Université Paris-Saclay — Polytech APP5 — Année 2025-2026

IA et contraintes

Devoir Escampe

Conception et réalisation d'un joueur artificiel

Rapport — version finale

Ethan Puyaubreau & Antonin Russac

30 mai 2026

Joueur : `escampe.JoueurPuyaubreauRussac`

Encadrement : Yue Ma (yue.ma@universite-paris-saclay.fr)

Sommaire

1. Présentation et règles

Escampe se joue sur un plateau de 36 cases (6×6). Chaque case porte un liseré *simple*, *double* ou *triple*. Chaque joueur dispose d'une **licorne** et de cinq **paladins** (noirs ou blancs). Les lignes vont de 1 à 6, les colonnes de A à F, et le but est de **prendre la licorne adverse**.

Ce qui fait l'originalité du jeu, c'est la **contrainte de liseré** : la pièce que l'on joue doit partir d'une case dont le liseré est le même que celui de la case où l'adversaire vient de poser sa pièce. Ce liseré de départ fixe aussi le nombre de pas (1, 2 ou 3), orthogonaux, sans traverser ni repasser sur une case déjà visitée. On ne capture qu'en s'arrêtant, au dernier pas, sur la licorne adverse ; les paladins, eux, sont imprenables. Un joueur qui ne peut rien jouer passe son tour. Toute la difficulté revient donc à coincer l'adversaire en lui imposant des liserés qui le bloquent.

Pour le déroulement, Noir place d'abord ses six pièces sur les deux lignes d'un bord de son choix (haut ou bas), puis Blanc fait de même sur le bord opposé, et c'est **Blanc qui joue le premier coup**. Le rapport reprend nos choix de modélisation (parties 1 et 2) puis détaille la conception du joueur pour le tournoi (partie 3), chiffres à l'appui.

2. Analyse des caractéristiques du jeu

Nous reprenons les sept questions de la première partie, cette fois à la lumière du code que nous avons réellement écrit.

Q1 — Modélisation d'un état

Le plateau est un tableau `int[6][6]` : `board[ligne][colonne]`, avec `ligne 0` = ligne 1 (en bas) et `colonne 0` = A. Chaque case vaut une constante de pièce (`EMPTY`, `WHITE_LICORNE`, `WHITE_PALADIN`, `BLACK_LICORNE`, `BLACK_PALADIN`). Quatre informations que le tableau ne porte pas, mais dont la règle a besoin, sont gardées à côté :

- `lastTileType` : le liseré imposé au coup suivant (`-1` quand il n'y a pas de contrainte) ;
- `currentPlayer` : le joueur au trait ;
- `blackPlaced`, `whitePlaced` : la fin des phases de placement ;
- `blackRows` : le bord choisi par Noir, qui détermine celui de Blanc.

Le tableau d'entiers donne un accès en O(1) à n'importe quelle case et se copie sans effort, ce qui compte pour l'arbre de recherche ; il se sérialise aussi directement vers le format de fichier. Surtout, il autorise un schéma `make`/`unmake` qui n'alloue rien (voir §6). Le seul point gênant est que la contrainte de liseré vit en dehors du tableau : il faut penser à la mettre à jour à chaque coup, ce que nous centralisons dans `play`.

La carte des liserés est figée dans la constante `TILE_MAP`, recopie de la figure 4 de l'énoncé (ligne 1 en bas) :

```
      A  B  C  D  E  F
6     3  2  2  1  3  2
5     1  3  1  3  1  2
4     2  1  3  2  3  1
3     2  3  1  2  1  3
2     3  1  3  1  3  2
1     1  2  2  3  1  2
```

> Nous avons extrait par réflexion la carte qu'utilise l'arbitre dans sa propre classe de jeu, et elle coïncide case pour case avec la nôtre (elle colle aussi à l'exemple de la figure 6). La vérification valait le coup :

> une carte fausse aurait fait rejeter nos coups par l'arbitre.

