

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)

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** (couleur noire ou blanche). Les lignes sont numérotées de 1 à 6, les colonnes de A à F. Le but est de **prendre la licorne adverse**.

La règle caractéristique du jeu est une **contrainte de liseré** : la pièce que l'on joue doit partir d'une case dont le liseré est *identique* à celui de la case d'arrivée du coup adverse précédent. Le liseré de la case de départ fixe en outre le nombre de pas (1, 2 ou 3), orthogonaux, sans traverser ni revisiter de case. On ne capture qu'en se posant, au dernier pas, sur la licorne adverse — les paladins sont imprenables. Si un joueur ne peut rien jouer, il passe son tour.

Le déroulement : Noir place ses six pièces sur les deux lignes d'un bord (haut ou bas) ; Blanc fait de même sur le bord opposé ; **Blanc joue le premier coup**. Ce rapport décrit nos choix de modélisation (parties 1 et 2) puis la conception du joueur artificiel pour le tournoi (partie 3), avec les mesures qui justifient nos choix.

2. Analyse des caractéristiques du jeu

Nous reprenons ici les sept questions de la première partie, en les étayant par l'implémentation finalement réalisée.

Q1 — Modélisation d'un état

Le plateau est un tableau `int[6][6]` : `board[ligne][colonne]` avec `ligne 0` = ligne 1 (bas) et `colonne 0` = A. Chaque case contient une constante de pièce (`EMPTY`, `WHITE_LICORNE`, `WHITE_PALADIN`, `BLACK_LICORNE`, `BLACK_PALADIN`). L'état complémentaire, indispensable à la règle, est maintenu hors du plateau :

- `lastTileType` : liseré imposé au coup suivant (`-1` = aucune contrainte) ;
- `currentPlayer` : joueur au trait ;
- `blackPlaced`, `whitePlaced` : fin des phases de placement ;
- `blackRows` : le bord choisi par Noir (en déduit celui de Blanc).

Avantages. Accès $O(1)$ à toute case ; copie immédiate de l'état pour l'arbre de recherche ; sérialisation triviale ; surtout, un schéma `make/unmake` sans aucune allocation (essentiel pour la vitesse, §6).

Inconvénient. La contrainte de liseré est un état séparé qu'il faut maintenir explicitement à chaque coup ; nous l'encapsulons dans `play`.

La carte des liserés est une constante `TILE_MAP` reproduisant 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

Fait vérifié : cette carte est *identique*, case pour case, à celle utilisée en interne par l'arbitre du tournoi — nous l'avons extraite par réflexion de la classe de jeu du serveur fourni. Elle est aussi cohérente avec l'exemple tactique de la figure 6 de l'énoncé. Une carte divergente aurait produit des coups jugés illégaux : ce point était critique.

Q2 — Détection de fin de partie

La partie est finie dès qu'une des deux licornes a disparu du plateau (seul cas de fin, pas de match nul). La vérification est un simple balayage $O(1)$ du plateau (`gameOver`) ; le moteur, lui, détecte la capture directement au moment où elle est jouée (§6).

Q3 — Sources de difficulté et facteur de branchement

Les principales sources de difficulté sont :

- la **contrainte de liseré**, qui limite fortement et variablement la mobilité ;
- la **dépendance entre tours** : la case d'arrivée choisie détermine les pièces que l'adversaire pourra jouer ;
- l'**asymétrie** du plateau (zones riches en liserés triples, donc mobiles, vs zones simples) ;
- le risque de **blocage** d'une pièce, voire d'un joueur (pass forcé).

Facteur de branchement. En première partie nous avons estimé une borne théorique de l'ordre de 120 (6 pièces \times jusqu'à ~ 20 destinations sur liseré triple). La mesure réelle est bien plus basse, car la contrainte de liseré ne laisse jouables que les pièces du bon liseré. Sur 30 000 parties aléatoires simulées (utilitaire `escampe.Branching`) :

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

Le branchement effectif modeste (moyenne < 10) explique qu'une recherche alpha-bêta atteigne des profondeurs élevées en quelques secondes (§6).

Q4 — Coups imparables

Il n'existe pas de coup « imparable » universel garanti dès le départ : la contrainte de liseré peut toujours empêcher l'exécution d'une menace au mauvais moment. En revanche, certaines configurations créent un **zugzwang partiel** où l'adversaire ne peut éviter d'imposer le liseré qui nous arrange — l'énoncé en donne l'exemple (figure 6 : le paladin blanc en C2 prend la licorne en C1 dès que Noir est forcé d'imposer un liseré double). Construire de tels pièges est un axe stratégique ; notre recherche les exploite implicitement quand ils sont à portée d'horizon.

