TME1-4. Statistique et informatique

BATAILLE NAVALE: ANALYSE PROBABILISTE

Mattéo Pietri
matteo.pietri@etu.sorbonne-universite.fr
Sara Castro López
sara.castro-lopez@etu.sorbonne-universite.fr

Contents

1 Introduction

Dans le cadre de ce projet de statistique et d'informatique, nous avons réalisé une étude probabilistique du jeu de la bataille navale. Le but principal de notre travail était d'analyser le jeu sous un angle combinatoire, d'en proposer une modélisation pour optimiser les chances de succès d'un joueur, puis d'étudier une variante plus réaliste du jeu. Tout au long du projet, nous avons introduit des libertés créatives tout en respectant les lignes directrices du sujet, afin d'améliorer à la fois l'optimisation et la compréhension du processus.

2 La philosophie du jeu

Le jeu de la bataille navale se déroule sur une grille de 10 cases par 10 cases, où l'adversaire place cinq types de bateaux, de tailles différentes :

- **Porte-avions** : 5 cases
- **Croiseur** : 4 cases
- **Contre-torpilleur** : 3 cases
- **Sous-marin** : 3 cases
- **Torpilleur** : 2 cases

Les bateaux peuvent être positionnés soit horizontalement, soit verticalement, sans chevauchement, et leur placement est gardé secret. L'objectif du joueur est de couler tous les bateaux de l'adversaire en un minimum de coups. Chaque tir vise une case, et la réponse varie selon le statut de la case :

- **Vide** : Il n'y a aucun bateau sur la case.
- **Touché** : Une partie d'un bateau est sur la case.
- **Coulé** : Toutes les cases d'un bateau ont été touchées et les cases correspondantes sont révélées.

Nous avons établi quelques régles par rapport à la version originale du jeu et nous avons décidé quelques aspects liés à l'implementation du code :

- **Placement par défaut des bateaux**: Les bateaux sont initialement placés horizontalement de droite à gauche ou verticalement de bas en haut selon la case sélectionnée, pour simplifier le processus de placement.
- **Modularisation du code**: En utilisant une philosophie orientée objet, nous avons créé des classes distincts pour la grille, les bateaux, la bataille et le joueur, ainsi que les classes test. Cela a permis une meilleure organisation et facilité la compréhension et l'évolution du code.
- **Fenêtre graphique**: On a renforcé notre visualisation en utilisant le module Tkinter et Matplotlib. Matplotlib a été utile, car elle offre une représentation visuelle très bonne de notre information (par exemple, la représentation de la grille ou des histogrammes). Afin d'interagir avec la grille, nous avons choisi de faire un mouse_clicked_event. De cette façon, l'utilisateur du programme peut cliquer sur la grille pour sélectionner la position qui l'intéresse. Vous pouvez tester la jouée manuelle (et la méthode affiche) en exécutant le main.

3 Combinatoire du jeu

3.1 Méthodes d'estimation et de calcul des configurations possibles

Pour maximiser l'efficacité des stratégies que nous allons étudier, il est essentiel d'analyser la combinatoire associée à ce jeu. L'objectif est d'examiner et de comprendre les différentes configurations possibles pour le placement des bateaux. Ce problème présente une complexité notable en raison de l'interdépendance des événements. En d'autres termes, si un bateau est placé à une position x, cela réduit l'espace disponible sur la grille et modifie l'ensemble des configurations possibles pour le placement des bateaux suivants. Ainsi, la fonction de probabilité du placement d'un bateau dépend des positions déjà occupées, ce qui implique une réduction progressive des degrés de liberté pour chaque nouveau placement.

