BOUSSIDAN Aaron, PERRIN Jérémie et REY Simon

Projet de Programmation 2 - Ksparov 2.0

Année 2016-2017, Semestre 2

Depuis les début de l'informatique moderne et le développement grand public des micro-processeur de nombreuses applications ont vu le jour pour permettre à tout le monde de jouer aux échecs.

En effet qui ne rêve pas de pouvoir s'entraîner sans relâche pour atteindre le niveau d'un Bobby Fischer, ou de pouvoir revivre des parties de légende comme la finale de 1985 opposant Kasparov à Karpov.

Vous faire vivre cela, et bien plus encore, a été notre leitmotiv durant le développement de cette application. Voici la première version, la version 1.0, du meilleur jeu d'échec jamais créé, voici Ksparov !

Plus sérieusement ce document a pour vocation de décrire les choix d'implémentation fait et de présenter les fonctionnalités de notre application.

Le plus grand jeu de l'esprit jamais inventé, plus vous l'apprenez, plus vous y prenez du plaisir.
Garry Kasparov

1 L'organisation générale du code

Le code de notre projet a été divisé en plusieurs parties (chacune dans un fichier) qui correspondent chacune à une fonction particulière du code. On retrouve donc :

- **draw.scala** qui comprend tout ce qui concerne l'affichage de l'interface du jeu
- **parameter.scala** qui définit l'affichage, et la sauvegarde/chargement des paramètres depuis le fichier de paramètres
- **rules.scala** qui implémente toutes les règles du jeu pour chaque pièce
- **game.scala** qui gère le lancement du jeu et qui fait l'interface entre les règles et l'application
- **solve.scala** qui définit ce qui a trait à la résolution par l'intelligence artificielle
- **parse.scala** qui implémente la sauvegarde et le chargement des parties

Nous allons donc à présent étudier en détail chacun de ces fichiers.

Il est important de noter que cette documentation vient en complément du site suivant qui comporte toute l'api de notre projet :

`http://siolan.ddns.net/Ksparov/`

2 draw.scala ou l'interface utilisateur

Le code a ici été séparé en plusieurs objets, chaque objet comprend tout ce qui concerne l'affichage d'une fenêtre en particulier. Nous retrouvons donc un objet pour le menu initial, un pour le réglage des paramètres, un pour le plateau de jeu...

Les fonctionnalités de l'interface : L'interface a été créé de manière à pouvoir implémenter tout le projet, les trois parties, sans avoir à la modifier à chaque phase. Ainsi le jeu s'ouvre sur un menu général permettant pour le moment de lancer une partie, de modifier les paramètres et de quitter le jeu. Des boutons existent déjà pour le chargement d'une partie, les scores des joueurs... mais ne sont pas fonctionnels.

Le lancement d'une partie fait appel à un autre menu, celui qui permet de choisir le mode de jeu pour la partie : Humain/Humain, Humain/IA ou IA/IA. Après tout ces choix, l'échiquier apparaît enfin !

L'organisation générale d'un objet DrawQuelquechose : Chaque objet du fichier *draw.scala* a un nom construit comme *DrawUnechose* où *Unechose* est l'élément affiché par cet objet (*DrawParameters*, *DrawMenu*, *DrawBoard*...). Tous les objets sont construits de la même manière : tout les éléments d'une fenêtre sont des extensions de *GridPanel* qui sont ensuite organisés via un *BorderPanel* dans l'interface.

Pour avoir un code plus propre une classe commune à tous les objets a été créé : *BackgroundCase* qui étend un *GridPanel* et qui permet selon ses paramètres de dessiner un certain nombre de cases de fond. Avec l'utilisation des *Panel* et des images bufferisées, ces cases sont extensibles, ce qui permet à l'utilisateur d'adapter la taille de la fenêtre tout en gardant un fond uni.

Le plateau de jeu : Le plateau de jeu est, heureusement, la fenêtre la plus complète, on y retrouve plusieurs éléments.

Tout d'abord le plateau, qui est constitué de case sur lesquelles les pièces se rajoutent via une icône. La particularité étant que le plateau ne dessine pas les pièces lorsque l'on en créé une nouvelle instance, il faut appeler *DrawActions.draw_board* pour cela.