Q2 — Détection de fin de partie

La partie s'arrête dès qu'une des deux licornes quitte le plateau ; il n'y a pas d'autre cas, donc pas de nul. Le test (`gameOver`) est un simple balayage en O(1). En recherche, le moteur n'attend même pas ce balayage : il repère la capture à l'instant où le coup la produit (§6).

Q3 — Sources de difficulté et facteur de branchement

Quatre choses rendent le jeu retors : la contrainte de liseré, qui fait varier fortement la mobilité ; la dépendance entre tours, puisque la case d'arrivée qu'on choisit dicte les pièces que l'adversaire pourra bouger ; l'asymétrie du plateau, avec des zones riches en liserés triples (mobiles) et d'autres en liserés simples ; et le risque qu'une pièce, voire un joueur entier, se retrouve bloqué et doive passer.

Côté **facteur de branchement**, nous avions avancé en première partie une borne théorique de l'ordre de 120 (six pièces, jusqu'à une vingtaine de destinations sur un liseré triple). En pratique c'est beaucoup moins, parce que la contrainte de liseré ne laisse jouables que les pièces du bon type. Une simulation de 30 000 parties aléatoires (utilitaire `escampe.Branching`) donne :

Situation	Branchement maximal observé
Coup contraint (un liseré imposé)	45
Coup libre (1er coup ou après un pass)	49
Branchement moyen (toutes positions)	≈ 8,9

Avec une moyenne sous 10, l'alpha-bêta descend profond en quelques secondes (§6).

Q4 — Coups imparables

Il n'y a pas de coup gagnant à coup sûr dès le départ : comme l'adversaire choisit sa case d'arrivée, donc le liseré qu'il nous impose, il peut toujours désamorcer une menace au mauvais moment. Ce qui existe, en revanche, ce sont des positions de **zugzwang partiel**, où il est forcé d'imposer précisément le liseré qui ouvre la capture.

L'énoncé en donne un cas net (figure 6). Noir vient de jouer en D4, une case à liseré double, donc Blanc doit partir d'une case double : il choisit **F6 – E5** (F6 est double). Noir est alors contraint de jouer depuis un liseré simple comme E5, et son seul coup raisonnable est **A1 – A2**. Or A2 est à liseré triple : Blanc enchaîne **C2 × C1**, son paladin en C2 parcourant les trois pas C2 → D2 → D1 → C1 pour prendre la licorne noire. La séquence est imparable localement : une fois Noir poussé en A2, il ne peut plus empêcher la prise.

Ce genre de combinaison ne se construit pas mécaniquement depuis l'ouverture, il y a trop de degrés de liberté ; mais notre alpha-bêta la trouve et la joue dès qu'elle entre dans son horizon de recherche.

Q5 — Critères pour l'heuristique

Cinq critères nous semblaient pertinents : la distance à la licorne adverse, la mobilité différentielle, le contrôle du liseré qu'on impose, la protection de sa propre licorne et l'avancée des pièces. Au final (§7), l'évaluation retenue tient surtout à deux d'entre eux, la proximité de nos paladins à la licorne adverse (attaque) et l'éloignement des paladins adverses de la nôtre (défense) ; le reste, la recherche s'en charge assez bien toute seule.

Q6 — Stratégie selon la phase

- **Ouverture (placement)** : c'est irréversible, donc on sécurise d'emblée la licorne et on s'arrange pour pouvoir toujours jouer (§5).
- **Milieu** : on manœuvre pour menacer la licorne adverse tout en gardant la main sur le liseré qu'on impose, en visant le zugzwang partiel.
- **Finale** : dès qu'une capture est en vue, c'est le calcul tactique qui décide.

Q7 — Majorant du nombre de coups et gestion du temps

Aucune pièce ne disparaît avant la prise finale, donc une partie peut traîner. En bornant le branchement par tour sur quelques dizaines de tours, on arrive à un ordre de grandeur de 400 à 600 demi-coups. Pour rester dans les 300 s par joueur, on s'appuie sur l'approfondissement itératif, l'élagage alpha-bêta et un budget par coup calculé à partir du temps restant (§8).