Tout d'abord, nous avons calculé la borne supérieure simple du nombre de configurations possibles pour tous les bateaux présents dans le jeu (c'est-à-dire, nous cherchons à positionner les cinq bateaux sur une même grille). Nous n'avons pas consideré dans ce cas le chevauchement entre bateaux; on l'etudiera plus en detail dans les calculs suivants. Sur une grille vide, nous avons trouvé que chaque bateau a :

$$\begin{aligned} \text{positions\_possibles} &= 2 \times (N \times (N - L + 1)) \\ \text{où} \quad N &= \text{taille de la grille}, \quad L = \text{taille du bateau} \end{aligned} \tag{1}$$

Car il y a le même nombre de positiones horizontales (N * (N - L +1) que verticales posibles (N * (N - L +1). Pour mécaniser ce calcul, nous avons créé une fonction **nb_bateau** dans la classe **Bateau**, qui cherche précisément ce nombre de configurations pour un bateau donné sur une grille vide. Nous pouvons constater que notre estimation théorique et expérimentale sont identiques.

	Longueur bateau	**Cases occupées horizontalement**	**Cases occupées verticalement**	**Façons de placement théorique**	**Façons de placement (nb_bateau)**
Porte-avion	5	60	60	120	120
Croiseur	4	70	70	140	140
Contre-torpilleur et sous-marin	3	80	80	160	160
Torpilleur	2	90	90	180	180

Pour calculer le nombre de configurations possibles avec la liste de cinq bateaux[1], nous nous baserons sur le principe multiplicatif de la combinatoire, qui indique que si un événement peut se produire de `a` manières et qu'un autre événement indépendant (rappelons que nous considérons qu'il n'y a pas de chevauchement entre les bateaux) peut se produire de `b` manières, alors le nombre total de combinaisons des deux événements est `a*b`. Par conséquent:

$$\text{configurations\_possibles\_5\_bateaux} = 120 \times 140 \times 160 \times 160 \times 180 = 77,414,400,000 \tag{2}$$

Cependant, ce cas s'éloigne de la réalité ; comme nous l'avons bien dit, les événements de placement des bateaux sont dépendants entre eux, car chaque fois que nous plaçons un bateau, nous réduisons l'espace de notre grille et éliminons de nouvelles possibilités de placement pour le bateau suivant. Nous avons essayé de créer une fonction appelée **nb_liste_bateaux** dans la classe **Grille** où, étant donnée une liste de bateaux, les configurations possibles pour cette liste sont calculées. Dans cette fonction, nous suivons la philosophie d'un algorithme branch and bound, où tout l'arbre de recherche est parcouru de manière exhaustive afin de trouver une par une toutes les configurations possibles.

[1]Sur le code, vous pouvez voir la fonction **nb1_liste_bateaux** (classe **Grille**), qui automatise ce calcul et peut le faire avec n'importe quelle liste de bateaux.

En testant cet algorithme, nous avons pu constater qu'il fonctionnait bien avec des listes de 1, 2 et 3 bateaux. Cependant, il était évident qu'au fur et à mesure que nous augmentions le nombre de bateaux, le temps d'exécution de la fonction augmentait considérablement en raison de la nature exponentielle d'un algorithme récursif. En utilisant le module Time, nous avons mesuré ce temps. Ainsi, pour une liste d'un bateau, le programme a pris pratiquement 0 secondes pour s'exécuter de moyen (0,002 s). Pour 2 bateaux, 0,11 secondes, tandis qu'avec 3 bateaux, il a déjà fallu 15,98 secondes.

Si nous calculons le facteur de croissance approximatif (puisque nous ne savons pas comment la fonction croît exactement) entre le temps de la liste de 1, 2 et 3 bateaux, nous obtenons que :

$$\text{Facteur de croissance 1 - 2} = \frac{\text{Temps moyen liste avec 2 bateaux}}{\text{Temps liste avec 1 bateaux}} = \frac{0{,}12}{0.001} \approx 120$$

$$\text{Facteur de croissance 2 - 3} = \frac{\text{Temps moyen liste avec 3 bateaux}}{\text{Temps liste avec 2 bateaux}} = \frac{15.98}{0.11} \approx 145.27$$

Aplicant la moyenne de ces deux facteurs (132,635), nous estimons que :

- Le temps pour une liste de 4 bateaux sera d'environ 2119,5 secondes (environ 35,3 minutes).
- Le temps pour une liste de 5 bateaux sera d'environ 281119,88 secondes (environ 93,68 heures).