Le nombre de plateau dessiné dépend d'une variable de la partie en cours *nb_grid*, dans tous lors de la construction du plateau, on vérifie sa valeur et on y adapte l'affichage. Ceci permet de garder un code très modulaire pour l'implémentation des échecs d'Alice.

Sur chaque coté du plateau on retrouve un compte des pièces mortes pour chaque joueur. Ceci est fait par le dessin des pièces via *DrawActions.draw_board* pour lequel les pièces ayant des positions négatives sont mortes.

Ce sont ces cases de pièces mortes qui serviront dans le cas d'une promotion. En effet ce sont des buttons qui sont par défaut désactivés mais qui lorsqu'une promotion doit être faite s'activent pour que le joueur sélectionne la pièce de la promotion. En respect des règles des échecs, toutes les cases s'activent sauf celle du pion.

Sous l'échiquier on retrouve un encart qui permet d'afficher des messages, c'est ici que les échecs et échecs et mat s'affichent.

C'est aussi sous l'échiquier que l'on retrouve les horloges, il y en a une par joueur et affichent toutes les informations de la période. Ainsi, l'horloge affiche :

— Le temps restant pour la période
— Le nombre de coup restant à jouer, ou 0 si tous les coups ont été joué
— L'incrément de la période s'il y en a un

Si à un moment de la partie le temps restant du joueur passe sous les 5 secondes et qu'il n'a pas fait le nombre de mouvement nécessaire, l'horloge s'affiche en rouge pour l'indiquer au joueur.

Dans le cas d'une partie chargée, des boutons pour prendre la main sur la partie sont présents, ils sont décrit dans la section 6 qui traite du chargement de la sauvegarde des parties.

Finalement on retrouve en haut de la fenêtre un menu rapide qui permet de sauvegarder, de recommencer une partie, de revenir au menu ou de quitter Ksparov.

Les actions dynamiques de l'interface : Le plateau de jeu est plus ou moins dynamique pour afin en temps réel les mouvements des pièces, les messages et le nombre de pièce morte. Tout cela est défini dans l'objet *DrawActions*.

L'affichage des pièces passe par l'appel de la méthode *DrawActions.draw_board*. Cette méthode prend chaque pièce et définit l'icône correspondante sur la case des coordonnées de la pièces. Les pièces ayant des coordonnées négatives sont des pièces mortes et donc *draw_board* actualise le compte des pièces mortes à chaque fois qu'il en trouve une. La méthode fini par une actualisation de la fenêtre ce qui permet d'afficher les pièces sur le plateau et le compte des pièces mortes.

A chaque fin de mouvement, la méthode *Ksparov.play_move*, qui applique le déplacement, vérifie si un évènement s'est produit suite à ce mouvement (échec, échec et mat ou pat). Si elle détecte un évènement elle affiche le message correspondant, sinon et affiche un message indiquant que la main est pour le joueur suivant.

Dans le cas d'une promotion on fait appel à la méthode *enable_promotion* qui active les boutons des pièces mortes de manière à ce que l'utilisateur puisse choisir la pièce de la promotion. A la fin de la promotion, on appel *disable_promotion* pour désactiver les buttons.

Pour l'affichage de toutes informations du jeu la méthode *DrawActions.draw_game_message* est appelée. Elle prend en argument le type de message a afficher et le joueur auquel il s'applique. C'est ainsi que s'affiche les messages de mate, de partie de nulle. Lors d'une partie chargée, c'est aussi ces méthodes qui permet d'afficher les commentaires présents dans le PGN, comme par exemple les !, !?, ?? ... De plus les tags des noms des joueurs présent dans un PGN sont lus pour actualiser l'affichage correctement.

3 parameter.scala ou définir son propre Ksparov

Ce fichier permet de personnaliser l'application Ksparov. On y retrouve trois objets : *Display* qui définit les paramètres d'affichage, *Parameter* pour les autres paramètres et enfin l'objet *DrawParameters* qui permet d'afficher le menu de choix des paramètres.

L'objet *Display* : Cet objet comprend toutes les variables ajustant l'affichage, c'est-à-dire les chemins vers les ressources, les dimensions des composants, les dimensions du texte, les couleurs du texte...

