2022-2023

LU3IN005 – PROBAS, STATS ET INFO

PROJET 1 : BATAILLE NAVALE

SALIOU BARRY & TIMO BRISSET

Introduction

Ce petit projet a comme objectif d'étudier le jeu de la bataille navale d'un point de vue probabiliste. Il s'agira dans un premier temps d'étudier la combinatoire du jeu, puis de faire une modélisation du jeu afin d'optimiser les chances d'un joueur de gagner, et enfin d'étudier une variante du jeu plus réaliste. Le jeu de la bataille navale se joue sur une grille de 10 cases par 10 cases sur laquelle sont positionnés 5 bateaux de tailles respectives 5, 4, 3, 3 et 2.

1- Combinatoire du jeu

Dans cette partie, nous nous intéressons au dénombrement du nombre de grilles possibles dans différentes conditions pour appréhender la combinatoire du jeu.

Nous disposons d'une grille de taille de 10x10 cases et de 5 bateaux de taille 5, 4, 3, 3, 2, le nombre de case que peut occuper tous les bateaux une fois placés sur la grille est de 17 cases.

On veut trouver une borne supérieure du nombre de configurations possibles de placer 5 bateaux parmi 100cases, une façon simple de résoudre ce problème est de chercher le nombre de configurations possibles de chaque bateau sur une grille vide.

Avec cette approche, on peu avoir une borne supérieure mais qui est loin d'être le nombre de configurations exact.

Théoriquement, sur une ligne de la grille, un bateau de taille n dispose $(10 - n + 1)$ configurations possibles. Etant donné que la grille contient 10 lignes et qu'un bateau peut être placé de deux manières différentes donc on en déduit qu'il existe $2 \times 10 \times (10 - n + 1)$ façons de positionner un bateau de taille n sur la grille vide.

Taille du bateau	**Nombre de configurations possibles**
5	120
4	140
3	160
3	160
2	180

Une Borne supérieur possible est donc : 120*140*160*160*180 configurations

Nous avons implémenté une fonction `nb_possiblite(self, bateau)` qui permet de donner le nombre de configuration possibles d'un bateau, cette fonction parcours toute la grille et vérifie pour chaque case si le bateau donné en paramètre peut être placé de façon horizontale et verticale et à la fin nous renvoie le nombre total de configuration possible sur la grille vide. **Le résultat retourné par la fonction est équivalent à notre étude théorique précédemment réalisé.**

A présent au lieu de dénombrer le nombre de configuration pour un seul bateau nous allons le faire pour une liste complète des cinq bateaux.

Pour calculer le nombre de façons de placer une liste de bateaux sur une grille vide, nous avons implémenté une fonction récursive `nb_place_possible(self, liste)` qui prend une liste de bateau en paramètre et parcours toute la grille, pour chaque case, on commence par placer le premier bateau de la liste sur la grille vide et on rappelle notre fonction avec une liste sans son premier bateau de manière à tester toutes les combinaison possible, et a la fin le bateau placé est supprimé de la grille avec la fonction `del_ship(self, bateau, position, direction)`, la fonction se termine une fois que toutes les combinaison ont été réalisé.
Avec une liste de 3 bateaux, cette fonction tourne environ **3min** avant de se terminer.

Le calcul du nombre de grilles avec 4 bateaux est trop gourmand en ressources, il est impossible de faire de même avec la liste complète de bateaux. Il nous faut donc trouver une autre méthode pour estimer le nombre de grilles possibles. Pour ce faire, nous allons étudier le lien entre le nombre de grilles possibles et la probabilité de tirer une grille donnée.

Lien entre le nombre de grilles et la probabilité de tirer une grille donnée :

En considérant toutes les grilles équiprobables et ω l'événement élémentaire de « tirer une grille donnée »

On pose G = le nombre de grilles totales

Etant donné que toutes les grilles sont équiprobables, donc la probabilité de tirer une grille au hasard est :

$$p(\omega) = \frac{1}{G}$$

On en déduit donc que :

$$G = \frac{1}{p(\omega)}$$

Pour visualiser un peu a quoi peut correspondre cette probabilité :

Nous avons implémenté une fonction `eq_grille_alea(self)` qui génère une grille aléatoire et fait une boucle en en générant des grilles aussi aléatoires jusqu'à ce que l'une d'elle soit égale à celle généré en premier lieu, et à la fin elle renvoie le nombre de grilles générées.

Malheureusement on a testé plusieurs fois la fonction sans jamais obtenir de résultat, donc trouver une nouvelle méthode paraît donc nécessaire.