Cela signifie que, en raison de la nature de la fonction, calculer les configurations à partir de 3 bateaux devient une tâche trop coûteuse en termes de ressources informatiques et de temps.

Nous disposons de deux fonctions : l'une qui calcule de manière approximative le nombre de configurations possibles pour une liste donnée de bateaux, et l'autre qui effectue ce même calcul de manière récursive. Bien que la première soit très rapide, elle ne prend pas en compte le chevauchement des bateaux dans ses calculs. En revanche, la seconde offre une précision de 100%, car elle explore toutes les solutions possibles, mais le calcul devient extrêmement coûteux à mesure que le nombre de bateaux augmente.

Nous avons aussi créé une fonction (**nb2_liste_bateaux**)[2] qui nous permet d'obtenir une meilleure approximation en combinant précision et temps, tout en simulant la contrainte de chevauchement des bateaux. Au lieu de compter les différentes façons de positionner chaque bateau sur une grille vide, nous plaçons successivement chaque bateau de manière aléatoire dans la grille, ce qui génère automatiquement des contraintes de positionnement lors de l'ajout des bateaux. Nous joignons ci-dessous les résultats des 3 expériences.

Nombre de bateaux (L = longueur du bateau)	**nb_liste_bateaux (nombre de grilles)**	**nb1_liste_bateaux (sans chevauchement)**	**nb2_liste_bateaux (plus approximé)**
1 (L1 = 5)	120	120	120
2 (L2 = 4)	14400	16800	14880
3 (L3 = 3)	1850736	2688000	1992600
4 (L4 = 3)	*-timeout-*	430080000	198315000
5 (L5 = 2)	*-timeout-*	77414400000	29467294560

[2]Nous pouvons corroborer avec cette étude de C. Liam Brown (https://cliambrown.com/battleship/) que cette approximation est assez précise par rapport au nombre réel de configurations possibles (30 093 975 536)

3.2 Simulation probabiliste des configurations de grilles

Lorsque nous parlons du nombre de configurations possibles, nous pouvons également évoquer, de manière analogue, le nombre de grilles que l'on pourrait configurer pour une liste de bateaux donnée. Si nous considérons toutes les grilles comme équiprobables, nous pouvons en déduire que, si l'on choisit une grille au hasard, la probabilité d'obtenir une grille spécifique est inversement proportionnelle au nombre total de configurations possibles. Ainsi, en plaçant un seul bateau de longueur 5, nous obtiendrions que la probabilité d'obtenir une grille concrète serait de 1/120.

Nous avons développé la fonction prob_grille dans la classe Grille dont l'objectif est de simuler la génération[3] aléatoire de grilles, en tenant compte des contraintes imposées par la position des bateaux. Cette fonction prend en entrée une grille spécifique, représentant une configuration de bateaux sur une grille de jeu, et génère de manière aléatoire différentes configurations possibles jusqu'à ce qu'une grille identique[4] à celle passée en paramètre soit obtenue. L'idée derrière cette approche est de modéliser et d'estimer la probabilité d'obtenir une configuration donnée parmi toutes les configurations possibles. Avec cela, nous avons pu constater que, tout comme dans la fonction nb_liste_bateaux, à mesure que le nombre de bateaux augmente, le nombre de tentatives nécessaires pour trouver la grille donnée augmente également, rendant l'exécution de cette fonction rop coûteuse à partir de 3 bateaux.

