# Projet 1 : Bataille navale

Bruce ROSE et Nasser CHAKER

22 mars 2020

---

## Table des matières



---

3I005 – Probas, Stats et Info
Année 2019 - 2020

SCIENCES SORBONNE UNIVERSITÉ

# Introduction

L'étude probabilistique d'un objet donné représente un grand intérêt dans la prise de décision. Connaître les probabilités d'apparition d'un évènement équivaut à avoir une certaine connaissance des futurs possibles, ce qui nous permet d'agir en prévision de ceux-ci. Le jeu de la bataille navale se prête bien à une étude probabiliste. Nous disposons d'une grille de $10 \times 10$ cases sur laquelle sont positionnés 5 bateaux de tailles respectives 5, 4, 3, 3 et 2. Chaque tour, il nous faut faire feu sur une case de la grille, sans que l'on connaisse la position des bateaux. Comment la probabilistique peut nous aider à gagner au jeu de la bataille navale en un nombre minimum de coups ?

# 1 Combinatoire du jeu

Afin de nous rendre compte de la combinatoire du jeu, nous allons nous intéresser au dénombrement du nombre de grilles possibles dans différentes conditions.

### Borne supérieure du nombre de configurations possibles pour la liste complète de bateaux sur une grille de taille 10

Nous disposons d'un échiquier de 10 par 10 cases, ainsi que de 5 bateaux qui ensemble comptabilisent $5 + 4 + 3 + 3 + 2 = 17$ cases. Une manière simple d'obtenir une borne supérieure du nombre de configurations possibles pour la liste complète de bateaux consiste à placer les 17 cases parmi 100, qui se traduit par un arrangement $A_{100}^{17}$. Ce résultat est supérieur au nombre de configurations réel puisqu'elle ignore que les cases constituant un bateau doivent être adjacentes.

Pour obtenir un nombre plus exact de configurations possibles, nous allons les énumérer expérimentalement. Pour commencer, nous énumèrerons le nombre de façon de placer un seul bateau sur une grille vide.

### Dénombrement du nombre de façons de placer un bateau donné sur une grille vide

En théorie, sur une ligne, un bateau de taille $n$ donné dispose de $10 - n + 1$ configurations possibles. Etant donné que la grille contient 10 lignes et qu'un bateau peut être soit horizontal soit vertical, il existe donc $2 \times 10 \times (10 - n + 1)$ façons de positionner un bateau de taille $n$ sur la grille entière.

Notre fonction `pos_un_bateau` prend en entrée une grille et le numéro d'un bateau et renvoie le nombre de possibilités de placer ce bateau sur cette grille. Elle passe en revue chaque case de la grille et appelle la fonction `peut_placer` qui renvoie `vrai` si le bateau peut être placé avec pour origine cette case. Cette dernière prend en compte la longueur du bateau, les bordures ainsi que les bateaux éventuellement déjà présents sur la grille. La table 1 présente nos résultats.

| Résultats \ $n=$ | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| théoriques | 180 | 160 | 140 | 120 |
| obtenus | 180 | 160 | 140 | 120 |

TABLE 1 : Comparaison du nombre de façons de placer un bateau de longueur $n$ sur une grille vide, calculé de manière théorique et expérimentale.

On observe la validation expérimentale de nos calculs théoriques. Toutefois, ceux-ci ne concernent que le positionnement d'un bateau. Il s'agit à présent de faire de même mais pour la liste complète des cinq bateaux.

**Dénombrement du nombre de façons de placer une liste de bateaux sur une grille vide**

Pour calculer le nombre de façons de placer une liste de bateaux sur une grille vide, nous avons conçu la fonction récursive `pos_des_bateaux` qui prend en paramètre une liste de numéros de bateaux. La fonction procède ainsi : elle place le premier bateau à la première position viable sur une grille vide au départ. Puis, le second est posé à sa première position viable, etc, jusqu'à ce que tous les bateaux aient été placés. À ce moment là, le dernier bateau est placé à sa seconde position viable, puis à sa troisième jusqu'à ce qu'il ait épuisé toutes ses positions. L'avant dernier bateau est alors posé à sa deuxième position viable et on recommence à poser le dernier bateau à toutes ses positions viables comme précédemment. On réitère l'opération jusqu'à ce que tous les bateaux aient épuisés leurs positions viables. On renvoie finalement le nombre de grilles possibles contenant tous les bateaux rencontrées pendant l'énumération.

| Fonction \ Nombre de bateaux | 1<br>$n = 2$ | 2<br>$n_1 = 2, n_2 = 3$ | 3<br>$n_1 = 2, n_2 = n_3 = 3$<br>environ 3m |
|---|---|---|---|
| `pos_des_bateaux`<br>(nombre de grilles) | 180 | 27 336 | 3 848 040 |

TABLE 2 : Résultats renvoyés par la fonction `pos_des_bateaux` avec 1, 2 et 3 bateaux, de longueurs $n_i$ .