Q5 — Critères pour l'heuristique

Nous avons identifié cinq critères : distance à la licorne adverse, mobilité différentielle, contrôle du liseré imposé, protection de sa propre licorne, et avancée sur le plateau. L'heuristique finalement retenue (§7) s'appuie sur la **proximité des paladins à la licorne adverse** (pression d'attaque) et l'**éloignement des paladins adverses de notre licorne** (sécurité) — les autres critères sont, en pratique, largement pris en charge par la recherche elle-même.

Q6 — Stratégie selon la phase

- **Début (placement)** : irréversible et déterminant. On protège la licorne et on garantit de toujours pouvoir jouer (§5).
- **Milieu** : manœuvre pour construire des menaces sur la licorne adverse tout en contrôlant le liseré imposé ; recherche de zugzwang partiel.

- **Fin** : dès qu'une capture est à portée, le calcul tactique (recherche profonde) prime.

Q7 — Majorant du nombre de coups et gestion du temps

Aucune pièce ne disparaît avant la capture finale ; une partie peut donc théoriquement s'étirer. En bornant le branchement par tour et en comptant quelques dizaines de tours, une borne raisonnable se situe vers 400–600 demi-coups. Pour tenir la contrainte de temps (300 s par joueur et par partie), nous combinons **approfondissement itératif**, **élagage alpha-bêta** et un **budget par coup** dérivé du temps restant (§8).

3. Modélisation : la classe `EscampeBoard`

`EscampeBoard` (≈ 860 lignes) implémente l'interface fournie `Partiel` : `setFromFile` / `saveToFile`, `isValidMove`, `possiblesMoves`, `play`, `gameOver`. Les conventions de l'arbitre sont respectées à la lettre :

- coup régulier `"B1-D1"` ;
- placement `"C6/A6/B5/D5/E6/F5"` (licorne en tête, puis les 5 paladins) ;
- pass `"E"`.

Format de fichier. Six lignes de plateau (bas vers haut), caractères `N/n` (licorne/paladin noir), `B/b` (blanc), `-` (vide), chaque ligne encadrée d'un numéro ; toute autre ligne commence par `%` (commentaire). Nous y ajoutons en commentaires l'état hors-plateau (liseré courant, joueur, bord de Noir) afin que la sauvegarde soit fidèlement rechargeable.

Génération des coups. Depuis une case, on énumère les destinations par un parcours en profondeur (DFS) avec retour arrière : exactement `N` pas (`N` = liseré de départ), cases intermédiaires vides, dernière case vide ou occupée par la licorne adverse (capture). `possiblesMoves` filtre les pièces sur le bon liseré et renvoie `["E"]` si aucun coup n'est possible. Une méthode `main` illustre placements, contrainte de liseré, pass, round-trip fichier et capture.

Bug latent corrigé en partie 3 : un placement légal mais disposé sur une *seule* ligne faisait planter le calcul du bord de Noir (il supposait deux lignes distinctes). Le bord est désormais déduit de façon robuste à partir 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 enveloppe un `EscampeBoard` tenu à jour à chaque coup (le nôtre comme celui de l'adversaire, via `mouvementEnnemi`). Trois points d'adaptation, dont deux **vérifiés par analyse du jar de l'arbitre** car l'infrastructure fournie est obfusquée :

- **Couleurs.** `IJoueur` parle en entiers (`NOIR = 1`, `BLANC = -1`) ; `EscampeBoard` en chaînes `"noir"/"blanc"`.
- **Pass = `"E"`, et non `"PASSE"`.** Le Javadoc d'`IJoueur` indique `"PASSE"`, mais la classe de jeu du serveur teste `move.equals("E")` (et `"PASSE"` n'apparaît nulle part dans le jar). Envoyer `"PASSE"` aurait valu une défaite sur coup illégal.
- **Carte des liserés** identique à celle du serveur (cf. Q1).

Machine à états. Le placement et les coups transitent par le même canal `choixMouvement/mouvementEnnemi`. Le premier appel à `choixMouvement` renvoie donc un *placement*, les suivants des coups ; la phase est détectée via `blackPlaced/whitePlaced`. La séquence (déduite de la classe `Solo`

fournie) est :

```
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) → ...
```

En appliquant chaque coup à l'`EscampeBoard` interne dans cet ordre, le joueur au trait reste naturellement synchronisé avec l'arbitre.

