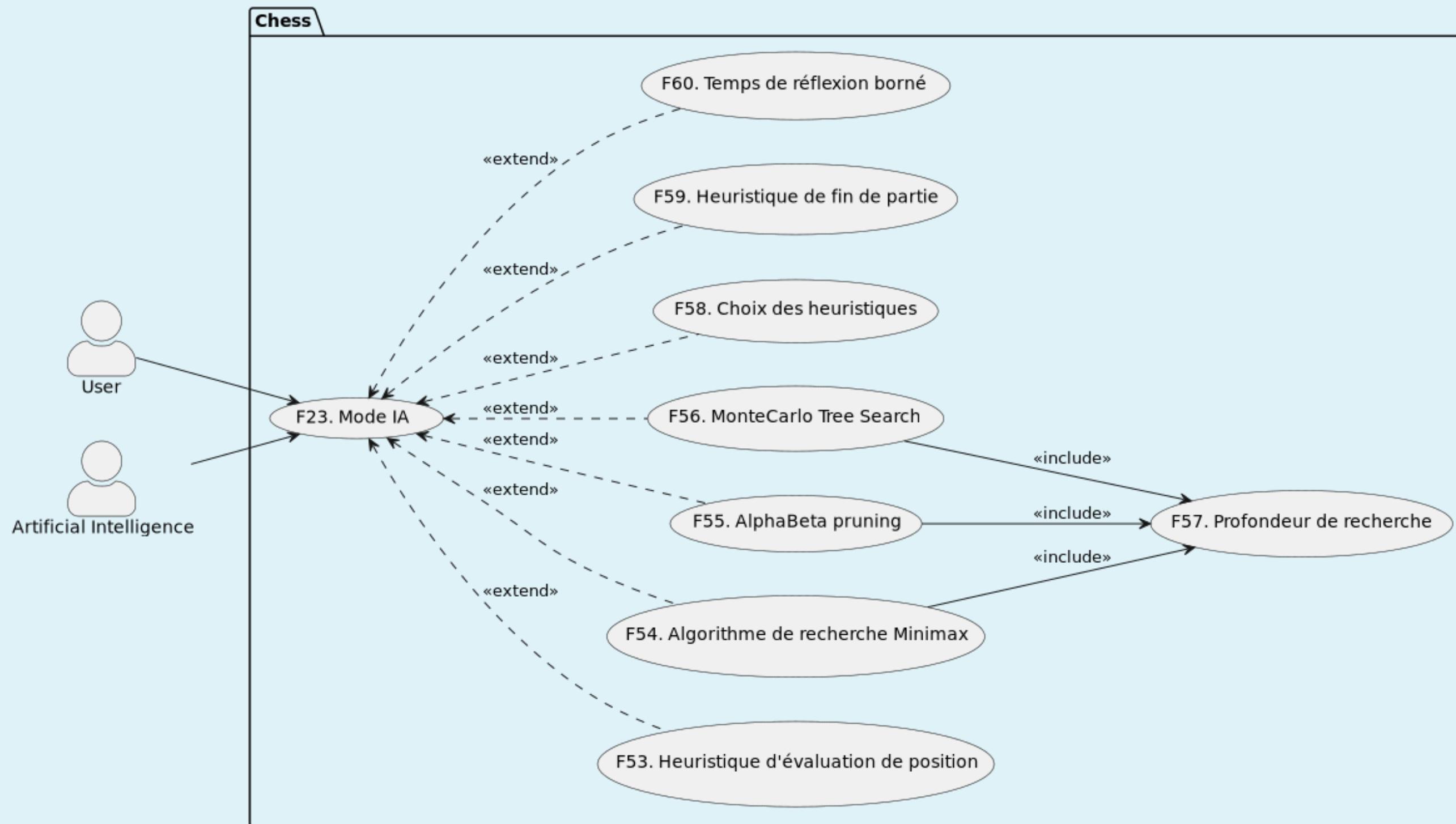
# DÉPENDANCES

## Sauvegarde de l'historique



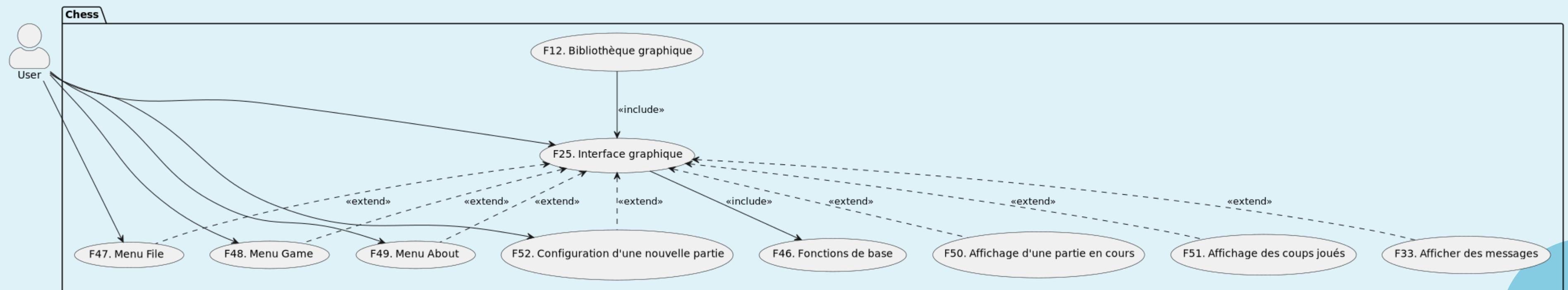
# DÉPENDANCES

Intelligence Artificielle



# DÉPENDANCES

## Interface Graphique



# MERCI

Avez vous des questions ?