Projet de Programmation
Master Informatique
2025–2026

université de BORDEAUX

Scrabble – Java

CODOGNAN Thomas, KAHLOUL Aïssa, BETTCHER Léonard, BLOTHIAUX Yanis, RAZE Erwan

29 janvier 2026

Introduction

Ce rapport préliminaire expose les fondations théoriques et architecturales du projet Scrabble en Java. Nous présentons l'historique du jeu, sa nature combinatoire, les défis algorithmiques qu'il pose, puis détaillons nos choix d'architecture et de structures de données.

1 Historique et Nature du Sujet

1.1 Historique du sujet

En 1931, l'architecte Alfred Mosher Butts crée un jeu combinant chance et réflexion. Après avoir analysé la fréquence des lettres dans le New York Times, il conçoit le système de points spécifique aux jetons. D'abord nommé Lexiko, puis Criss-Cross Words, le jeu est racheté en 1948 par James Brunot qui le baptise Scrabble. Le succès explose dans les années 1950 grâce à Jack Straus (président de Macy's) qui en lance la commercialisation massive.

1.2 Nature et enjeux du jeu

Le Scrabble est un jeu de lettres à **information imparfaite** (les tirages adverses sont cachés) et **stochastique** (dû au hasard de la pioche). L'objectif est de maximiser un score sur une grille (standard de 15×15 ou Super Scrabble de 21×21) en plaçant des mots d'un dictionnaire de référence.

Le jeu repose sur trois piliers :

1. Logique spatiale : Respect de la connexité (tout mot doit être rattaché aux lettres déjà posées et toutes les lettres posées se touchent entre elles) et de l'orthogonalité (validité de tous les mots formés par collision).

2. Logique mathématique : Calcul dynamique des points incluant la valeur des lettres et les multiplicateurs de cases (lettre/mot compte double/triple), appliqués dans l'ordre (lettres puis mots), ainsi que le bonus de 50 points (*Scrabble*) pour l'usage des 7 lettres du chevalet.

3. Logique linguistique : Arbitrage instantané de la validité des mots selon le lexique choisi.

Déroulement et règles de jeu

Début de partie : Le premier joueur doit placer un mot de deux lettres ou plus couvrant la case centrale, doublant ainsi le score de ce mot.

Exemple : Si le premier joueur place le mot « AXE » avec le « X » sur la case centrale (H8), le score du mot ($9+8+1 = 18$) est doublé pour un total de 36 points.

Actions du joueur : À chaque tour, le joueur peut poser un mot, échanger des jetons avec le sac (si celui-ci contient au moins 7 lettres) ou passer son tour.

Exemple de collision : Si le mot « FEU » est déjà sur le plateau, un joueur peut placer « R » au-dessus du « E » pour former « RE » verticalement tout en posant « RATS » horizontalement.

Notation des coups : Les coups sont notés par leurs coordonnées de départ suivies du mot et du score (ex : `H8 - VERSION - 34 pts`).

Exemple : La notation « `8H` » indique un mot horizontal commençant à la ligne 8, colonne H, tandis que « `H8` » indique un mot vertical commençant à la colonne H, ligne 8.

Fin de partie et victoire : La partie s'achève quand le sac est vide et qu'un joueur a épuisé son chevalet, ou après trois tours sans score (blocage). Le vainqueur est celui ayant le score le plus élevé après déduction de la valeur des lettres restantes sur les chevalets. Un match nul est possible si les scores finaux sont identiques.

Exemple de calcul final : Si le joueur A finit avec 300 points et que le joueur B possède encore un « W » (10 pts) et un « X » (10 pts) sur son chevalet, le score de B sera réduit de 20 points, et ces 20 points seront ajoutés au score de A.

1.3 Complexité du jeu

Il est crucial de distinguer la complexité intrinsèque du jeu de celle des algorithmes utilisés.

Espace d'état : Le nombre de positions légales sur un plateau de 15×15 est estimé à environ 10^{30}. Bien que moindre que celui des échecs, l'incertitude sur les jetons restants complexifie la recherche.

Détail du calcul combinatoire : Pour un chevalet de 7 lettres et un dictionnaire type ODS (380 000 mots), le nombre de placements théoriques au premier tour est le produit des arrangements $P(7, n)$ par les orientations et positions sur l'étoile centrale.