Exécution. Trois processus (serveur + 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 est irréversible : nous l'avons conçu à partir d'un constat issu de l'auto-jeu — une licorne mal placée peut se retrouver *seule pièce jouable et bloquée* sur le liseré imposé, forçant des passes **successifs qui livrent l'initiative à l'adversaire**. Trois principes y répondent :

1. **Licorne dans un coin.** Un coin n'a que deux cases voisines : seulement deux cases d'où l'adversaire peut l'atteindre.
2. **Murs.** On occupe ces deux voisines par des paladins. La licorne devient *incapturable* tant que les murs tiennent (impossible de franchir le dernier pas sur une case occupée).
3. **Couverture des liserés.** Les trois paladins restants sont placés sur des cases de liserés **1, 2 et 3 distincts** : quel que soit le liseré imposé, on dispose toujours d'une pièce mobile — jamais de pass forcé, jamais besoin de déplacer un mur ou la licorne.

Dispositions retenues (légalité et propriétés vérifiées) ; pour Blanc, on joue le bord complémentaire de celui de Noir :

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, mobiles E1·F1·C2 = liserés 1·2·3)		(licorne A6, murs A5/B6, mobiles C5·F5·E6 = liserés 1·2·3)	

6. Moteur de décision

La décision repose sur un **negamax** avec **élagage alpha-bêta** et **approfondissement itératif** (classe `Moteur`). La recherche s'effectue sur une *copie* du plateau, jamais sur l'état réel. Capturer la licorne adverse est traité comme un nœud terminal de valeur **WIN - ply** (gagner vite plutôt que tard).

Astuces de performance.

- **Coups encodés en entier** (case = `ligne×6+colonne`, coup = `départ×36+arrivée`) : aucune chaîne manipulée dans la boucle chaude.
- **DFS sur masque de bits long** : les 36 cases tiennent dans un `long` ; les ensembles « visité » et « atteignable » sont des masques — pas d'allocation de tableau par appel.
- **make/unmake sans allocation** : un petit jeton d'annulation suffit à défaire un coup, ce qui permet d'explorer des millions de nœuds sans pression sur le ramasse-miettes.

- **Buffers de coups pré-alloués**, un par profondeur.
- **Ordonnancement** : tout coup capturant la licorne est essayé en premier (coupure immédiate) ; le meilleur coup d'une itération est remplacé en tête à l'itération suivante.

Cohérence des deux chemins. Le chemin « entier » du moteur double le chemin « chaîne » vérifié de *EscampeBoard*. Pour exclure toute divergence silencieuse entre ces deux implémentations des règles, un test croisé (*VerifMoves*, §9) vérifie qu'ils produisent exactement les mêmes coups et les mêmes états — c'est la garantie qu'optimiser n'a pas changé les règles.

Performance mesurée. Environ **4 à 5 millions de nœuds par seconde**. En milieu de partie, l'approfondissement itératif atteint une profondeur de **12 à 15 demi-coups** en 6 s (davantage dans les positions étroites). Les annonces de gain forcé du moteur se matérialisent bien par une capture effective lors des parties de contrôle.

7. Heuristique d'évaluation

Le matériel étant constant (paladins imprenables, licornes présentes jusqu'à la capture), l'évaluation d'une position non terminale est purement *positionnelle*, exprimée du point de vue du joueur au trait. Elle somme, à partir des distances de Manhattan :

- **Pression d'attaque** : proximité de nos paladins à la licorne adverse — un terme de *somme* (pression globale) et un terme de *minimum* (l'attaquant le plus proche pèse davantage) ;
- **Sécurité** : éloignement des paladins adverses de notre licorne — mêmes deux termes, de signe opposé.

Concrètement, avec les poids retenus (somme = 2, minimum = 8) :

```
eval = 2 * Σ(10 - d_attaque) - 2 * Σ(10 - d_défense)
      + 8 * (10 - min d_attaque) - 8 * (10 - min d_défense)
```

Heuristiques testées et choix. Le réglage s'est fait par auto-jeu déterministe et matchs arbitrés contre le joueur aléatoire fourni. Nous avons comparé : (a) *somme seule* — jeu trop diffus, le moteur tarde à concentrer une menace ; (b) *somme + minimum* (retenue) — le terme minimum, fortement pondéré, oriente nettement les paladins vers la licorne adverse et améliore le taux de capture ; (c) ajout d'un terme défensif symétrique — conservé, il évite d'exposer notre licorne sans nuire à l'attaque. Le fort poids du terme minimum reflète que, dans ce jeu, c'est l'attaquant *le plus avancé* qui décide d'une prise.

Limite assumée. Faute d'adversaires IA tiers disponibles avant le tournoi, ces poids sont validés contre l'aléatoire et en auto-jeu, non contre d'autres joueurs forts. Les tactiques de capture à court terme sont, elles, gérées par la recherche, ce qui rend le joueur robuste même avec une évaluation positionnelle simple.