Le calcul du nombre de grilles avec 4 bateaux étant trop grourmand en ressources pour pouvoir l'ajouter au tableau, 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**

La probabilité uniforme sur un univers fini Ω est définie par la fonction de masse :

$$p(\omega) = \frac{1}{card(\Omega)}$$

En posant $\omega$ l'événement élémentaire « tirer une grille donnée », Ω toutes les grilles possibles et $g$ le nombre de grilles on a donc :

$$p(\omega) = \frac{1}{card(\Omega)} = \frac{1}{g}$$

Nous pouvons exploiter cette formule pour déduire d'une probabilité le nombre de grilles :

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

Or la probabilité de tirer une grille donnée peut être approchée grâce à une fonction simple.

Soit la fonction `grilleAleaEgale`. Elle génère des grilles aléatoires jusqu'à ce que l'une d'elle soit égale à celle passée en paramètre. Finalement, elle renvoie le nombre de grilles générées. Pour tester cette fonction (et pour obtenir des résultats plus précis), nous avons implémenté une fonction `test_grilleAleaEgale` qui pour une durée donnée, itère autant que possible sur la fonction `grilleAleaEgale` pour renvoyer au final une moyenne des résultats obtenus. Cette fonction de test nous permet de tester la fonction `grilleAleaEgale` en un temps raisonnable.

| Nombre de bateaux ($n$ : taille du bateau) | `test_grilleAleaEgale` | | `pos_des_bateaux` (nombre de grilles) (2) | Différence entre (1) et (2) |
|---|---|---|---|---|
| | Nb d'itérations possibles en **15s** | Moyenne des nombres de grilles retournés (1) | | |
| 1 ($n = 2$) | 2 997 | 182 | 180 | 2 |
| 2 ( $n_1 = 2$ $n_2$=3) | 16 | 29 566 | 27 336 | 2 230 |
| 3 ($n_3 = 3$) | 2 (103s) | 1 744 521 | 3 848 040 | 2 103 519 |
| 4 ($n_4 = 4$) | / | / | / | / |
| 5 ($n_5 = 5$) | / | / | / | / |

TABLE 3 : Comparaison des résultats de `test_grilleAleaEgale` et `pos_des_bateaux`.

Malheureusement, la fonction `test_grilleAleaEgale` appliquée à plus de trois bateaux termine en un temps indéterminé supérieur à 45 minutes. Par conséquent, nous n'avons pas pu ajouter les résultats correspondants dans le tableau.

D'autre part, on observe que la différence entre `test_grilleAleaEgale` et `pos_des_bateaux` est inversement proportionnelle au nombre d'itérations possibles dans la limite de temps donnée. En revanche, pour un grand nombre d'itérations (pour l'application à un seul bateau par exemple), les résultats de `test_grilleAleaEgale` semblent plutôt corrects (si on considère les valeurs renvoyées par `pos_des_bateaux` valides).

Une fois de plus des contraintes techniques nous empêchent ne serait-ce que de pouvoir estimer le nombre de grilles possibles. Trouver une nouvelle méthode paraît donc nécessaire.

### Approximation du nombre total de grilles pour une liste de bateaux

Une manière un peu grossière consiste à multiplier les résultats respectifs de la fonction `pos_un_bateau` appliquée successivement aux cinq bateaux. On obtient en quelque sorte le cardinal du produit cartésien $\prod_{i=1}^{m} \Omega_i$ où $\Omega_i$ représente l'ensemble des grilles contenant uniquement le bateau $i$. Le résultat émanant d'une telle fonction constituerait une borne supérieure, puisque deux bateaux pourraient se chevaucher, ce qui est impossible en pratique et réduit donc le nombre de possibilités.

| Nombre de bateaux ($n$ : taille du bateau) | `pos_des_bateaux` (nombre de grilles) | `approx_nbGrille1` | Différence |
|---|---|---|---|
| 1 ($n = 2$) | 180 | 180 | 0 |
| 2 ( $n_1 = 2$ $n_2$=3) | 27 336 | 28 800 | 1 464 |
| 3 ($n_3 = 3$) | 3 848 040 | 4 608 000 | 759 960 |
| 4 ($n_4 = 4$) | / | 645 120 000 | ? |
| 5 ($n_5 = 5$) | / | 77 414 400 000 | ? |

TABLE 4 : Comparaison de `pos_des_bateaux` et `approx_nbGrille1`.

La différence des résultats croît en même temps que le nombre de bateaux. Nous devrions pouvoir affiner notre fonction d'approximation.