3. Modélisation : la classe `EscampeBoard`

`EscampeBoard` implémente l'interface fournie `Partie1` (`setFromFile`/`saveToFile`, `isValidMove`, `possiblesMoves`, `play`, `gameOver`) et suit les conventions de l'arbitre : coup régulier `"B1-D1"`, placement `"C6/A6/B5/D5/E6/F5"` avec la licorne en tête puis les cinq paladins, et pass `"E"`.

Le format de fichier reprend celui de l'énoncé : six lignes de plateau du bas vers le haut, avec `N/n` pour le noir, `B/b` pour le blanc et `-` pour le vide, chaque ligne encadrée de son numéro ; les lignes en `%` sont des commentaires. Nous y rangeons justement l'état hors-plateau (liseré courant, joueur, bord de Noir), de sorte qu'une sauvegarde se recharge à l'identique.

Pour générer les coups, on part d'une case et on énumère les arrivées par un parcours en profondeur avec retour arrière : exactement N pas (N = liseré de départ), cases intermédiaires vides, et case finale soit vide soit occupée par la licorne adverse, auquel cas c'est une capture. `possiblesMoves` ne garde que les pièces sur le bon liseré et renvoie `["E"]` quand plus rien n'est jouable. Une méthode `main` fait la démonstration sur des exemples : placements, contrainte de liseré, pass, aller-retour fichier et capture.

> Un bug s'était glissé là et nous l'avons corrigé en partie 3 : un placement légal mais aligné sur une seule ligne faisait planter le calcul du bord de Noir, qui supposait toujours deux lignes. Le bord se déduit maintenant de la ligne de la licorne.

4. Intégration au tournoi : le protocole de l'arbitre

Le joueur `escampe.JoueurPuyaubreauRussac` implémente l'interface fournie `IJoueur` et garde à jour un `EscampeBoard` à chaque coup, le nôtre comme celui de l'adversaire reçu par `mouvementEnnemi`. Trois détails ont demandé une adaptation, et deux d'entre eux ont dû être **confirmés en regardant dans le jar de l'arbitre**, qui est obfusqué :

- **Les couleurs** : `IJoueur` raisonne en entiers (`NOIR = 1`, `BLANC = -1`) alors que `EscampeBoard` utilise les chaînes `"noir"` et `"blanc"`.
- **Le pass se note `"E"`, pas `"PASSE"`** : le Javadoc d'`IJoueur` annonce `"PASSE"`, mais le serveur teste bel et bien `move.equals("E")`, et `"PASSE"` n'apparaît nulle part dans le jar. Suivre le Javadoc nous aurait coûté la partie sur coup illégal.
- **La carte des liserés** doit être celle du serveur (cf. Q1).

Placement et coups passent par le même canal (`choixMouvement`/`mouvementEnnemi`) : le premier

`choixMouvement` renvoie un placement, les suivants des coups, la phase se lisant sur `blackPlaced`/`whitePlaced`. En lisant la classe `Solo` fournie, on reconstitue l'ordre des appels :

```
Noir  : choixMouvement(placement) → mvtEnnemi(placement Blanc)
        → mvtEnnemi(1er coup Blanc) → choixMouvement(coup) → ...
Blanc : mvtEnnemi(placement Noir) → choixMouvement(placement)
        → choixMouvement(1er coup, Blanc rejoue) → mvtEnnemi(coup Noir) → ...
```

Comme on rejoue chaque coup sur l'`EscampeBoard` interne dans cet ordre, le joueur au trait reste synchronisé avec l'arbitre sans traitement particulier.