On y retrouve aussi la méthode *apply_resolution* qui permet d'ajuster l'affichage selon la résolution de l'écran. En effet trois jeux de ressource sont disponibles, on les retrouve dans les dossiers *Min/*, *Max/* et *Jeroboam/* dans les ressources. Ainsi cette méthode récupère la résolution utilisée par le joueur et choisit le jeu de ressource en fonction. Les ressources dans *Min/* sont utilisées si la largeur de la résolution est inférieure à 1000, entre 1000 et 1100, on utilise le jeu présent dans *Max/*, pour les autres utilisateurs on utilise le jeu *Jeroboam/*.

***Parameter* et le fichier *src/main/resources/Parameters* :** L'objet *Parameter* définit les autres variables de paramétrisation de l'application ainsi que les méthodes pour lire et appliquer les paramètres dans le fichier de paramètre.

Les autres variables de paramétrisations sont aux nombre de trois : *speed_ai* qui définit le temps de l'attente de l'IA avant qu'elle ne joue son mouvement, *nb_alice_board* qui indique combien de plateau sont utilisés dans la variante des jeux d'Alice et *nb_case_board* qui donne le nombre de case du plateau.

Plus importantes sont les méthodes *apply* et *write* pour conserver les paramètres. La méthode *apply* est appelée au lancement de l'application, elle cherche à ouvrir le fichier *src/main/resources/Parameters* dans lequel sont enregistrés les paramètres. Si elle réussit à trouver et à ouvrir un tel fichier, elle le lit ligne après après ligne pour ajuster les variables courantes aux paramètres sauvegardés. Dans le cas où le fichier est manquant, ou bien qu'une exception est lancée à un moment de la lecture de ce dernier, l'application est ouverte avec les valeurs par défaut et ces valeurs sont enregistrés dans un nouveau fichier qui vient écrasé le précédent.

De la même manière, *write* prend les valeurs des variables à sauvegarder et les enregistre dans le fichier de sauvegarde des paramètres. Pour ces deux méthodes les variables enregistrées sont :

— La texture de fond
— Le jeu de pièce utilisé
— La vitesse de l'IA
— Le nombre de plateau pour les échecs d'Alice
— Le nombre de période d'horloge
— Le descriptif de chaque période

Pour enregistrer chaque période de façon détaillé, une méthode est appelée pour transformer le tableau en une chaîne de caractères. Une autre méthode est appelée pour lire de fichier de paramètre pour parser les périodes.

Afficher le menu de sélection des paramètres : Le dernier objet de ce fichier permet d'afficher le menu des paramètres. Étant donné le nombre de paramètre enregistré, c'est un menu un petit peu plus complexe que les précédents. Il est organisé en trois sous-menus qui sont : affichage, jouabilité ou pour l'horloge.

Le menu pour régler l'affichage permet de sélectionner le fond utilisé pour l'application et le jeu de pièce. Celui pour la jouabilité permet d'ajuster la vitesse de l'IA et le nombre de plateau pour les parties d'Alice. Enfin le menu pour l'horloge permet de sélectionner le nombre de périodes, l'affichage dynamique permet de renseigner les différents champs d'une période : temps, nombre de mouvement et incrément.

4 rules.scala ou l'implémentation des règles des échecs

Les fonctionnalités : Toutes les règles de base des échecs sont présentes, dont en particulier la prise en passant, la promotion, et les règles conduisant à une égalité.

Structure de rules : Toutes les pièces héritent d'une classe abstraite *Piece*, qui contient principalement une méthode utile pour le reste du jeu : la fonction *move*, qui effectue un mouvement.

Chaque pièce particulière (pion, tours,..) hérite ensuite de *Piece* et se distingue par un "pattern" particulier, qui traduit l'ensemble de ses mouvements possibles.

Des méthodes plus générales sont présentes dans *Aux*, et ce qui concerne la vérification de l'échec et mat appartient à *Checkmate*.

La fonction *move*, et ses fonctions auxiliaires : La fonction *move* sert à vérifier si un mouvement est valable, à l'appliquer si cela est le cas, et à modifier la plateau en conséquence (suppression de pièces, échec...).