### Une meilleure approximation

L'approximation précédente ne prend pas en compte l'impossibilité pour deux bateaux de se chevaucher. Pour simuler cette contrainte, nous avons conçu un nouvel algorithme. Celui-ci est le même que précédemment, à la différence qu'au lieu de dénombrer le nombre de façons de placer chaque bateau dans une grille vide, on les place au fur et à mesure aléatoirement dans la grille, créant ainsi les contraintes de positions recherchées.

Soit la fonction `approx_nbGrille2` qui prend en paramètre une liste (de numéros) de bateaux et qui utilise la fonction `place_alea` qui prend en paramètre une grille et un bateau et place ce dernier à une position aléatoire dans la grille. Le pseudo-code vous est présenté ci-dessous.

```
approx_nbGrille2
    Entrée :
              listNum, liste de numéros de bateaux.
    Sortie :
              Une approximation du nombre de grilles différentes possibles
              contenant la liste de bateaux passée en paramètre.
  res ⇐ 1
  grilleCour est une grille vide
  Pour chaque bateau b de listNum
      res ⇐ res × pos_un_bateau(grilleCour, b)
      place_alea(grilleCour, b)

  retourner res
```

| Nombre de bateaux ($n$ : taille du bateau) | pos_des_bateaux (nombre de grilles) (1) | approx_nbGrille1 (2) | approx_nbGrille2 (3) | Différence entre (1) et (2) | Différence entre (1) et (3) |
|---|---|---|---|---|---|
| 1 ($n = 2$) | 180 | 180 | 180 | 0 | 0 |
| 2 ( $n_1 = 2$ $n_2$=3) | 27 336 | 28 800 | 27 360 | 1 464 | 24 |
| 3 ($n_3 = 3$) | 3 848 040 | 4 608 000 | 3 753 000 | 759 960 | 95 040 |
| 4 ($n_4 = 4$) | / | 645 120 000 | 389 232 000 | ? | ? |
| 5 ($n_5 = 5$) | / | 77 414 400 000 | 29 456 161 920 | ? | ? |

TABLE 5: Comparaison des résultats de pos_des_bateaux, approx_nbGrille1 et approx_nbGrille2.

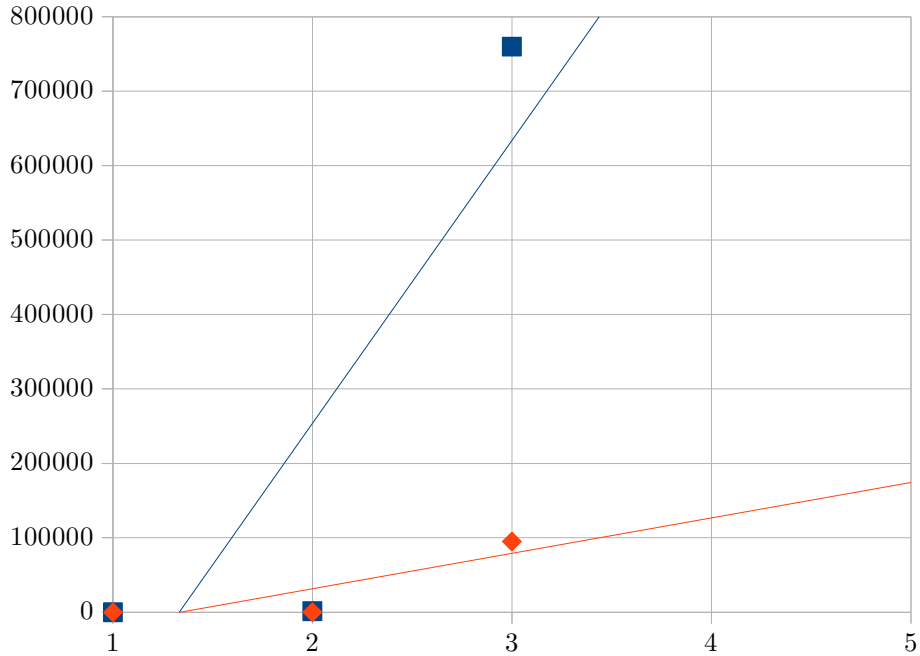


FIGURE 1: Comparaison des écarts de résultats avec `pos_des_bateaux` en nombre de grilles en fonction du nombre de bateaux, avec leurs lignes de tendance.
En bleu, `approx_nbGrille1`.
En rouge, `approx_nbGrille2`.

La figure 1 nous montre dans quelle mesure la fonction `approx_nbGrille2` rend des résultats beaucoup plus affinés que `approx_nbGrille1`. Pour obtenir des résultats encore plus précis, nous pourrions faire la moyenne des résultats de 1000 exécutions de `approx_nbGrille2`.

Ainsi d'après `approx_nbGrille2` il existerait $29\,456\,161\,920 \pm 200\,000$ grilles possibles.