Nous affichons les résultats de nos tests pour observer le comportement[5] :

```
Bateau L=5 :
    Tentatives pour trouver une grille donnée :  51
    Tentatives pour trouver une grille donnée :  427
    Tentatives pour trouver une grille donnée :  85
    Tentatives pour trouver une grille donnée :  111
    Tentatives pour trouver une grille donnée :  75
Bateaux L=5, L=4 :
    Tentatives pour trouver une grille donnée :  19480
    Tentatives pour trouver une grille donnée :  22257
    Tentatives pour trouver une grille donnée :  1432
    Tentatives pour trouver une grille donnée :  6295
    Tentatives pour trouver une grille donnée :  133
Bateaux L=5, L=4, L=3 :
    Timeout atteint après 10 secondes et 299582 tentatives.
    Tentatives pour trouver une grille donnée :  -1
    Tentatives pour trouver une grille donnée :  57763
    Timeout atteint après 10 secondes et 291518 tentatives.
    Tentatives pour trouver une grille donnée :  -1
    Timeout atteint après 10 secondes et 271575 tentatives.
    Tentatives pour trouver une grille donnée :  -1
    Timeout atteint après 10 secondes et 298795 tentatives.
    Tentatives pour trouver une grille donnée :  -1
```

[3] Pour générer des grilles, nous avons créé une fonction genere(), qui place des bateaux sur une grille vide.

[4] Pour comparer, nous avons utilisé la fonction eq(), qui compare case par case les cases de la grille afin de vérifier qu'elles sont toutes identiques à celles de la grille passée en paramètre. Si c'est le cas, la fonction renverra False.

[5] Il convient de mentionner que le timeout pour la fonction a été fixé à 10 secondes. C'est pourquoi nous pouvons voir que, pour 3 bateaux, il est souvent impossible de trouver une grille identique dans ce délai.

4 Modélisation probabiliste du jeu

Nous utiliserons la classe Joueur pour développer diverses stratégies de jeu. Nous débuterons par une approche arbitraire, puis évoluerons vers des stratégies plus élaborées, fondées sur des méthodologies plus consensuelles et exploitant le calcul des probabilités.

4.1 Version aléatoire

Nous présenterons la stratégie la plus aléatoire et la moins adaptée pour gagner au jeu. Dans cette section, nous l'étudierons en détail, d'un point de vue théorique et pratique.

4.1.1 Espérance théorique du nombre de coups nécessaires pour couler tous les bateauxs

Considérons une grille de $N = 100$ cases, avec m cases occupées par des bateaux et les $N - m$ restantes représentant l'eau. On effectue un tir aléatoire de k cases et on souhaite calculer combien de tirs sont nécessaires pour toucher toutes les cases des bateaux.

La probabilité de toucher exactement n cases contenant des bateaux en sélectionnant k cases au hasard suit une distribution hypergéométrique. La probabilité que Y représente le nombre de cases de bateaux touchées est donnée par la formule suivante :

$$p(Y = n) = \frac{\binom{m}{n}\binom{N-m}{k-n}}{\binom{N}{k}} = \frac{\binom{17}{n}\binom{83}{k-n}}{\binom{100}{k}}$$

Cette expression permet de calculer la probabilité d'obtenir exactement n cases de bateaux parmi k cases tirées.

Pour estimer le nombre total de coups k nécessaires pour toucher les 17 cases de bateaux, nous pouvons utiliser une approche en estimant directement la probabilité cumulative de tirer toutes les cases de bateaux parmi k tentatives :

$$p(X \leq k) = \sum_{n=0}^{17} \frac{\binom{m}{n}\binom{N-m}{k-n}}{\binom{N}{k}}$$

Cela veut dire que, si $p(X \leq k) \approx 1$, cela signifie qu'il est probable que toutes les cases de bateaux aient été touchées.

La probabilité exacte pour avoir exactement k coups pour toucher les 17 cases est ensuite donnée par la différence des probabilités cumulatives :

$$p(X = k) = p(X \leq k) - p(X \leq k - 1)$$

Nous savons que $p(X \leq 16) = 0$, puisqu'il est impossible de toucher les 17 cases de bateaux avec moins de 17 tirs, et que $p(X > 100) = 0$, car le jeu ne peut pas excéder 100 tirs.