Elle appelle pour cela une fonction *pre_move*, qui vérifie la faisabilité du mouvement sans modifier le plateau (avoir une telle fonction se révèle utile pour, par exemple, afficher l'ensemble des cases atteignables par une pièce). *pre_move* vérifie plusieurs critères : si le mouvement est conforme aux déplacements de la pièce et à quelques règles basiques (ceci est effectué par *Aux.checks_pre_move*), si le chemin que veut suivre la pièce est libre et s'il y a une pièce adverse à l'arrivée (effectué par *clear_path*), et si ce mouvement met un ou deux rois en échec (effectué par *check_check*).

Spécificité de certaines pièces : Le pion, qui est la seule pièce à avoir un déplacement non symétrique, possède un pattern un peu différent, qui est obligé de référer au joueur le contrôlant. Le cavalier, quant à lui, peut passer au dessus des autres pièces, et nécessite donc une modification de *clear_path*. Le roi, enfin, possède en attribut la liste des pièces le mettant en échec, des booléens indiquant s'il est en échec et s'il a bougé, et a une version de *move* différente pour gérer le cas du roque.

***check_mate*, vérification de l'échec et mat :** Plutôt que de vérifier si tous les mouvements possibles laissent le roi en échec (ce qui aurait une complexité assez grande), la fonction *Checkmate.check_mate* vérifie si le roi peut bouger, et si une autre pièce peut s'interposer sur le chemin de son attaquant, voir le prendre.

Les conditions d'égalité : Plusieurs cas de partie nulle sont testés à chaque tour dans la fonction *check_game_status* :

- L'impossibilité de mater : A chaque tour, le programme vérifie s'il reste assez de pièces pour aboutir à un mat. Les configurations nulles sont : Roi contre Roi, Roi et (cavalier ou fou) contre Roi, et Roi et fou contre Roi et fou de la même couleur. Ce travail est effectué par *check_nulle_finale* dans rules.
- La règle des 50 coups : S'il y a 50 coups sans prise ni mouvement de pion, la partie est nulle. Ceci est simplement fait en incrémentant le compteur _itnb_boring_moves à chacun de ces coups, et en vérifiant qu'il ne dépasse pas 50.
- La triple répétition de position : Si 3 positions identiques se répètent lors d'une partie, celle-ci est nulle. Pour éviter de stocker tous les plateaux, on convertit chaque plateau en chaîne de caractère, qu'on hash ensuite grâce à la fonction de hashage intégrée de Scala *string.hashCode()*. Ceci est effectué par *array_to_hashed_string*. Il n'y a après plus qu'à vérifier s'il n'y a pas de triplicata.
- La vérification du pat : La vérification du pat se fait par une méthode définit dans la classe joueur. Pour un joueur humain et consiste à faire la somme du nombre de mouvements possibles pour chaque pièce. Si cette somme fait 0, le joueur est en pat.
  Pour une AI, le pat est déjà implémenté dans sa fonction de mouvement, ainsi la méthode *check_pat* pour une IA consiste juste à renvoyer le booléen *pat* définit par la méthode *get_move*.

Les échecs d'Alice : Pour la variante nous avons implémenté les échecs d'Alice, une variante utilisant deux plateaux pour jouer. Pour plus de détails :
La partie commence avec un plateau rempli comme aux échecs conventionnels, et un plateau vide. A tout moment du jeu, les mouvements valides sont les mouvements qui sont valides sur la plateau de la pièce, et dont la case d'arrivé est inoccupée sur le second plateau. Après un mouvement, la pièce bougée change de plateau, et se retrouve à la case correspondante sur l'autre plateau.

Notez donc en particulier qu'on ne peut manger qu'une pièce qui se trouve sur le même plateau que celui d'origine. Les règles de l'échec, du mat, et de la partie nulle sont les mêmes pour les échecs d'Alice.

Le roque et la prise en passant ne sont toutefois pas présents dans les échecs d'Alice, faute de règle officielle (et la prise en passant perd de son sens étant donné l'alternance des plateaux).

On notera également que, comme le code était facilement modulable, la variante des échecs d'Alice est aussi bien jouable à N plateaux qu'à 2. Le principe reste le même, les pièces cyclent juste entre les différents plateaux. Cette extension restant anecdotique, l'interface graphique n'est pas prévue pour être très fonctionnelle au-delà de 3 plateaux.