Approximation du nombre total de grilles pour une liste de bateaux

On peut approximer le nombre totale de grilles en multipliant les résultats respectifs de la fonction `nb_possiblite(self, bateau)` appliquée successivement aux cinq bateaux. Vu que cette fonction est appliquée sur grille vide pour chaque bateau et ne prend pas en compte le chevauchement des bateaux donc le résultat de l'approximation serait une borne supérieure de nombre de grille.

Voici un algorithme qu'on a proposé pour approximation le nombre total de grilles :

```
def approximation_nbGrille(grille_vide, list_bateau) :
    resultat = 1
    for bateau  in list_bateau :
        resultat = resulat * pos_un_bateau(grille_vide, bateau)
    return resutlat
```

On remarque que cet algo donne le même résultat que la borne supérieure qu'on avait réalisé théoriquement sur les questions précédentes. Donc on aperçoit que ce n'est peut-être pas la bonne manière d'approximer mieux le nombre total de grilles, nous devons trouver une meilleure approximation à notre fonction.

Meilleure approximation

Une meilleure manière d'approximer le nombre de grille pour un liste de bateau serait de procéder de la même manière que l'approximation précédente mais à la différence qu'au lieu de dénombrer le nombre de façons de placer chaque bateau dans une grille vide indépendamment, on place plutôt au fur et à mesure chaque bateau de la liste aléatoirement sur la grille, de cette manière le nombre de placement possible de chaque bateau dépend des cases disponibles sur la grille.

Voici ce que ça donne si on traduit en algorithme :

```
def meilleur_approximation_nbGrille(grille, list_bateau) :
    resultat = 1
    for bateau  in list_bateau :
        resultat = resulat * pos_un_bateau(grille, bateau)
        place_alea(grille, bateau)
    return resutlat
```

2- Modélisation probabiliste du Jeu

Nous allons à présent tenter de modéliser différentes stratégies de jeu que nous comparerons à une stratégie qui tire avantage de la connaissance des probabilités.

2.1 Version aléatoire

Comme base de comparaison, la stratégie qui semble la moins efficace pour gagner une partie est de jouer aléatoirement chaque coup. Cette section rend compte de son étude.

Espérance théorique du nombre de coups joués avant de couler tous les bateaux si on tire aléatoirement chaque coup

Soit notre grille contenant $N = 100$ cases, dont m sont des bateaux et $N - m$ cases vides. On tire alors un échantillon de k cases. On calcule d'abord le nombre de combinaisons correspondant à n cases bateaux en multipliant le nombre de possibilités de tirage de n cases bateaux parmi m par le nombre de possibilités de tirage du reste, soit $k - n$ cases vides parmi $100 - m$. On divise ensuite ce nombre de possibilités par le nombre total de tirages.

La probabilité d'obtenir n cases bateaux est par conséquent donnée par une loi hypergéométrique. Si on appelle X le nombre de cases bateaux tirées, la probabilité d'en avoir n s'écrit $p(X = n)$ et vaut :

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

De cette formule on déduit la probabilité d'avoir au moins 17 cases bateaux parmi k tirées qui vaut donc :

$$p(X \leq k) = \frac{\binom{17}{17}\binom{83}{k-17}}{\binom{100}{k}} = \frac{\binom{83}{k-17}}{\binom{100}{k}}$$

Ainsi, la probabilité d'avoir besoin d'exactement k coups pour obtenir les 17 cases bateaux est donnée par la simple soustraction

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

Sachant que $p\,(X \leq 16) = 0$ (impossible d'obtenir les 17 cases bateaux en jouant moins de 17 coups) et que $p\,(X > 100) = 0$ (impossible de jouer plus de 100 coups).