Calcul de l'espérance : L'espérance mathématique du nombre de coups X nécessaires pour toucher toutes les cases de bateaux est donnée par la formule suivante :

$$E(X) = \sum_{k=17}^{100} k \cdot p(X = k)$$

Où $p(X = k)$ est la probabilité d'avoir exactement k coups pour toucher les 17 cases de bateaux. En utilisant cette relation, les simulations montrent que la valeur de l'espérance pour un jeu de bataille navale aléatoire est d'environ :

$$E(X) \approx 95,42$$

Cela signifie qu'en moyenne, 95,42 coups sont nécessaires pour couler tous les bateaux lorsqu'on tire de manière complètement aléatoire.

Avec ces informations, nous pouvons tracer un graphique de la distribution de probabilité théorique dans notre code. L'image suivante a été réalisée avec **test_esperance_theorique** et **test_distribution_theorique**, tous deux issus de la classe **TestAleatoire**.

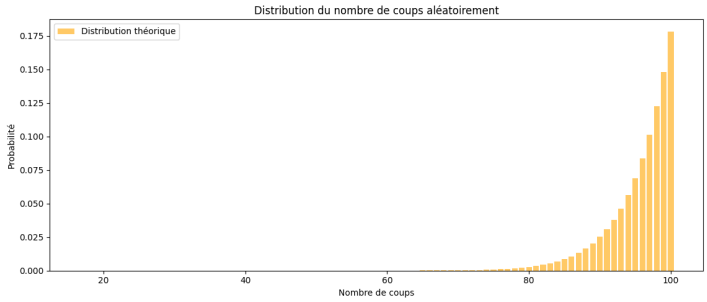

Figure 1: Distribution de probabilité. Nous pouvons voir qu'elle suit une loi exponentielle.

4.1.2 Modélisation de la jouée aléatoire

Nous avons créé une méthode dans la classe Joueur, appelée **jouee_aleatoire()**. Elle s'occupe de simuler des tirs sur des positions aléatoires de la grille jusqu'à ce que tous les bateaux ennemis soient détruits (**self.bataille.victoire()**). De plus, nous avons également créé la méthode **simuler_aleatoire()**, qui appelle simplement jouee_aleatoire un certain nombre de fois (par défaut 1000). De cette manière, nous obtenons suffisamment de données pour tracer la distribution empirique de probabilité aléatoire.

Avec la fonction **test_comparaison_theorique_empirique()** de la classe TestAleatoire, nous pouvons comparer les résultats théoriques et empiriques, ainsi que consulter l'espérance empirique issue de ces simulations. Dans ce cas, nous pouvons constater que, dans cette stratégie, nos données théoriques coïncident avec nos données empiriques.