Au niveau du code, cette variante demande peu de changements. Il y en a cependant quelques-uns :
— Chaque pièce a dorénavant un attribut grid, qui dénote le plateau sur lequel elle est. Associés à cet attribut sont les méthodes *next_grid* et *previous_grid* qui avance ou recule la pièce d'un plateau
— La fonction *mirror_free*, qui vérifie qu'une case est libre sur le plateau suivant
— *clear_path* a été modifiée de façon à inclure *mirror_free* dans ses tests préliminaires, ce qui permet d'assurer la nouvelle règle.
— De même, certains override ont également du être modifiés pour gérer le cas des échecs d'Alice
En définitive, la variante n'impose pas une trop grande contrainte sur le code, et est implantée de façon à ce qu'il n'y ait pas sans cesse à vérifier si le jeu en cours est une partie Alice ou non. La modularité de la programmation orientée objet s'est avérée très utile.

5 game.scala ou l'on peut enfin jouer

L'objet *Time* : Cet objet sert à gérer tout ce qui a trait aux temps dans l'application, il comprend la classe de threads *TimeThread* qui est décrite à la fin de cette section, il comprend aussi des méthodes permettant de passer d'un temps en seconde en un temps d'affichage plus facile à lire au format hh :mm :ss, et réciproquement.

Les joueurs : La classe *Player* est une classe abstraite qui contient l'identifiant du joueur (0 pour le noir et 1 pour le blanc), des booléens utiles pour le déplacement (*ai* et *move*) ainsi que la méthode *get_move* qui permet le déplacement des pièces.

Cette classe est étendue en deux : *AI* pour les joueurs de type intelligence artificielle et *Human* pour les joueurs humains. La classe *AI* est présenté en section 5 sur le fichier solve.scala.

La méthode *get_move* de la classe *Human* : Pour un joueur humain le déplacement se fait en deux étapes : sélection de la pièce et sélection de la case où se déplacer. La méthode *get_move* s'applique pour une case sur laquelle le joueur a cliqué, elle commence par vérifier que la pièce sélectionnée appartient bien au joueur (méthode *isHis*). Si tel est le cas, le booléen *first_choice_done* passe à *true* ce qui ouvre la sélection de la case, et les cases possibles du déplacement se colorie en rouge par appel de la méthode *DrawActions.draw_possiblie_moves*.

Lorsque le joueur clique sur une deuxième case *get_move* est à nouveau appelée, puisque l'on ne peut pas se déplacer sur une case contenant une de nos pièce, elle passe la première sélection (*isHis* renvoi false) et si le premier choix a été fait elle teste si la case est atteignable par la pièce dans ce cas elle applique le mouvement, sinon elle repasse *first_choice_done* à *false*.

Dans le cas d'une partie d'Alice, le mouvement se fait de la même manière rien ne change, puisque toutes les fonctions prennent l'échiquier en paramètre, ce qui permet de garder toutes les même fonctions sans les changer.

La classe *game* : Cette classe comprend toutes les variables nécessaires pour une partie : les différents booléens, les tableaux de pièces, de message, les joueurs... Au début de chaque partie on instancie une *game* dans une variable.

L'objet *Ksparov* : C'est l'objet central de notre application, c'est lui qui définit tout et qui fait appel au reste du code. Il comprend plusieurs éléments différents.

L'interface graphique : La variable *frame* définie dans *Ksparov* est la variable de base de l'interface graphique, c'est elle dont le contenu change pour changer de fenêtre. La fonction *main* fait simplement appel à la méthode *application.main(Array())* qui lance l'application en elle même avec la première frame, à savoir le menu principal.

L'initialisation des pièces : On retrouve dans *Ksparov* la variable *board* qui est un tableau de 32 pièces contenant toutes les pièces du jeu, les 16 premières cases étant les pièces du joueur blanc et les 16 dernières celle du joueur noir. La méthode *Ksparov.init_board* permet de définir le tableau initial avec toutes les pièces à leur position initiale.

L'initialisation du jeu : La méthode *Ksparov.init_game* est appelée après le choix du type de jeu dans le second menu de sélection, elle initialise les variables nécessaires pour commencer une partie. Elle commence par initialiser les pièces (via *Ksparov.init_board*) ensuite selon le type de jeu choisi elle définit les variables des joueurs correspondantes, enfin elle modifie les booléens aux bonnes valeurs. On peut noter ici que dans le cas d'une partie opposant une IA et un humain le choix de la couleur des joueurs est aléatoire.