Côté lancement, il faut trois processus, le serveur et deux clients :

```
java -cp escampeobf.jar        escampe.ServeurJeu 1234 1
java -cp Puyaubreau_Russac.jar escampe.ClientJeu escampe.JoueurPuyaubreauRussac localhost
java -cp escampeobf.jar        escampe.ClientJeu escampe.JoueurAleatoire        localhost
```

5. Placement d'ouverture

Le placement ne se rejoue pas, donc autant le soigner. Le constat est venu de l'auto-jeu : une licorne mal posée peut devenir la seule pièce jouable sur le liseré imposé, et se retrouver bloquée, ce qui force des passes à répétition et abandonne l'initiative. Trois principes répondent à ça :

1. **La licorne dans un coin.** Un coin n'a que deux voisines, donc seulement deux cases d'où l'adversaire peut venir la prendre.
2. **Deux murs.** On occupe ces deux voisines avec des paladins, et la licorne devient imprenable tant que les murs tiennent, puisqu'on ne peut pas finir son dernier pas sur une case occupée.
3. **Trois liserés couverts.** Les trois paladins restants se posent sur des cases de liserés 1, 2 et 3 différents. Quel que soit le liseré imposé, il reste une pièce mobile, et on n'a jamais à passer ni à déranger un mur ou la licorne.

Voici les deux dispositions retenues (Blanc prend le bord opposé à Noir) ; nous en avons vérifié la légalité et les trois propriétés ci-dessus :

```
Bord bas  A1/A2/B1/E1/F1/C2          Bord haut  A6/A5/B6/C5/F5/E6
   A B C D E F                          A B C D E F
 2 n . . . . .                        6 N b . . b .
 1 N n . n n n                        5 b . b . . b
 (licorne A1, murs A2/B1,             (licorne A6, murs A5/B6,
  mobiles E1·F1·C2 = liserés 1·2·3)    mobiles C5·F5·E6 = liserés 1·2·3)
```

Nous n'avons pas de bibliothèque d'ouvertures. Elle aurait peu de valeur ici : la contrainte de liseré rend toute séquence pré-calculée caduque dès que le placement adverse diffère du cas prévu, souvent au deuxième coup. Et comme le branchement moyen (~8,9) laisse l'alpha-bêta atteindre la profondeur 12 à 15 d'entrée, le placement fixe ci-dessus suffit à démarrer sur une position saine et reproductible.

6. Moteur de décision

Le choix du coup repose sur un **negamax** avec **élagage alpha-bêta** et **approfondissement itératif** (classe `Moteur`). La recherche travaille sur une copie du plateau, jamais sur l'état réel. Une capture de licorne compte comme une feuille de valeur `WIN - ply`, ce qui pousse à gagner tôt plutôt que tard.

Plusieurs choix tirent la vitesse vers le haut :

- **Coups codés sur un entier** (case = `ligne×6+colonne`, coup = `départ×36+arrivée`), pour ne manipuler aucune chaîne dans la boucle chaude.
- **DFS sur masque de bits** : les 36 cases tiennent dans un `long`, et les ensembles « visité » et « atteignable » sont de simples masques, sans tableau alloué à chaque appel.
- **`make`/`unmake` sans allocation** : un petit jeton suffit à défaire un coup, donc on explore des millions de nœuds sans solliciter le ramasse-miettes.
- **Buffers de coups réservés** à l'avance, un par profondeur.
- **Ordre des coups** : on essaie d'abord toute prise de licorne (coupure immédiate), et on remet en tête le meilleur coup de l'itération précédente.

> Le moteur a sa propre génération de coups en entiers, en parallèle de celle, vérifiée, d'`EscampeBoard` en chaînes. Pour être sûr qu'elles ne divergent pas en silence, le test `VerifMoves` (§9) confronte les deux et exige les mêmes coups et les mêmes états : c'est ce qui nous garantit qu'optimiser n'a pas modifié les règles au passage.