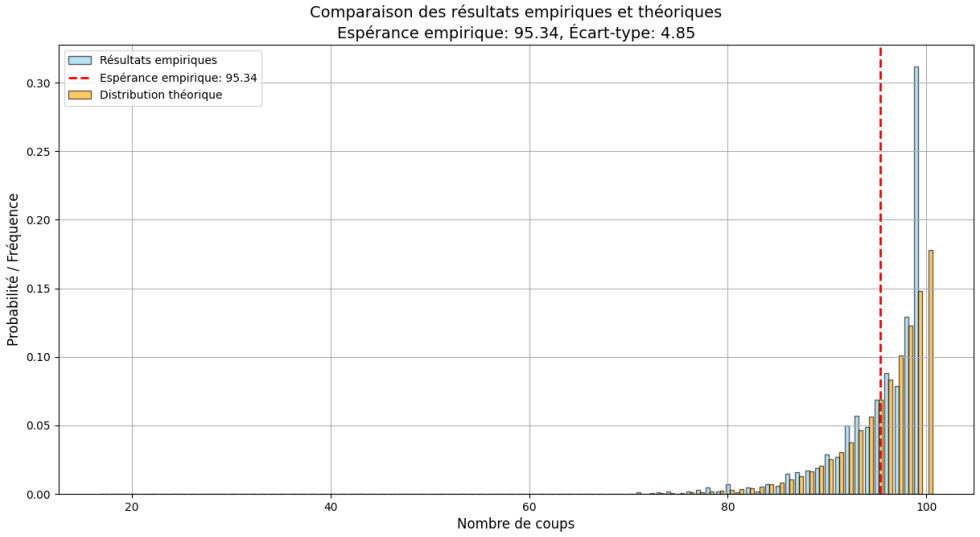

Figure 2: Comparaison aléatoire

4.2 Version heuristique

Afin d'améliorer notre stratégie précédente, nous avons décidé d'adopter une approche plus réfléchie. Dans cette section, nous allons explorer une stratégie heuristique, où les tirs seront effectués de manière aléatoire jusqu'à ce qu'un bateau soit touché. Une fois un bateau repéré, les tirs seront concentrés sur les cases adjacentes pour tenter de le couler.

4.2.1 Modélisation de la jouée heuristique

Avec le graphique que nous présentons ci-dessous, nous pouvons constater que la stratégie heuristique est plus précise en termes de nombre de tirs, avec une espérance approximative de 66,08. Nous avons fait ce graphique en utilisant la fonction **test_comparaison_aleatoire_heuristique** de la classe **TestHeuristique**.

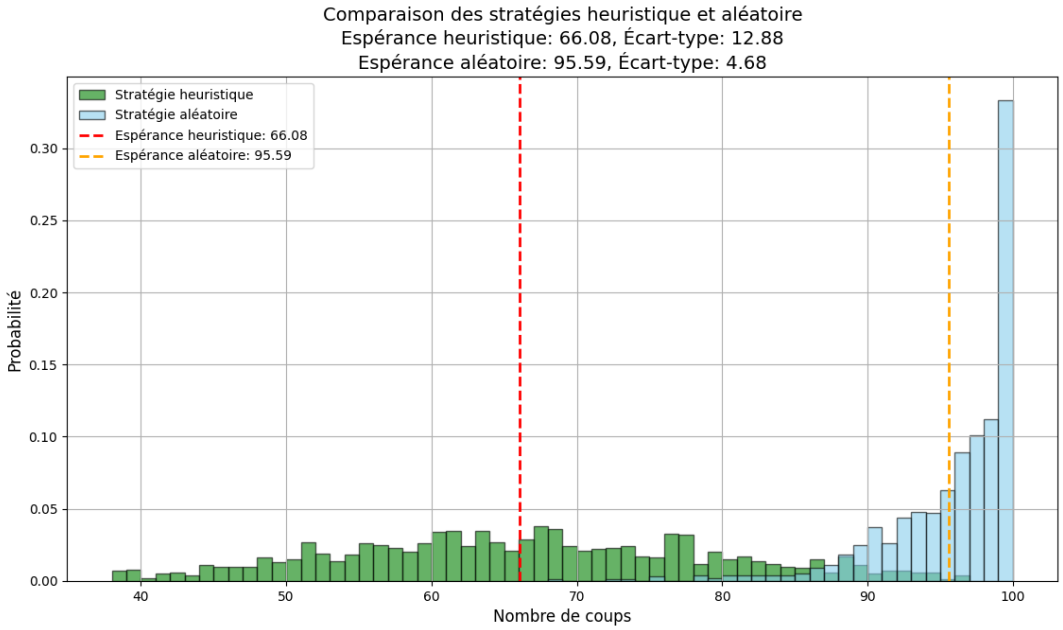

Figure 3: Comparaison de la distribution entre la jouée heuristique et aléatoire. Simulations avec 1000 jouées

4.3 Version probabiliste simplifiée

Dans cette section, nous explorons une nouvelle variante de la stratégie probabiliste pour le jeu de *bataille navale*. Cette variante utilise une **matrice de probabilités** pour déterminer quelle case frapper à chaque tour, en ajustant dynamiquement ces probabilités au fur et à mesure que le jeu progresse.