Jouer un mouvement : Lorsqu'un joueur clique sur une case la méthode *Ksparov.play_move* est appelée, c'est elle qui gère tous les déplacements, son exécution est une longue série de conditions imbriquées. Tout d'abord elle vérifie si la partie n'a pas déjà été gagnée par un des joueurs (le booléen *game_won* d'une partie), dans ce cas elle ne fait rien. En effet dans ce cas les cases ne doivent pas répondre aux clics des joueurs.

Dans un second temps elle vérifie si l'IA n'est pas en pat à travers le booléen *game_nulle*). En effet étant aléatoire l'IA finit souvent à n'avoir plus aucuns mouvements possibles, le pat a donc été implémenté pour l'IA, de manière à ne pas boucler à l'infini dans le choix du mouvement de l'IA.

Si ces deux conditions ne sont pas vérifiées *Ksparov.play_move* appelle la méthode *get_move* du joueur actif sur la case sélectionnée. Si à l'issu de *get_move* le joueur s'est déplacé (booléen *moved* du joueur), il faut vérifier l'état de la partie après ce déplacement.

Il y a alors une série de test qui vérifient, dans l'ordre d'exécution, s'il y a échec et mat, s'il y a nulle ou s'il y a échec. Dans chacun de ces cas un message particulier est affiché donnant l'évènement en question et les booléens correspondant sont mis à jour. Si aucun de ces tests n'est validés, alors la main passe au joueur suivant et le booléen *first_choice_done* est remis à *false*.

Enfin si l'on est dans le cas d'une partie entre un joueur et une IA et que le joueur courant (donc après déplacement du joueur qui vient de cliquer) est l'IA, on applique le mouvement de l'IA en ouvrant le thread du mouvement de l'IA.

Il est important de noter que dans le cas d'une partie entre deux IA il faut cliquer sur une case pour obtenir le prochain mouvement, ce qui n'est pas le cas lors d'un matche entre une IA et un humain.

Les threads lors d'une partie : Lors d'une partie il y généralement deux threads qui tournent en plus du thread principal. Un premier qui est une instance de *Time.TimeThread* qui actualise toutes les 200 millisecondes le temps des joueurs. Tant qu'il est en vie, le thread ajuste le temps actuel du joueur courant, si a un moment le temps arrive à 0 alors que le joueur n'a pas effectué le nombre de coups nécessaires le thread termine immédiatement la partie et annonce quel joueur a gagné.

Un autre thread, qui est une instance de *Ksparov.AIMoveThread* et qui attend patiemment que le créneau soit ouvert pour jouer les coups des IA. Lors que l'on ouvre la possibilité de se déplacer pour un IA le thread commence par s'endormir le temps définit dans les paramètres, ensuite il applique le coup.

6 parse.scala ou les dangers de la PGN

L'objet *Save* : La sauvegarde de la partie en cours se fait via cet objet. Les méthodes sont internes pour la plupart. Il faut faire particulièrement attention à comment on récupère un mouvement. En effet, les deux méthodes à utiliser sont *add_move1* et *add_move2* et doivent être insérées dans la méthode *get_move* de toutes les sous-classes de player afin de récupérer les mouvements les uns après les autres. La première doit être placé juste avant l'application du mouvement, la deuxième après si le mouvement a bien eu lieux. La méthode à appeler pour sauvegarder une partie est *write_to_file*.

Pour l'affichage de l'interface de sauvegarde deux menus sont présents. Tout d'abord un menu pour la sauvegarde, c'est un menu qui ne permet que de choisir le nom de la sauvegarde, il propose ensuite plusieurs buttons pour quitter ce menu en sauvegardant.

Il y a ensuite un deuxième menu, un menu de sauvegarde avancée qui permet de définir tous les champs d'un PGN : nom des joueurs, date, lieu... Dans le premier menu, ces champs sont automatiquement remplis.