En pratique, le moteur explore de l'ordre de **4 à 5 millions de nœuds par seconde**. En milieu de partie, l'approfondissement itératif atteint **12 à 15 demi-coups** en 6 s, davantage dans les positions étroites. Quand il annonce un gain forcé, la capture a bien lieu dans les parties de contrôle.

7. Heuristique d'évaluation

Le matériel ne bouge pas (paladins imprenables, licornes en place jusqu'à la prise), donc évaluer une position non terminale revient à juger un placement. L'évaluation se fait du point de vue du joueur au trait, à partir de distances de Manhattan, et combine deux idées :

- la **pression d'attaque**, c'est-à-dire la proximité de nos paladins à la licorne adverse, avec un terme de somme (pression d'ensemble) et un terme de minimum (le paladin le plus proche pèse plus lourd) ;
- la **sécurité**, soit l'éloignement des paladins adverses de notre licorne, avec les deux mêmes termes mais de signe opposé.

Avec les poids retenus (2 pour les sommes, 8 pour les minimums) :

```
eval = 2·Σ(10−d_attaque) − 2·Σ(10−d_défense)
       + 8·(10−min d_attaque) − 8·(10−min d_défense)
```

Pour régler ces poids, nous avons fait jouer le moteur contre lui-même et contre le joueur aléatoire fourni, en comparant trois variantes. Avec la somme seule (a), le jeu restait trop diffus et le moteur tardait à concentrer une menace. La somme plus le mininum (b), que nous avons gardée, recentre les paladins vers la licorne adverse grâce au fort poids du minimum et fait monter le taux de capture. L'ajout d'un terme défensif symétrique (c) a été conservé aussi : il évite d'exposer notre licorne sans pénaliser l'attaque. Ce poids élevé sur le minimum traduit une réalité du jeu, où c'est le paladin le plus avancé qui conclut une prise.

> Une limite que nous assumons : faute d'autres IA disponibles avant le tournoi, ces poids sont calés contre l'aléatoire et en auto-jeu, pas contre des joueurs forts. Cela dit, les prises à courte échéance relèvent de la recherche, ce qui rend le joueur solide même avec une évaluation aussi simple.

8. Gestion du temps réel

L'arbitre laisse 300 s par joueur et par partie. Nous travaillons sous une enveloppe interne de **280 s**, soit une vingtaine de secondes de marge. Le budget d'un coup est une fraction du temps restant, bornée des deux côtés :

```
tranche = clamp( temps_restant / 12 , 120 ms , 6000 ms )
```

Diviser le temps restant le fait décroître géométriquement, si bien que le budget ne peut pas s'épuiser, même sur une partie qui s'éternise. Le plafond de 6 s évite de gaspiller du temps en ouverture, le plancher de 120 ms garantit un minimum de réflexion, et un mode « panique » couvre les toutes dernières secondes. Comme la recherche est itérative, le meilleur coup déjà trouvé est disponible dès que la tranche expire, le temps étant relu toutes les 2048 explorations de nœuds.

En mesure (auto-jeu équilibré, plein budget), le coup le plus long approche le plafond, environ **6 s**, et le cumul sur une partie entière plafonne vers **36 s** par joueur, loin des 300 s. Le réglage est prudent et on pourrait l'ouvrir davantage sans risque.

9. Performances et tests

Chaque maillon de la chaîne est contrôlé contre une référence indépendante.

Test	Ce qu'il garantit	Résultat
`VerifMoves`	Génération en entiers (moteur) identique à la génération en chaînes (vérifiée) : mêmes coups, même `make`/`unmake`	3 000 parties · 142 165 positions · 1 281 985 contrôles · **0 divergence**
`RulesTest`	Règles vérifiées directement : nombre de pas = liseré, capture au dernier pas, paladins imprenables, non-traversée, contrainte de liseré, pass forcé, fin, zones de placement	**21 / 21**
Matchs arbitrés vs `JoueurAleatoire`	Protocole de bout en bout (placement, liseré, pass, couleurs) et légalité	**7 / 7 victoires**, 0 coup illégal, 0 exception (les deux couleurs)
Démo IA vs IA (serveur réel)	Partie complète moteur contre moteur, gestion des pass	21 coups, fin propre par capture
`Bench` / `Branching`	Vitesse, profondeur, facteur de branchement	≈ 4–5 M nœuds/s ; profondeur 12–15 ; branchement max 49, moyen ≈ 8,9