- **Matrice de probabilités** : Nous initialisons une matrice de probabilités, où chaque cellule représente la probabilité de contenir un navire. Ces probabilités sont ajustées dynamiquement à mesure que des tirs sont effectués et que de nouvelles informations sont recueillies.
- **Diminution de la probabilité** : Chaque fois qu'un tir est effectué sur une case, cette case reçoit une probabilité de **0**, indiquant qu'elle a déjà été investiguée et ne peut donc pas contenir de navire.
- **Augmentation dans les cases adjacentes** : Si le tir a été réussi, c'est-à-dire si un navire a été touché, les cases adjacentes à la case touchée augmentent leur probabilité. Cela reflète l'hypothèse que d'autres parties du navire pourraient se trouver à proximité de la case touchée.

- **Distribution initiale centrée** : Nous avons décidé de commencer avec une **distribution centrée** des probabilités afin d'optimiser le nombre de tirs nécessaires dans la simulation.

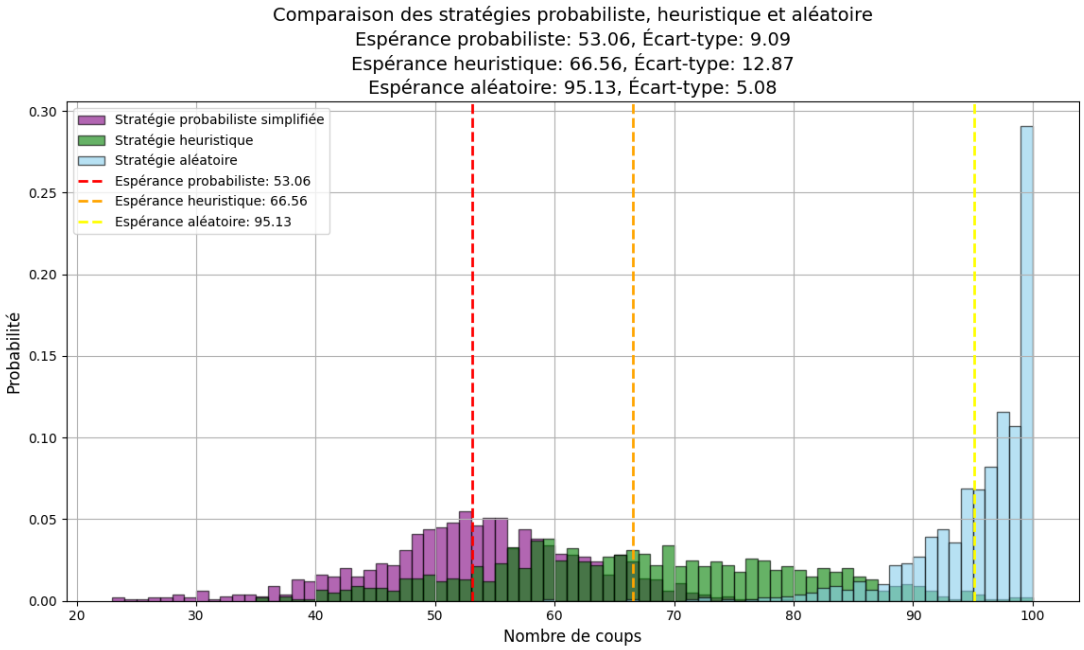

Figure 4: Comparaison de la distribution entre les 3 jouées. Simulations avec 1000 jouées

5 Senseur imparfait : à la recherche de l'USS Scorpion

5.1 Loi de Y_i

La variable Y_i suit une loi de Bernoulli avec une probabilité π_i que l'objet soit présent dans la cellule i :

$$\mathbb{P}(Y_i = 1) = \Pi_i \quad \text{et} \quad \mathbb{P}(Y_i = 0) = 1 - \pi_i$$