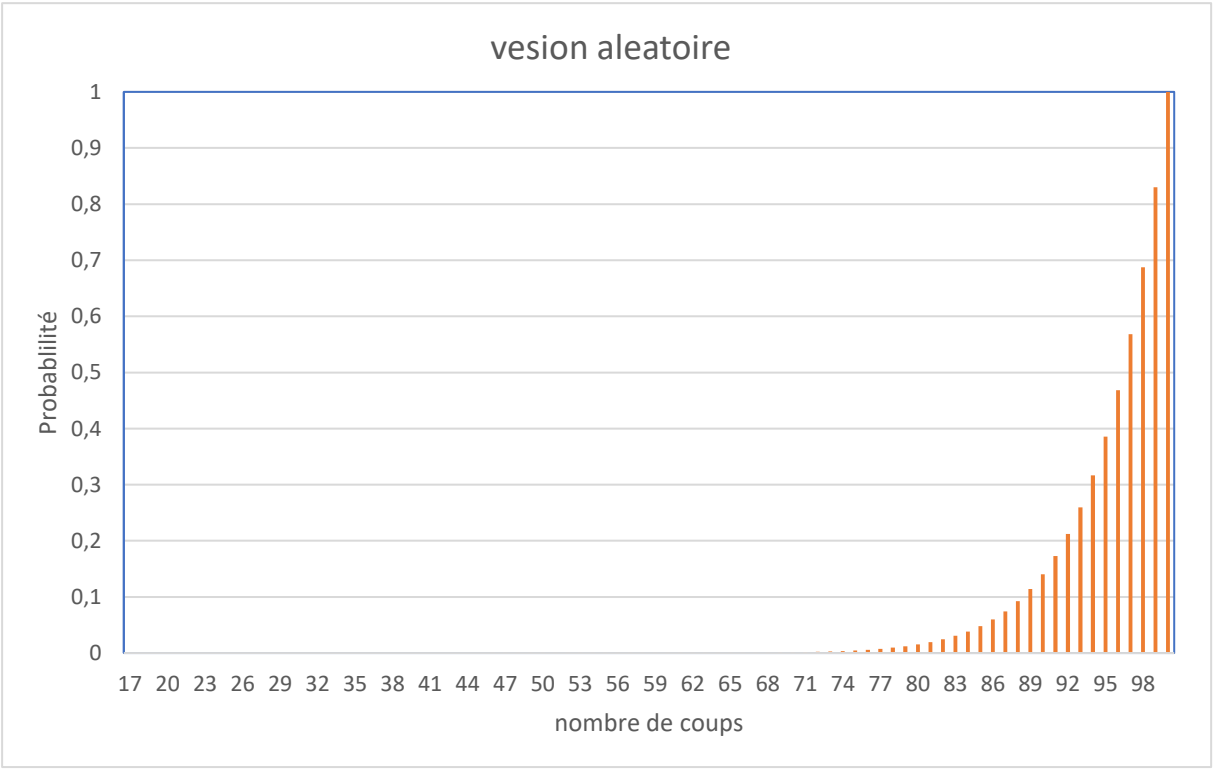


Fig1 : Graph de la distribution de la variable aléatoire

On calcule ainsi une espérance qui est égale à E(X) ≃ 95,39. On peut espérer gagner une partie de bataille navale avec 95 coups en moyenne si on joue au hasard.

Pour implémenter cette version du jeu, nous avons conçu une classe Joueur qui hérite de la classe Bataille (*l'implémentation des classes et méthodes sont expliquée dans la partie Annex*) et qui contient une méthode `jouer_alea()`, cette méthode choisie à chaque fois une position (x, y) aléatoire et tire sur la grille et s'arrête une fois tous les bateaux ont été coulés

Pour calculer l'espérance de la variable aléatoire correspondant au nombre de coups pour terminer une partie, nous lançons 1000 parties de bataille navale avec `jouer_alea()`, puis nous divisons la somme des nombres de coups joués par 1000. Nous obtenons un résultat égal à $\simeq 96{,}69$, très proche du résultat de notre calcul théorique ($\simeq 95{,}39$). En ce qui concerne la distribution, nous lançons 1000 parties et enregistrons dans un tableau le nombre de victoires en fonction du nombre de coups. Les résultats expérimentaux semblent valider notre modèle théorique (cf. figure 2 ci-dessous).