Heuristiques et incertitude : Le score immédiat n'est pas le seul facteur. Une stratégie gagnante doit évaluer la « qualité du reliquat » (garder des lettres polyvalentes comme les voyelles ou le « S ») et l'obstruction (ne pas ouvrir de cases bonus pour l'adversaire).

1.4 Problématique algorithmique

La difficulté majeure est l'explosion combinatoire. Le moteur doit :

1. Rechercher exhaustivement tous les coups légaux en un temps imperceptible.

2. Simuler une IA capable de prendre des décisions non-déterministes en gérant l'état futur du plateau et l'accessibilité aux bonus après le coup, afin de permettre la création de parties joueurs/intelligences artificielles.

1.5 Existant

L'état de l'art repose sur des travaux académiques et des logiciels éprouvés :

1. Littérature scientifique : L'algorithme de référence pour la génération rapide de coups est celui de Steven Gordon (GADDAG) ou le travail d'Appel et Guy (DAWG). Ces structures minimisent l'espace mémoire tout en maximisant la vitesse de parcours du dictionnaire.

2. Logiciels de référence : Quackle est le standard pour l'analyse stratégique. Il utilise des simulations de Monte-Carlo pour comparer les coups en fonction de l'espérance de gain sur les tours suivants.

3. Plateformes : L'ISC (*Internet Scrabble Club*) sert de référence pour la pratique compétitive ou classique, garantissant une application stricte des règles internationales.

2 Algorithmes et Structures de Données

L'implémentation d'un joueur artificiel performant au Scrabble repose sur la résolution de deux problèmes distincts : la génération efficace de tous les coups légaux et la sélection du meilleur coup parmi ceux-ci. Cette section détaille les choix algorithmiques retenus pour répondre à ces besoins.

2.1 Modélisation du problème

Contrairement aux Échecs ou au Go, qui sont des jeux à information parfaite et déterministes, le Scrabble est un jeu à information imparfaite, on ne connait pas lettres de l'adversaire et le tirage des lettres repose sur le hasard.

Une approche gloutonne, qui consisterait à jouer systématiquement le coup rapportant le plus de points à l'instant t, est sous-optimale. Elle néglige deux facteurs cruciaux :

1. Le reliquat (Rack Leave) : La qualité des lettres conservées pour le tour suivant.

2. L'ouverture du plateau : Le risque d'offrir une case "Mot Compte Triple" à l'adversaire.

2.2 L'Algorithme Minimax

L'algorithme Minimax est la base de la décision dans les jeux à somme nulle. Il parcourt l'arbre de jeu jusqu'à une profondeur d donnée.

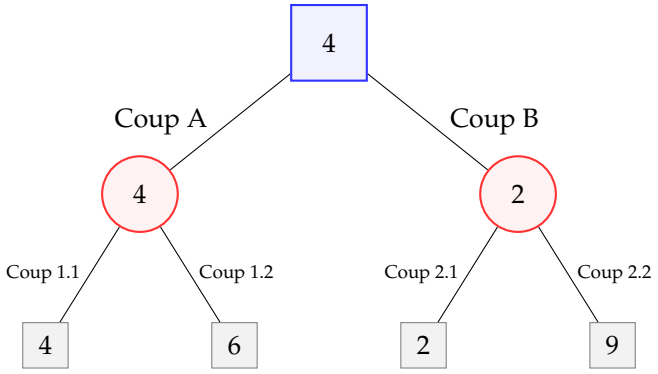

FIGURE 1 – Exemple de déroulement d'un arbre Minimax

Fonctionnement

L'algorithme alterne entre deux types de nœuds :
1. Nœuds MAX : Cherche à maximiser le score final.

2. Nœuds MIN : Cherche à minimiser le score.
L'application stricte de Minimax au Scrabble pose un problème de modélisation. Minimax suppose que l'adversaire jouera toujours le coup optimal pour nous contrer. Or, pour déterminer ce coup optimal, l'algorithme doit connaître les lettres de l'adversaire. Dans une simulation Minimax, nous devrions supposer le pire cas : l'adversaire possède exactement les lettres nécessaires pour réfuter notre coup (ex : un "Z" pour atteindre un mot compte triple ouvert). Cette approche rend l'IA excessivement défensive et inadaptée à la réalité probabiliste du tirage.