5.2 Loi de $Z\_i|Y\_i$

Si l'objet est dans la cellule i, la détection par le senseur est conditionnée par p_s :

$$\mathbb{P}(Z_i = 1|Y_i = 1) = p_s \quad \text{et} \quad \mathbb{P}(Z_i = 0|Y_i = 1) = 1 - p_s$$

Si l'objet n'est pas dans la cellule i :

$$\mathbb{P}(Z_i = 1|Y_i = 0) = 0 \quad \text{et} \quad \mathbb{P}(Z_i = 0|Y_i = 0) = 1$$

5.3 Probabilité de Non-détection

Supposons que l'objet se trouve dans la cellule k mais que la détection n'a pas fonctionné ($Z\_k = 0$). Nous souhaitons calculer la probabilité de cet événement :

$$\mathbb{P}(Z_k = 0|Y_k = 1) = 1 - p_s$$

Cela signifie que bien que l'objet soit dans la cellule, il y a une chance de $1 - p\_s$ que le senseur échoue à le détecter.

5.4 Mise à jour Bayésienne

Lorsqu'une cellule k est sondée et qu'il n'y a pas de détection ($Z\_k = 0$), la probabilité que l'objet se trouve dans cette cellule doit être mise à jour en utilisant la règle de Bayes. La nouvelle probabilité a posteriori que l'objet soit dans la cellule k est :

$$\pi'_k = \frac{\mathbb{P}(Z_k = 0|Y_k = 1) \cdot \pi_k}{\mathbb{P}(Z_k = 0)}$$

Cette mise à jour diminue la probabilité que l'objet se trouve dans la cellule k.

5.5 Mise à jour des autres cellules

Après avoir sondé la cellule k et trouvé $Z\_k = 0$, les probabilités a priori des autres cellules doivent être ajustées. Pour les autres cellules $i \neq k$, la probabilité devrait etre augmenter proportionnellement :

$$\pi'_i = \frac{\pi_i}{1 - \pi_k}$$

Cela permet de redistribuer la probabilité sur les autres cellules.

5.6 Algorithme de Recherche

L'algorithme utilisé pour la recherche de l'objet suit les étapes suivantes en lancant `TestObjetRecherchePerdu.py` :

1. Initialiser les probabilités a priori pour chaque cellule i de la grille $(\pi_1, \pi_2, \ldots, \pi_N)$
2. Choisir une cellule à sonder en fonction de la distribution des probabilités actuelles.
3. Si $Z_i = 1$, l'objet est trouvé. Fin de l'algorithme.
4. Si $Z_i = 0$, mettre à jour la probabilité de la cellule sondée et recalculer les probabilités des autres cellules à l'aide de la mise à jour bayésienne.
5. Répéter les étapes jusqu'à ce que l'objet soit détecté.

Cette approche probabiliste permet d'améliorer significativement la recherche d'un objet perdu en prenant en compte les détections imparfaites. En utilisant un algorithme bayésien inspiré de {ObjetRecherchePerdu.py}, nous maximisons les chances de localisation après chaque échec de détection.

6 Conclusion

Ce projet a révélé la complexité de la bataille navale. Un jeu où la stratégie fondée sur les probabilités peut réduire la dépendance à la chance et améliorer les chances de succès. Nous avons découvert que le calcul des probabilités, bien qu'il ne garantisse pas la victoire, permet d'optimiser chaque coup pour augmenter la probabilité de trouver les bateaux adverses. Cela a démontré l'importance d'intégrer des calculs précis et des stratégies réfléchies, soulignant comment des concepts mathématiques peuvent être appliqués de manière pratique, offrant un nouvel éclairage sur le potentiel des méthodes probabilistes dans les jeux de stratégie et au-delà.