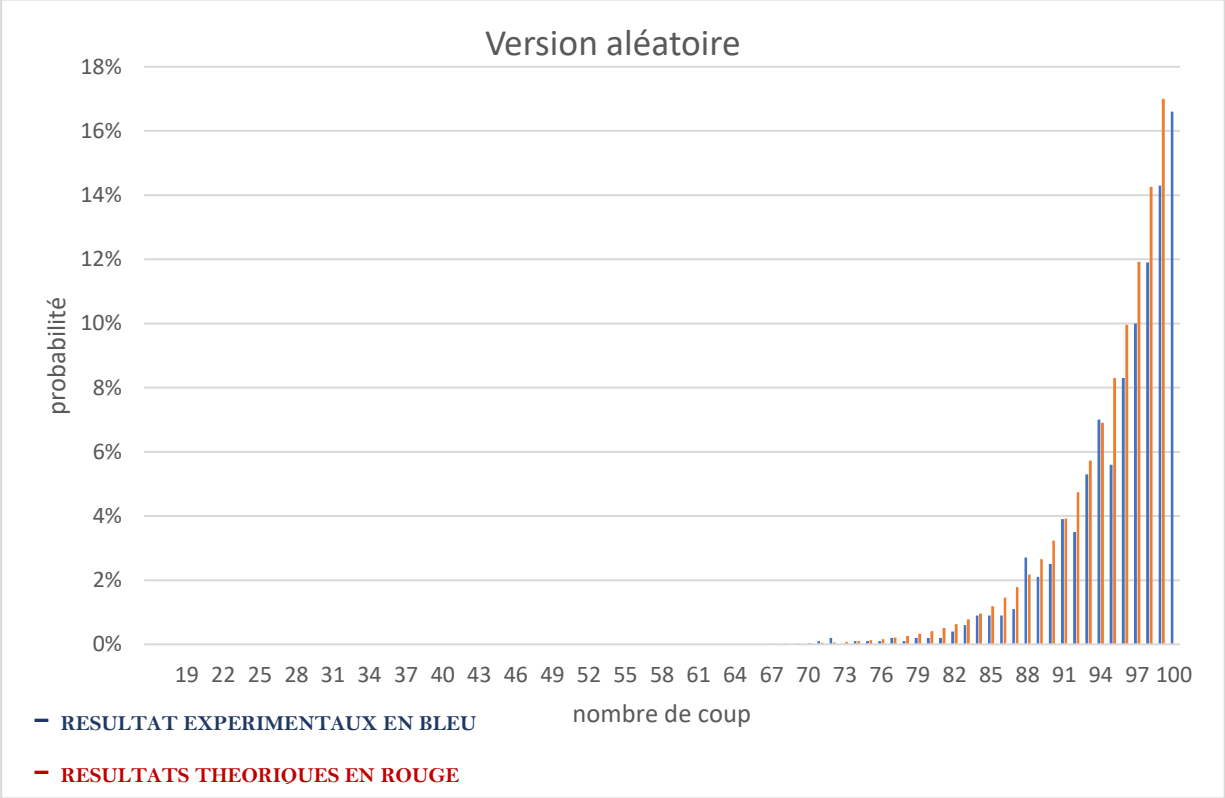


Fig2 : Graph de la distribution de v.a. correspondant au nombre de coups pour terminer une partie.

2.2 Version heuristique

La version aléatoire n'exploite pas les coups victorieux précédents, car elle continue à explorer aléatoirement tout l'échiquier alors que les coups victorieux apportent de l'information sur l'emplacement des cases restantes des bateaux touchés.

Pour jouer plus intelligemment, nous pensons à une stratégie qui consisterait à jouer aléatoirement jusqu'à toucher une case bateau, après on explore les cases adjacentes.

Pour réaliser cette stratégie nous avons implémenté une méthode `jouer_heuristique(self)` dans la classe Joueur (*l'implémentation de la méthode est expliqué dans Annex*).

Pour calculer la distribution de la variable aléatoire, nous avons réalisé la même expérience que précédemment en lançant 1000 parties avec la fonction jouer_heuristique et nous avons le graph ci-dessous.