Listing 1 – Pseudo-code de l'algorithme Minimax

```
Fonction Minimax(état, profondeur, est_joueur_max):
    // Cas de base : feuille de l'arbre ou profondeur atteinte
    SI profondeur == 0 OU partie_terminée(état) ALORS
        RETOURNER evaluer(état)

    SI est_joueur_max ALORS
        // Tour de l'IA (Cherche à maximiser son score)
        meilleure_valeur = -INFINI
        POUR CHAQUE coup DANS generer_coups_possibles(état) FAIRE
            nouvel_état = jouer_coup(état, coup)
            valeur = Minimax(nouvel_état, profondeur - 1, FAUX)
            meilleure_valeur = MAX(meilleure_valeur, valeur)
        RETOURNER meilleure_valeur

    SINON
        // Tour de l'adversaire (Cherche à minimiser notre gain)
        pire_valeur = +INFINI
        POUR CHAQUE coup DANS generer_coups_possibles(état) FAIRE
            nouvel_état = jouer_coup(état, coup)
            valeur = Minimax(nouvel_état, profondeur - 1, VRAI)
            pire_valeur = MIN(pire_valeur, valeur)
        RETOURNER pire_valeur
```

2.3 L'Algorithme Expectiminimax

Pour intégrer la dimension aléatoire du tirage de lettres, nous utilisons l'algorithme **Expectiminimax**. C'est une variation du Minimax qui introduit un troisième type de nœud : les **Nœuds de Hasard**.

L'arbre de décision se décompose désormais en trois niveaux par tour de jeu :
1. Nœud MAX : L'IA choisit un coup parmi les coups possibles générés.

2. Nœud CHANCE : Ce nœud représente l'incertitude sur les lettres de l'adversaire. Il possède une branche pour chaque tirage possible, pondérée par sa probabilité.

3. Nœud MIN : L'adversaire joue le meilleur coup possible avec le tirage attribué par le nœud chance.

La valeur d'un nœud de hasard n'est ni un minimum ni un maximum, mais une espérance mathématique. Pour un état s correspondant à un nœud de hasard :

$$\text{Expect}(s) = \sum_{r \in \text{Tirages}} P(r) \times v(\text{successeur}(s, r))$$

Où $P(r)$ est la probabilité que le tirage soit r, sachant les lettres déjà visibles.

Listing 2 – Pseudo-code de l'algorithme Expectiminimax

```
Fonction Expectiminimax(état, profondeur, type_nœud):
    SI profondeur == 0 OU partie_finie ALORS
        RETOURNER evaluer(état)

    SI type_nœud == MAX ALORS
        valeur_max = -INF
        POUR CHAQUE coup DANS generer_coups(état) FAIRE
            val = Expectiminimax(appliquer(état, coup), profondeur-1, CHANCE)
            valeur_max = MAX(valeur_max, val)
        RETOURNER valeur_max

    SINON SI type_nœud == CHANCE ALORS
        somme_valeurs = 0
        // Échantillonnage de Monte-Carlo
        mains_simulees = generer_mains_aleatoires(état, 20)
        POUR CHAQUE main DANS mains_simulees FAIRE
             // On attribue la main à l'adversaire
            état_simulé = set_main_adversaire(état, main)
            somme_valeurs += Expectiminimax(état_simulé, profondeur, MIN)
        RETOURNER somme_valeurs / 20

    // ... Cas MIN symétrique au MAX
```

2.3.1 Défi combinatoire et Échantillonnage

Le facteur de branchement aux nœuds de hasard est gigantesque. Simuler tous les tirages possibles de 7 lettres parmi les N lettres restantes est impossible en temps réel.

Pour contourner cette complexité, nous utiliserons une méthode d'approximation par échantillonnage. Au lieu de calculer la somme sur tous les tirages, nous générons un sous-ensemble représentatif de N mains adverses aléatoires (ex : 20 simulations) basées sur la distribution réelle des lettres restantes.