L'objet *Load* : Cet objet a pour fonction de lire un fichier au format PGN (Portable Game Notation). Pour cela une méthode *get_list_move_from_file* permet de récupérer la liste des mouvements (non encore parsés). Une fois cette liste établie, une partie peut être lancée avec comme joueur des *Reproducers*. *Reproducer* est une sous-classe de player et est un robot, en utilisant les règles du jeu définies dans Rules, il comprend le PGN. Il commence son tour en lisant la tête de l'attribut *list_of_moves*, qui est un coup au format PGN. S'il y a ambiguïté il décide, en faisant appel a la méthode *pre_move* des pièces, quelle pièce doit bouger et applique le mouvement.

Le menu de chargement permet deux méthodes de chargement. Dans un premier temps le joueur peut sélectionner un des fichiers présents dans le dossier *src/main/resources/Saves*, ce qui donne un accès rapide aux fichiers.

La seconde méthode consiste à ouvrir un explorateur de fichier pour sélectionner un fichier PGN. Lorsqu'un fichier est sélectionné, son chemin est conservé dans une variables, on peut alors le charger. Si aucun fichier ne portait le même nom dans le dossiers *Saves*, alors le fichier est copié dans le dossier pour faciliter son chargement.

Dans les deux cas l'utilisateur peut choisir deux façons de charger le fichier : soit il commence par le premier mouvement présent dans le fichier, soit il va directement à la fin du fichier ce qui permet de reprendre immédiatement le fichier là où il en était.

7 solve.scala ou l'intelligence limitée

L'intelligence artificielle : Pour le moment l'intelligence artificielle est encore pas très au point puisqu'elle choisit aléatoirement ses coups à chaque mouvement, mais elle permet de jouer contre quelqu'un. Son code a malgré tout été relativement optimisé.

La méthode *get_move* de l'IA : L'IA choisit son coup aléatoirement parmi les différents mouvements possible. Elle commence donc par choisir une pièce aléatoirement entre ses pièces et elle choisit ensuite lequel de ses mouvements elle applique.

Plus précisément, la méthode maintient un tableau de booléen à jour définissant si la pièce sélectionnée à déjà été testée. Ce tableau est initialisée en retirant les pièces mortes pour ne pas perdre de temps. Un nombre aléatoire en 1 et 16 est tiré, si la pièce n'a pas été testée (case du tableau *already_check* à *false*) alors la méthode cherche à savoir si un mouvement est possible.

Si un mouvement est possible, elle choisit alors une des cases possibles aléatoirement, sinon, si aucun mouvement n'a été appliqué, on vérifie s'il reste des mouvement possibles (s'il existe une case à *false* dans *already_check*), si tel est le cas on boucle, sinon l'IA est en pat.

La promotion pour les IA : La promotion pour les IA se fait de manière aléatoire, l'IA choisit donc une pièce aléatoirement parmi la tour, la dame, le fou ou le cavalier, on applique ensuite la promotion de manière classique.

8 pipe.scala ou la communication en point fort

Les threads : Pour la communication entre notre interface et gnuchess on utilise deux threads : ListenThread et SendThread. SendThread envoi, lorsque l'on indique qu'il y a quelque chose à envoyer, une chaîne de caractères à gnuchess. ListenThread écoute toutes les 100 millisecondes ce que gnuchess renvoi. Il lit ensuite caractère par caractère pour reconstruire la chaîne envoyé et la parse afin de passer les arguments comme il le faut.

Le joueur PipePlayer : Ce joueur prend les mouvements donnés par le thread ListenThread et applique les coups conformément.

9 Les défauts et erreurs connues

L'implémentation telle qu'elle est actuellement connaît certaines lourdeurs dans le code. En effet, pour les pièces le code utilise deux variables : une liste de 32 pièces et une liste de 64 cases qui sert à positionner chaque pièce de la première liste. Le code pourrait être modifier pour n'utiliser plus que la liste des 32 pièces.

De même pour chaque nouvelle fenêtre, chaque nouveau mouvement un grand nombre de classes sont instanciées, principalement dans la partie graphique, ce qui peut être lourd en mémoire. Ainsi on observe lors de certaines parties des ralentissement extrême dont nous n'avons pas réussi à comprendre la provenance.

En ce qui concerne rules, l'implémentation du roque est quelque peu lourde, et demande quelques répétitions de code.