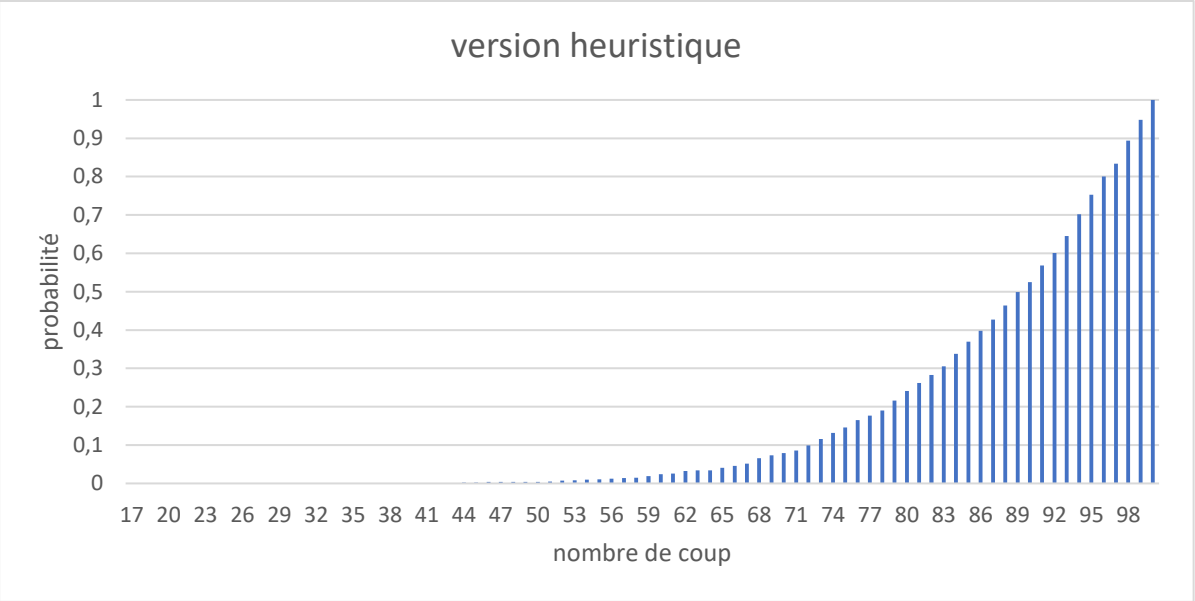
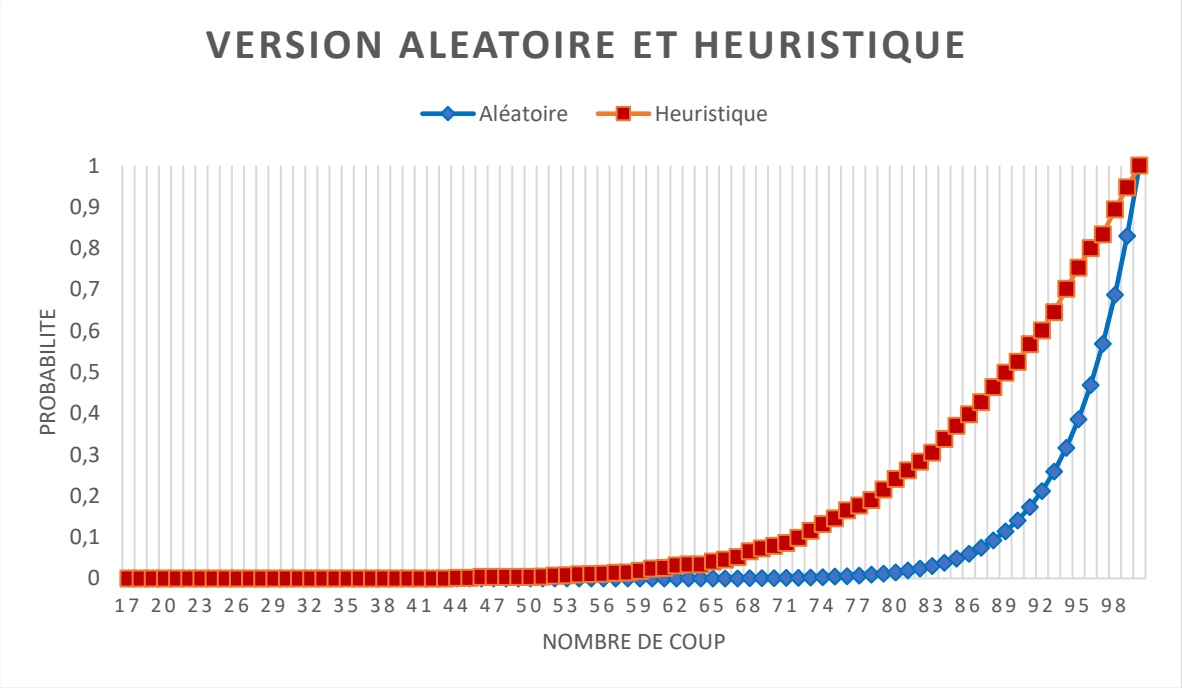


Fig3 : Graph de la distribution de la variable aléatoire

L'espérance obtenue est égale à E(X) ≃ 82,86. On gagne une partie de bataille navale avec 83 coups en moyenne avec la méthode heuristique.

Nous allons à présent comparer la distribution correspondant au nombre de coups pour terminer une partie avec chacune des méthodes



On observe que la stratégie heuristique est bien plus efficace que l'aléatoire, avec une espérance de ≃ 82,86 contre ≃ 96,69.