L'algorithme devient alors :

1. L'IA joue un coup C.

2. Nous simulons N mains adverses possibles.

3. Pour chaque main, nous calculons la meilleure réponse de l'adversaire.

4. La valeur du coup C est la moyenne des scores résultant de ces simulations.

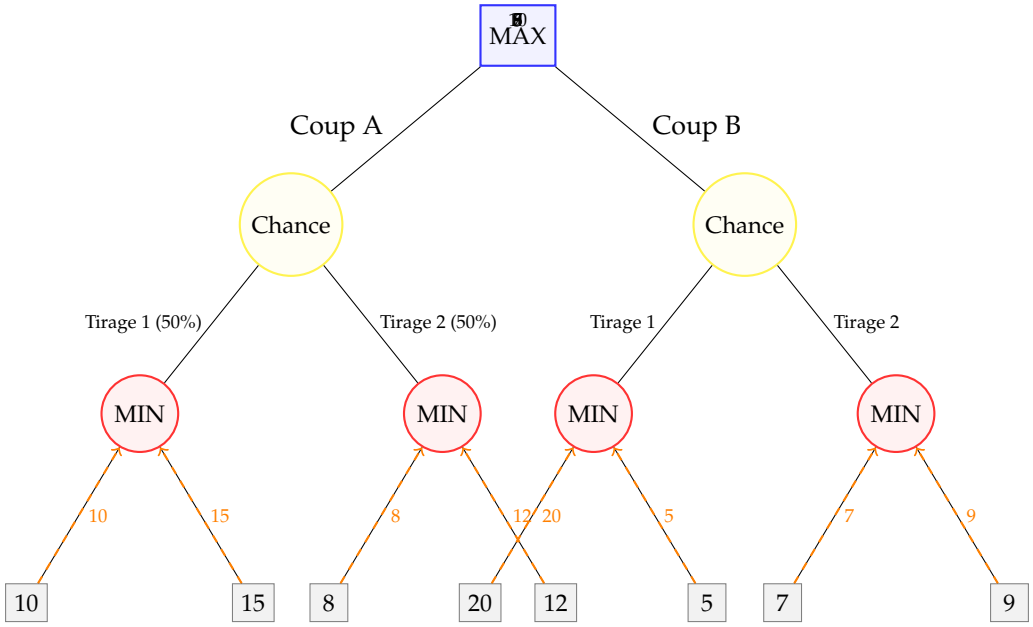

FIGURE 2 – Représentation corrigée d'un arbre Expectiminimax à 3 niveaux avec valeurs de remontée

3 Structure de données pour la recherche : GADDAG

Pour que les algorithmes de décision (Minimax ou Expectiminimax) soient réellement performants, ils doivent pouvoir accéder à la liste exhaustive des coups légaux de manière quasi instantanée. À cette fin, nous utilisons la structure de données **GADDAG**, inventée par Steven Gordon en 1994.

Inspiré du **DAWG** (Directed Acyclic Word Graph), le GADDAG est spécifiquement optimisé pour le Scrabble, car il facilite la construction de mots autour d'une "ancre" (une lettre déjà présente sur le plateau).

3.1 Fonctionnement et Transformation

Le principe du GADDAG repose sur une transformation du dictionnaire. Pour chaque mot, plusieurs représentations sont générées en le découpant autour de chaque lettre ancre. La partie située à gauche de cette ancre est inversée, puis séparée de la partie droite par un caractère spécial (souvent `>` ou `$`), puis la fin du mot est notée normalement.

Par exemple, pour le mot **"CHAT"**, les représentations stockées dans l'arbre sont les suivantes :

1. **`C > H A T`** (Ici, l'ancre est C)
2. **`H C > A T`** (Ici, l'ancre est le H, donc on met un C à sa gauche, puis on passe à droite pour poser A-T)
3. **`A H C > T`** (Ici, l'ancre est le A, donc on met un C et un H à sa gauche, puis on passe à droite pour poser le T)
4. **`T A H C >`** (Ici, l'ancre est le T, et on complète tout le mot vers la gauche : A, puis H, puis C)

Cette redondance permet à l'algorithme de trouver des mots quel que soit le point d'ancrage sur la grille, sans avoir à "deviner" le début du mot.

3.2 Comparaison : GADDAG vs DAWG

Contrairement au GADDAG, l'algorithme **DAWG** ne fonctionne que de manière unidirectionnelle (de la première à la dernière lettre).