Les rôles ne se recouvrent pas : `VerifMoves` montre que le moteur colle à `EscampeBoard`, `RulesTest` que `EscampeBoard` respecte les règles, et les parties arbitrées que l'ensemble dialogue correctement avec le vrai arbitre. Sur toutes les parties jouées, aucun coup illégal n'a été produit.

10. Compilation, exécution et livrables

Le script `build.sh` fabrique dans `dist/` les trois livrables de la version finale :

```
Puyaubreau_Russac.jar    jar exécutable (Main-Class : escampe.ClientJeu)
mainClass                jar:Puyaubreau_Russac.jar
                         clientClass:escampe.ClientJeu
```

```
                          mainClass:escampe.JoueurPuyaubreauRussac
Puyaubreau_Russac.tgz     archive de rendu : répertoire Puyaubreau_Russac/
                          contenant src/escampe/*.java + mainClass + le jar
```

Le jar ne contient que les classes de production (le joueur, le moteur, le plateau et les classes fournies) ; les utilitaires de test (`VerifMoves`, `RulesTest`, `Bench`, `Branching`) restent dehors. Le jeu en multijoueur, humain contre humain ou humain contre notre IA, en local comme à distance, est décrit dans `MULTIJOUEUR.md`.

11. Sources et bibliographie

- L'**énoncé du cours** (Université Paris-Saclay, Polytech APP5, 2025-2026) pour les règles d'Escampe, la carte des liserés (figure 4), l'interface `Partie1` et les classes fournies (`IJoueur`, `ClientJeu`, `Solo`, `Applet`, serveur).
- Des **algorithmes classiques**, comme inspiration et sans copie de code : l'élagage alpha-bêta (Knuth et Moore, *An Analysis of Alpha-Beta Pruning*, 1975), le minimax, le negamax et l'approfondissement itératif (Russell et Norvig, *Artificial Intelligence: A Modern Approach*), ainsi que les représentations par masques de bits et l'ordonnancement de coups (*Chess Programming Wiki*).
- Pour être clairs : nous n'avons recopié aucun programme d'Escampe existant. La seule rétro-ingénierie a porté sur le jar d'arbitre fourni avec le sujet, et uniquement pour confirmer le protocole (le pass `"E"`) et la carte des liserés, deux points que la documentation laissait dans le flou.

12. Conclusion et difficultés rencontrées

Le joueur mène une partie tout seul, dialogue correctement avec l'arbitre, ne joue jamais de coup illégal et tient le temps très largement. Les principaux obstacles ont été les suivants :

- **L'obfuscation du serveur.** Trancher l'ambiguïté du pass (`"E"` et non `"PASSE"`) et confirmer la carte des liserés a demandé de fouiller le jar, sans quoi on perdait sur coup illégal.
- **L'interface obfusquée face à nos sources.** Le joueur aléatoire du jar n'implémente pas notre `IJoueur`, donc les tests contre lui passent par le réseau, où seules des chaînes circulent.
- **L'avantage du trait.** En miroir, Blanc, qui joue le premier, garde l'initiative via la contrainte de liseré ; c'est une propriété du jeu, pas une question de force du moteur.
- **Le réglage de l'heuristique sans sparring-partner**, calé faute de mieux contre l'aléatoire et en auto-jeu.

Si nous devions continuer, plusieurs pistes se présentent : une table de transposition (hachage de Zobrist), une bibliothèque d'ouvertures de placement, un terme de mobilité différentielle dans l'évaluation et une recherche de quiescence sur les menaces de capture.