2.3 Version probabiliste simplifiée

Dans cette stratégie, nous allons faire appel à notre connaissance des probabilités pour optimiser le nombre de coup. La version précédente ne prend pas en compte le type de bateaux restant de l'admissibilité de leur positionnement : or, certaines positions ne seront pas possibles en fonction de la localisation de la case touchée (par exemple à côté des bords de la grille ou si un autre bateau a déjà été touché dans la région connexe). Chaque coup nous renseigne sur une position impossible ou possible d'un bateau.

Pour mettre cette stratégie en place, nous implémenté deux fonctions `jouer_probabiliste(self)` et `case_probable(self)` dans la classe Joueur (*détail des méthodes dans Annex*).

Cette version se repose sur le fonctionnement suivant :

Nous considérerons chaque bateau de manière indépendant, à chaque tour, pour chaque bateau restant, on calcule la probabilité pour chaque case de contenir ce bateau sans tenir compte de la position des autres bateaux. Pour cela, en examinant toutes les positions possibles du bateau sur la grille, on obtient pour chaque case le nombre de fois où le bateau apparaît potentiellement. On dérive ainsi la probabilité jointe de la présence d'un bateau sur une case (en considérant que la position des bateaux est indépendante.

Nous obtenons le graph de distribution ci-dessous après avoir lancé 1000 parties de jeu.

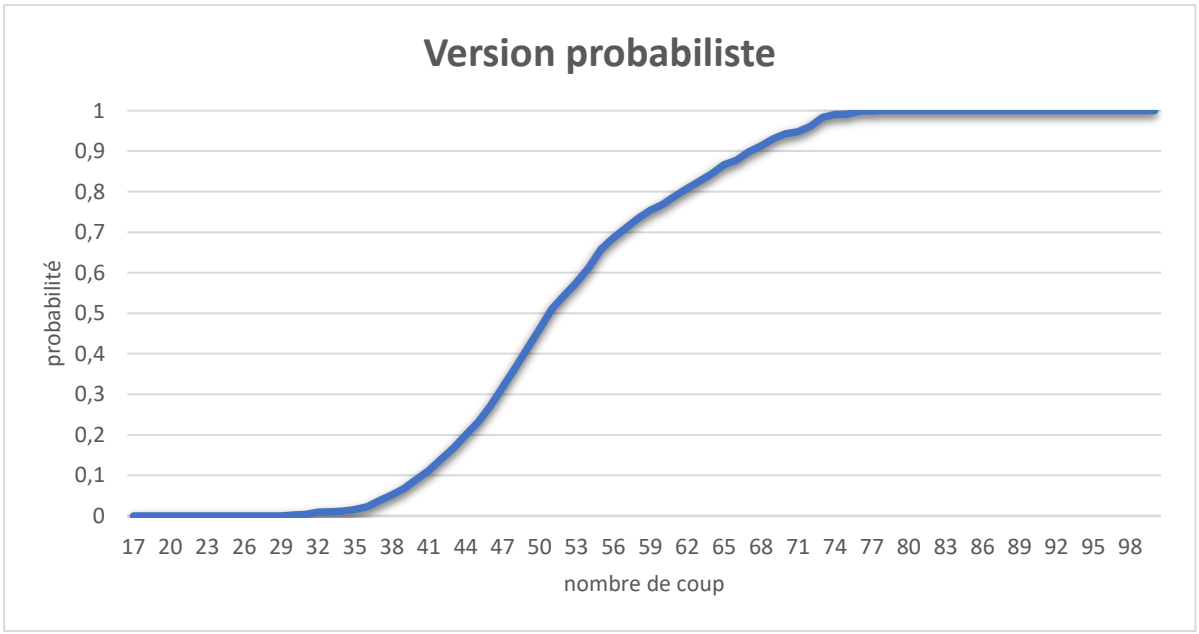


Fig5 : Graph de la distribution de v.a. correspondant au nombre de coups pour terminer une partie.

L'espérance obtenue est égale à E(X) ≃ 53,88. On gagne une partie de bataille navale avec 53 coups en moyenne avec la méthode probabiliste.