Critère	DAWG	GADDAG
Vitesse	Plus lent (doit deviner l'origine du mot).	**Très rapide** (accès direct via n'importe quelle lettre).
Mémoire	Très compact.	Plus lourd (5 fois la taille d'un DAWG en moyenne).
Complexité	Plus simple à implémenter.	Plus complexe et structure plus dense.

TABLE 1 – Comparaison des structures de données pour le Scrabble

De plus, il y a un moyen montré dans l'article originel publiant l'idée du GADDAG, de réduire la mémoire utilisée par le GADDAG. Le moyen utilisé est de réduire le nombre de sorties pour chaque mot en reliant les répétitions inutiles de fin de mots comme le montre cet exemple fait pour le mot CARE.

Cette minimisation est aussi faite en début d'arbre, juste après la racine, pour les mots finissant avec deux mêmes lettres, par exemple CALL, qui aura, sans minimisation, deux branches de l'arbre commençant avec L, mais en liant les deux à la première ligne, pour les séparer à partir de la lettre suivante, nous gagnons de la mémoire et la structure GADDAG devient déterministe, vu il n'y aura qu'un seul chemin pour chaque possibilité.

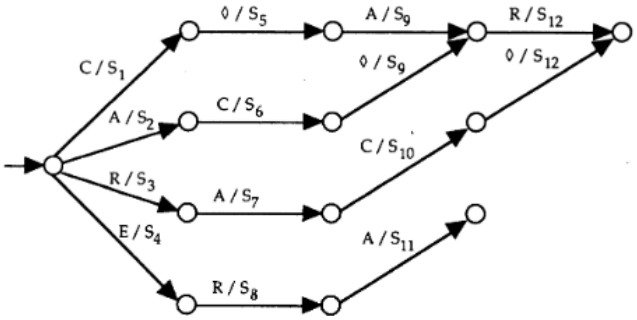

FIGURE 3 – Graphe du GADDAG semi-minimisé pour « CARE ». (Source : [2])

3.3 Synthèse des performances

L'utilisation du GADDAG présente des avantages majeurs pour notre moteur de jeu :

1. **Efficacité maximale** : C'est l'algorithme le plus rapide pour la génération de coups, sans répétitions inutiles.
2. **Polyvalence d'ancrage** : Il permet de trouver efficacement des mots s'insérant entre deux lettres ou se collant parallèlement à un mot existant.
3. **Unicité du chemin** : Une fois la structure construite, il n'existe qu'un seul chemin pour chaque mot avec un point d'ancrage donné.

En résumé, bien que le GADDAG soit plus exigeant en termes de mémoire vive et plus complexe à coder, il constitue le meilleur choix pour garantir une IA capable de rivaliser avec des joueurs experts en analysant des milliers de combinaisons par seconde.

4 Spécifications étendues

F16 : Proposer de sauvegarder avant de quitter

Description : Le programme doit demander à l'utilisateur s'il souhaite sauvegarder la partie avant de quitter, si des modifications non sauvegardées existent.

4.1 Prérequis :

1. **F2 :** Fichier de configuration (pour définir les chemins de sauvegarde par défaut).
2. **F21 :** Format de sauvegarde (pour savoir comment structurer les fichiers de sauvegarde).
3. **F15 :** Shell interactif (pour gérer les commandes comme quit et save).

4.2 Type de besoin :

Fonctionnel.

4.3 Découpage en sous-besoins :

Sous besoins 1. Détecter les modifications non sauvegardées

Description : Le programme doit détecter si l'état actuel du jeu diffère de l'état sauvegardé. Cela inclut les modifications du plateau, des scores et des lettres restantes.
Actions :

— Stocker l'état du jeu après chaque sauvegarde.

— Comparer l'état actuel avec l'état sauvegardé à chaque tentative de quitter le programme.

Fonction associée :

```
boolean modificationsNonSauvegardees(EtatJeu etatActuel, EtatJeu etatSauvegarde)
```

Paramètres :

— `etatActuel` : Représente l'état actuel du jeu (plateau, scores, lettres restantes).

— `etatSauvegarde` : Représente l'état du jeu lors de la dernière sauvegarde.

Retour :

— `true` : S'il y a des modifications non sauvegardées.