8. Gestion du temps réel

La limite de l'arbitre est de 300 s par joueur et par partie. Nous nous fixons une **enveloppe interne de 280 s** (≈ 20 s de marge). Le budget alloué à un coup est une fraction du temps restant, bornée :

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

La division par le temps restant décroît géométriquement : le budget ne peut **jamais** être épuisé,

même sur une partie très longue. Le plafond de 6 s évite de surinvestir en ouverture ; un plancher de 120 ms garantit un minimum de réflexion ; un mode « panique » sécurise les toutes dernières secondes. L'approfondissement itératif rend le meilleur coup déjà trouvé dès que la tranche expire (le temps est contrôlé toutes les 2048 explorations de nœuds).

Mesures (auto-jeu équilibré, plein budget) : temps **maximal par coup** $\approx 6,0$ s (le plafond), **cumul maximal** ≈ 36 s par joueur sur une partie complète — très loin des 300 s. Le réglage est volontairement conservateur et pourrait être augmenté sans risque.

9. Performances et tests

Notre démarche de validation est empirique et redondante : chaque maillon est contrôlé contre une référence indépendante.

Test	Ce qu'il garantit	Résultat
<code>VerifMoves</code>	Chemin entier (moteur) \equiv chemin chaîne (vérifié) : mêmes coups, même <code>make/unmake</code>	3 000 parties · 142 165 positions · 1 281 985 contrôles · 0 divergence
<code>RulesTest</code>	Règles directes : pas = liseré, capture au dernier pas, paladins imprenables, non-traversée, contrainte de liseré, pass forcé, fin, zones de placement	21 / 21
Matchs arbitrés vs <code>JoueurAleatoire</code>	Protocole de bout en bout (placement, liseré, pass, couleurs), 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
<code>Bench / Branching</code>	Vitesse, profondeur, facteur de branchement	$\approx 4\text{--}5$ M nœuds/s ; profondeur 12–15 ; branchement max 49 / moyen $\approx 8,9$

La séparation des rôles est délibérée : `VerifMoves` prouve que le moteur \equiv `EscampeBoard` ; `RulesTest` prouve que `EscampeBoard` respecte les règles ; les parties arbitrées prouvent que le tout dialogue correctement avec l'arbitre réel. Aucun coup illégal n'a été produit sur l'ensemble des parties jouées.

10. Compilation, exécution et livrables

Le script `build.sh` produit 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
```

Seules les classes de production entrent dans le jar (le joueur, le moteur, le plateau et les classes fournies) ; les utilitaires de test (`VerifMoves`, `RulesTest`, `Bench`, `Branching`) en sont exclus. Le jeu

en multijoueur (humain contre humain, ou humain contre notre IA, en local ou à distance) est documenté à part dans `MULTIJOUEUR.md`.

11. Sources et bibliographie

- **Énoncé du cours** (Université Paris-Saclay, Polytech APP5, 2025-2026) : règles d'Escampe, carte des liserés (figure 4), interface `Partiel`, et classes d'infrastructure fournies (`IJoueur`, `ClientJeu`, `Solo`, `Applet`, `serveur`).
- **Algorithmes classiques**, à titre d'inspiration et sans copie de code : élagage alpha-bêta (Knuth & Moore, *An Analysis of Alpha-Beta Pruning*, 1975) ; minimax, negamax et approfondissement itératif (Russell & Norvig, *Artificial Intelligence: A Modern Approach*) ; techniques de représentation par masques de bits et d'ordonnancement de coups (*Chess Programming Wiki*).
- **Déclaration**. Aucun programme d'Escampe externe n'a été recopié. La seule rétro-ingénierie effectuée porte sur le jar d'arbitre *fourni avec le sujet*, dans le seul but de confirmer le protocole (pass `"E"`) et la carte des liserés — points sur lesquels la documentation était ambiguë.

12. Conclusion et difficultés rencontrées

Le joueur conduit une partie de façon autonome, dialogue correctement avec l'arbitre, ne produit jamais de coup illégal et respecte très confortablement la contrainte de temps. Les principales difficultés ont été :

- **L'obfuscation du serveur** : lever l'ambiguïté du pass (`"E"` vs `"PASSE"`) et confirmer la carte des liserés a demandé une analyse du jar — étape décisive pour ne pas perdre sur coup illégal.
- **L'interface obfusquée vs nos sources** : le joueur aléatoire du jar n'implémente pas notre `IJoueur` ; les tests contre lui passent donc par le réseau (seules des chaînes circulent).
- **L'avantage du trait** : en miroir, Blanc (premier à jouer) conserve l'initiative via la contrainte de liseré — propriété du jeu, indépendante de la force du moteur.
- **Le réglage de l'heuristique sans adversaires** : validé contre l'aléatoire et en auto-jeu.

Pistes d'amélioration : table de transposition (hachage de Zobrist), bibliothèque d'ouvertures de placement, terme de mobilité différentielle dans l'évaluation, et recherche de quiescence sur les menaces de capture.