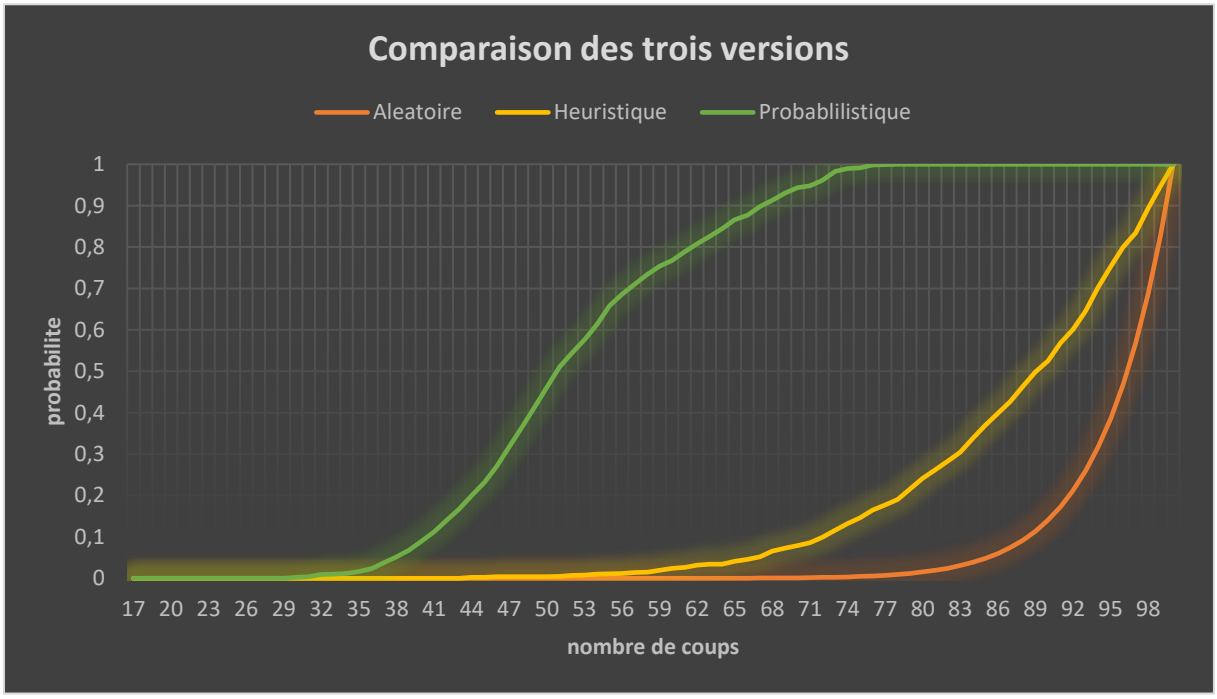


Fig6 : Comparaison des graphes de distribution des trois stratégies.

La stratégie probabiliste simplifiée est beaucoup plus efficace que les deux autres avec une espérance ≃ 52.88 contre ≃ 82.86 pour la version heuristique et ≃ 96.69 pour la version aléatoire.

3- Senseur imparfait : à la recherche de l'USS Scorpion

- P_s est la probabilité que le senseur trouve l'objet lorsque la zone sondée contient l'objet recherchée. Sachant que par définition :

-$Y_i \in \{0, 1\}$ est la variable aléatoire qui vaut 1 pour la case où le sous-marin se trouve et 0 partout ailleurs.

- Z_i est le résultat d'une recherche en case i, valant 1 en cas de détection et 0 sinon.

On suppose que les réponses aux sondages des cases sont indépendantes et identiquement distribuées ; on suppose que les valeurs de πi et de Ps sont données et fixées par les experts.

Nous pouvons formaliser Ps en fonction de Zi et Yi de cette manière

$$p_s = P(Z_i = 1 \mid Y_i = 1)$$

Y_i suit une loi de Bernoulli de paramètre π_i, soit :

$$Yi \sim B(\pi_i)$$

Concernant $Z_i | Y_i$, il apparait qu'ils suivent une loi de Bernoulli de paramètre Ps, soit :

$$Z_i | Y_i \sim B(p_i)$$

On s'intéresse au cas où le sous-marin se trouve en case k et un sondage est effectué à cette case mais ne détecte pas le sous-marin.

Soit $Z_k = 0$ et $Y_k = 0$, on cherche donc $P(Z_k = 0, Y_k = 1)$
D'après la formule de Bayes on a :

$$P(Z_k = 0, Y_k = 1) = P(Z_k = 0 \mid Y_k = 1) P(Y_k = 1)$$

$$P(Z_k = 0, Y_k = 1) = (1 - p_s)\pi_k$$

Il est admis dans l'énoncé que le sous-marin se trouve bien en case k, soit $\pi k = 1$

On a donc :

$$P(Z_k = 0, Y_k = 1) = (1 - ps)$$

Nous souhaitons mettre à jour π_k après avoir effectué un tel sondage infructueux. Pour i=k le nouveau π_k après un sondage infructueux en case k s'exprime de la façon suivante d'après la question précédente :

$$\pi_k = \frac{P(Z_k = 0, Y_k = 1)}{(1 - p_s)}$$

Si l'objet n'est pas détecté dans la cellule k, alors on pourrait croire qu'il se trouve dans une autre cellule, la probabilité que l'objet se trouve sur la case i avec $i \neq k$ devrait donc augmenter, étant donné que l'espace de recherche est divisé en un quadrillage de N cellules. On obtient que :

$$\pi_i = \pi_i + \frac{\pi_i - \pi_k}{1\text{-}N}$$

Annexe

Nous allons expliquer dans cette partie comment nous avons implémenté nos différentes versions.