— `false` : Si aucun changement n'a été fait depuis la dernière sauvegarde.

Test unitaire :

— **Cas 1** : `etatActuel` contient le mot "CHAT" sur le plateau, `etatSauvegarde` est vide. Résultat attendu : `true` (car il y a des modifications).

— **Cas 2** : `etatActuel` et `etatSauvegarde` sont identiques. Résultat attendu : `false` (pas de modifications).

Sous besoins 2. Demander confirmation à l'utilisateur

Description : Le programme doit afficher un message de confirmation et lire la réponse de l'utilisateur.
Actions :

— Afficher le message : `Save the game before quitting? [y/N].`

— Lire la réponse de l'utilisateur.

— Retourner `true` si la réponse est "y" ou "Y", `false` sinon.

Fonction associée :

```
boolean demanderConfirmation()
```

Retour :

— `true` : Si l'utilisateur répond "y" ou "Y".

— `false` : Si l'utilisateur répond "n", "N" ou toute autre réponse.

Test unitaire :

— **Cas 1** : L'utilisateur répond "y". Résultat attendu : `true`.

— **Cas 2** : L'utilisateur répond "n". Résultat attendu : `false`.

Sous besoins 3. Sauvegarder la partie

Description : Si l'utilisateur choisit de sauvegarder, le programme doit écrire l'état actuel du jeu dans un fichier.
Actions :

— Demander à l'utilisateur le chemin du fichier de sauvegarde.

— Écrire l'état du jeu dans le fichier au format défini (JSON ou format personnalisé).

Fonction associée :

```
boolean sauvegarderPartie(EtatJeu etatJeu, String cheminFichier)
```

Paramètres :

— `etatJeu` : L'état actuel du jeu à sauvegarder.

— `cheminFichier` : Le chemin du fichier où sauvegarder la partie.

Retour :

— `true` : Si la sauvegarde a réussi.

— `false` : Si une erreur survient.

Test unitaire :

— **Cas 1** : Sauvegarder un état de jeu valide dans un fichier accessible. Résultat attendu : `true` et le fichier contient les données correctes.

— **Cas 2** : Sauvegarder dans un chemin invalide (ex. : `/dossier/inexistant/fichier.json`). Résultat attendu : `false`.

Sous besoins 4. Gérer les erreurs de sauvegarde

Description : Le programme doit gérer les erreurs lors de la sauvegarde (ex. : permissions insuffisantes, disque plein) et redemander à l'utilisateur s'il souhaite réessayer.
Actions :

— Capturer les exceptions lors de l'écriture du fichier.

— Afficher un message d'erreur et redemander à l'utilisateur s'il veut réessayer.

Fonction associée :

```
boolean gererErreurSauvegarde(EtatJeu etatJeu)
```

Paramètres :

— `etatJeu` : L'état actuel du jeu à sauvegarder.

Retour :

— `true` : Si la sauvegarde a finalement réussi.

— `false` : Si l'utilisateur abandonne après une erreur.

Test unitaire :

— **Cas 1** : L'utilisateur entre un chemin invalide, puis un chemin valide. Résultat attendu : `true` après la deuxième tentative.

— **Cas 2** : L'utilisateur abandonne après une erreur. Résultat attendu : `false`.

4.4 Intégration dans l'architecture du projet

Voici comment la fonctionnalité de sauvegarde sera intégrée dans l'architecture globale du projet :

1. **Module CLI :** La logique de F16 (demande de sauvegarde avant de quitter) sera intégrée dans le module gérant les commandes utilisateur. Ce module est responsable de l'interaction avec l'utilisateur via le terminal.

2. **Module Sauvegarde :** Les fonctions de sauvegarde (`sauvegarderPartie`, `gererErreurSauvegarde`) seront regroupées dans un module dédié. Ce module contiendra toute la logique de gestion des fichiers de sauvegarde.

3. **Fichiers de configuration :** Le chemin par défaut des sauvegardes sera défini dans le fichier `.scrabblerc` (F2). Ce fichier de configuration permettra de personnaliser les chemins de sauvegarde et d'autres paramètres du jeu.

4.5 Scénario utilisateur

Voici un scénario complet montrant comment l'utilisateur interagit avec la fonctionnalité de sauvegarde :
Étape 1 : L'utilisateur lance une partie et joue un coup en posant le mot "CHAT" sur le plateau.
Étape 2 : L'utilisateur tape `quit` pour quitter le jeu.
Étape 3 : Le programme demande : "Save the game before quitting ? [y/N]".
Étape 4 : L'utilisateur répond "y" pour confirmer la sauvegarde.
Étape 5 : Le programme demande alors le chemin du fichier de sauvegarde. L'utilisateur entre "ma_partie.json".
Étape 6 : Le programme sauvegarde la partie dans le fichier "ma_partie.json" et quitte.
Étape 7 : L'utilisateur relance le programme et charge "ma_partie.json" pour vérifier que l'état de la partie est restauré correctement.

5 Architecture du projet

L'objectif de ce projet est de concevoir une application robuste du Scrabble, capable de gérer aussi bien une interface console (CLI) qu'une interface graphique (JavaFX), tout en intégrant des fonctionnalités avancées comme le jeu en réseau et l'intelligence artificielle. Le projet est actuellement structuré sous forme de squelettes de classes.

5.0.1 Structure des Répertoires

Le projet est organisé selon une structure hiérarchique claire séparant le modèle, les vues et le contrôleur :

5.0.2 Structure des Répertoires

```
src/main/java/fr/u_bordeaux/scrabble/
|-- model/
|   |-- core/          (Bag, Board, Game, Move...)
|   |-- ai/            (AIPlayer, MinimaxSolver)
|   |-- dictionary/    (Lexicon, Gaddag, Dawg)
|   |-- network/       (GameServer, GameClient)
|   +-- enums/
|-- view/
|   |-- cli/           (TerminalView, CommandHandler)
|   +-- gui/           (JavaFxView)
|-- controller/        (GameController)
+-- io/                (ConfigLoader, GameLoader)
```

5.1 Organisation Globale et MVC

Pour répondre à l'exigence de modularité (F7, F39), nous avons adopté le patron **MVC (Modèle-Vue-Contrôleur)**. Cette séparation est indispensable pour isoler la logique métier des différentes interfaces. L'implémentation s'appuie sur le patron Observer pour notifier automatiquement les interfaces de toute modification du modèle, garantissant une synchronisation réactive tout en maintenant un couplage faible.

1. **Le Modèle (`fr.u_bordeaux.scrabble.model`)** : Il contient l'état du jeu (plateau, sac, joueurs), ainsi que les packages `network`, `ia` et `dictionary`. Il est conçu pour être indépendant de l'affichage.

2. **La Vue (`fr.u_bordeaux.scrabble.view`)** : Elle est divisée entre une implémentation **cli** (shell interactif avec codes ANSI pour les couleurs du plateau) et une implémentation **gui** sous JavaFX.

3. **Le Contrôleur (`fr.u_bordeaux.scrabble.controller`)** : Il gère les entrées utilisateur et les options de lancement via `commons-cli`. Il assure également le chargement de la configuration `.scrabblerc` (format INI).

5.2 Architecture du Cœur de Jeu et Flexibilité

L'architecture que nous visons comporte une séparation nette entre les données brutes et les règles de gestion, permettant au logiciel de s'adapter dynamiquement aux variantes du jeu et aux environnements linguistiques.

1. **Plateau Dynamique** : La classe `Board` utilise une matrice d'objets `Square`, chacun associé à un `SquareType` (énumération gérant les bonus de score). Cette structure permet de passer d'un plateau standard 15×15 au mode Super-Scrabble 21×21 sans modifier la logique de placement. Les modules du package `factory` implémenteront le design pattern **Factory** pour générer ces différents plateaux, dans le but de séparer l'instanciation des grilles de leur utilisation par le moteur de jeu (et de faciliter l'ajout de nouvelles variantes sans modifier le code existant).

2. **Abstraction des Joueurs** : Nous utilisons une classe abstraite `Player` pour traiter de manière uniforme les joueurs humains (locaux), les intelligences artificielles et les joueurs réseaux. Chaque joueur possède son propre `Rack` (chevalet) de 7 objets `Tiles` piochés dans `Bag`, qui pourront être joués sur le `Board` avec un `Move`.