Version aléatoire :

On génère un tableau contenant toutes les positions de la grille, ainsi qu'un compteur initialisé à 0.

Tant que tous les bateaux n'ont pas été abattus (donc que self.victoire() renvoie False), on réalise une boucle while, qui, à chaque itération, prend une position (i,j) au hasard à l'aide de la fonction random.choice(), joue sur cette case-là puis incrémente le compteur de coups de 1 et enlève la position qui vient d'être jouée de la liste de positions initialisée au début de la fonction.

Version heuristique :

Tout comme pour la version aléatoire, on génère un tableau contenant toutes les positions de la grille, ainsi qu'un compteur initialisé à 0. En plus de ces deux attributs, on rajoute un booléen touche qui est initialisé à False.

Tant que tous les bateaux n'ont pas été abattus, on réalise une boucle while, qui, à chaque itération prend une position (i,j) au hasard, joue sur cette case et efface la position choisie de la liste « positions » initialisée au début. On associe cette fois ci la valeur de joue(i,j) (si on est tomé sur un bateau, alors True, sinon False) à notre argument touche.

Si touche est vrai, alors on va jouer sur les cases connexes à (i,j) à l'aide d'une boucle for qui va parcourir les positions (i+1,j), (i,j+1), (i-1,j), (i,j-1). Pour chaque itération de la boucle on vérifie si la case sondée n'est pas en dehors de la matrice Grille. Si la case est bien dans le jeu, on joue sur cette position, toujours en enlevant cette dernière de la liste « positions », et on incrémente ensuite cpt de 1.

En Sortant du if, on incrémente cpt de 1 (pour la première recherche qui était aléatoire) on conclut en retournant le compteur.

Version probabiliste simplifiée :

Pour cette version nous avons implémenté deux fonctions :

jouer_probabiliste(self) et case_probable(self)

jouer_probabiliste() : on initialise un compteur à 0, tant que tous les bateaus n'ont pas été coulé, on réalise une boucle, qui, à chaque itération, joue sur (x,y) tel que (x,y) est la case la plus probable de contenir un bateau (fonction case_probable). A chaque itération est également incrémenté de 1 le compteur. Une fois la boucle terminée, on retourne le compteur.

case_probable() : on génère une grille 10*10 vide qui permettra d'y mettre la probabilité

de chaque case d'avoir un bateau, deux attributs debut et boost tous les deux initialisés à 0, un booléen peut_poser initialisé à False, ainsi que list_bateau la liste des bateaux.

On réalise une boucle for qui va parcourir chaque bateau de list_bateau. Si le bateau est déjà coulé, alors on passe au suivant, sinon :

On implémente une boucle imbriquée qui parcourt chaque case de la grille de jeu :

- Si la case a déjà été jouée, sa probabilité reste à 0 comme lors de l'initialisation de la grille, et on passe à la case suivante.
- Si la case est un bateau touché, on boost la probabilité qu'il y ai un bateau à côté en donnant aux attributs boost et debut la valeur 1.

  Pour attribuer aux cases verticales une probabilité d'avoir un bateau, on parcourt en vertical les cases allant de celle sur laquelle on se trouve jusqu'à ce que le nombre de cases parcouru soit égal à la taille du bateau étudié. Si on déborde de la grille ou que le cas est déjà joué, on remet peut_poser à False. En revanche, si on rencontre une case dont un bateau est touché, on incrémente boost de 1 (on boost la probabilité).

  De plus, si on peut poser le bateau étudié, on associe à la case (i,j) étudiée la valeur 1+boost*2.

  On fait de même ensuite pour l'horizontale.

On transforme maintenant notre grille en liste de probabilités. On récupère ensuite la probabilité max appelée maxiProba. On récupère l'indice de ligne et de colonne de maxiProba. Enfin, on retourne cet indice (x,y) qui correspond à la case avec la plus forte probabilité.