3. **Internationalisation et Localisation (i18n/l10n)** : L'application utilise les mécanismes `ResourceBundle` et `Locale` de Java pour supporter l'anglais et le français (choisi automatiquement en fonction des variables d'environnement `LANG` ou `LC_ALL`). Cette gestion impacte directement la logique du jeu en chargeant dynamiquement la distribution des lettres, leurs valeurs respectives dans le sac (`Bag`), ainsi que le dictionnaire associé via le module `lexicon`.

5.3 Gestion des Coups et Persistance

La gestion des actions repose sur une traçabilité complète, nécessaire pour les fonctionnalités de confort utilisateur et la sauvegarde.

1. **Pattern Command** : Chaque action est encapsulée dans un objet `Move`, avec le choix de l'action spécifié par un enum `MoveType` (PLAY, EXCHANGE et PASS). La logique d'application des `Move` sera géré par la classe `MoveHandler`, pour séparer cette logique des données. Ces choix d'implémentation faciliteront l'implémentation d'undo/redo, des fichiers de sauvegarde et des différents joueurs, car chaque coup devient une action indépendante qu'on peut simplement empiler, annuler ou stocker sans toucher au reste du code.

2. **Format de Sauvegarde Robuste** : Les fichiers de sauvegarde (.txt en ASCII) sont structurés en trois blocs obligatoires : `[settings]`, `[game]` et `[history]`. Notre parseur sera robuste en ignorant les commentaires en ligne (#) ou en bloc ({...}) et en signalant précisément la ligne en cas d'erreur de format.

5.4 Algorithmes, Dictionnaire et Réseau

Cette section regroupe les composants les plus techniques du projet, où la performance du moteur de jeu rencontre les capacités d'extension réseau.

1. **Moteur de recherche et Dictionnaire** : La validation des mots et la recherche de coups reposent sur le module `dictionary`. Il s'agit d'une interface abstraite permettant d'interchanger deux structures de données haute performance : le **DAWG** (F28), optimisé pour la vérification rapide, et le **GADDAG** (F29), indispensable pour générer efficacement tous les coups possibles à partir d'un tirage. Le module `lexicon` assure le chargement de ces données en fonction de la langue choisie (F26).

2. **Intelligence Artificielle** : L'IA exploite les capacités du GADDAG pour trouver les meilleurs placements et simuler des joueurs artificiels `AIPlayer`. Au-delà d'une simple recherche de score maximal, l'activation de l'option `-ai-exptiminimax` lance un algorithme qui anticipe les futurs tirages et les réponses probables de l'adversaire (F36). Pour la recherche de mots expérimentale, nous prévoyons l'intégration de la bibliothèque **TensorFlow Java API** (F37) avec un temps de réflexion borné à 5 secondes par défaut.

3. **Architecture Réseau** : L'infrastructure est centralisée par un `NetworkManager` qui orchestre les échanges de données. Le `GameServer` gère les connexions multi-threadées via TCP sur le port 12345, tandis que le `GameClient` assure la liaison avec le moteur local. Les interactions (coups, invitations) sont encapsulées dans des objets `Packet` transitant au format ASCII. La découverte automatique s'appuie sur un broadcast UDP sur le port 12346, avec un système d'invitations expirant après 5 minutes (F40).

5.5 Environnement et Qualité

Le projet est automatisé via **Maven** pour un déploiement sur **GNU/Linux**. La fiabilité est garantie par une suite de tests **JUnit** associée à **Jacoco**, avec un objectif de **85% de couverture**. L'intégralité du code et de la **Javadoc** respecte les standards **Google** en anglais. Enfin, la robustesse est assurée par un traitement systématique des erreurs sur **stderr**, garantissant une exécution sans crash.

Références

[1] Fédération Internationale de Scrabble Francophone. *Règlement international du Scrabble classique*. Fédération Française de Scrabble, 2024.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] Steven A. Gordon. A faster scrabble move generation algorithm. *Softw. Pract. Exp.*, 24(2) :219–232, 1994.

[4] Le Particulier. Il était une fois une marque : l'histoire du Scrabble, février 2025. Consulté le 29 janvier 2026.

[5] University of Illinois Urbana-Champaign. Cs440 lecture : Games 5 - stochastic games, expectimax, 2020. Course : Artificial Intelligence (Fall 2020).