## 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. Les autres sont la mer et sont au nombre de $N - m$. 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 mer 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 $Y$ le nombre de cases bateaux tirées, la probabilité d'en avoir $n$ s'écrit $\mathbb{P}(Y = n)$ et vaut :

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

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).

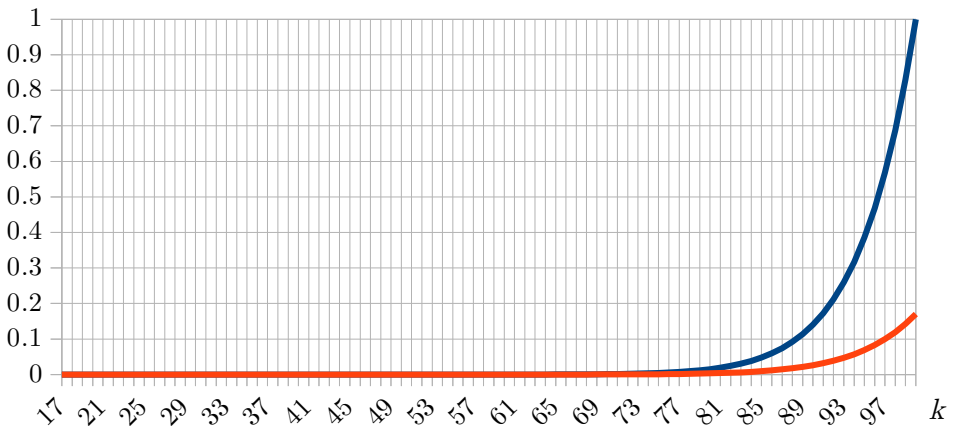


FIGURE 2 : En bleu, probabilité de gagner en au plus $k$ coups; en rouge, probabilité de gagner en exactement $k$ coups.

On calcule ainsi une espérance égale à $\simeq 95{,}39$ . On peut espérer gagner une partie de bataille navale en 95 tours en moyenne si on joue au hasard.

### Implémentation du joueur aléatoire

Nous avons conçu une classe `JoueurAlea` qui contient une méthode `joue(self, bataille)`. `bataille` est un objet de classe `Bataille` qui contient un attribut `grille` et un attribut `vies`, ainsi qu'une méthode `joue(self, position)`, `victoire(self)`, `affiche(self)`, et `checkbound(self, position)`.

La méthode joue de `JoueurAlea` a pour but de décider d'une position `(x, y)` à jouer à un tour donné. Comme la stratégie du `JoueurAlea` est de tirer aléatoirement, cette méthode se contente de générer une position au hasard qu'elle retourne.

### Distribution de la variable aléatoire correspondant au nombre de coups pour terminer une partie

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 `JoueurAlea`, puis nous divisons la somme des nombres de coups joués par 1000. Nous obtenons un résultat égal à $\simeq 95{,}48$, 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 3 ci-dessous).

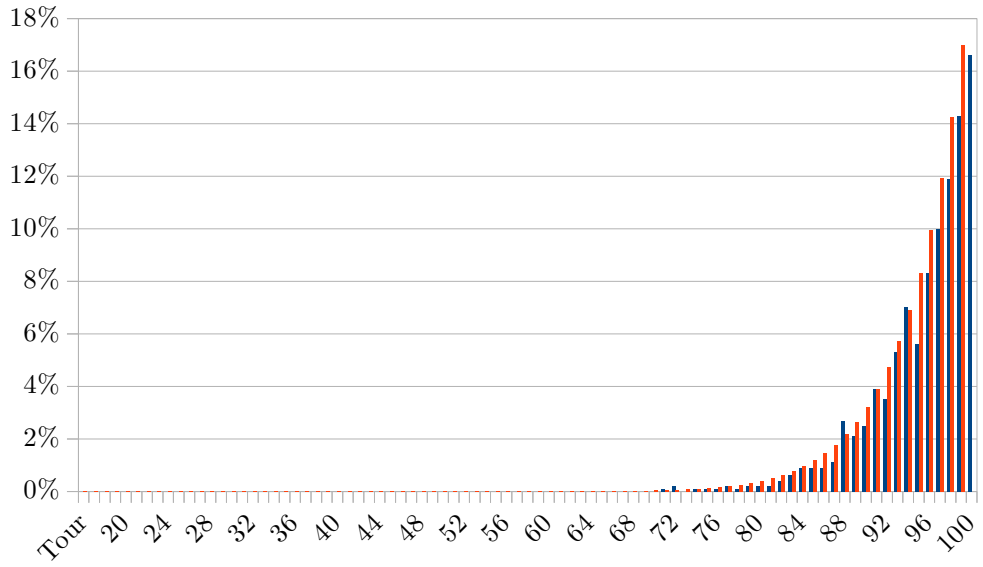


FIGURE 3 : Distribution de la variable aléatoire correspondant au nombre de coups pour terminer une partie. En bleu : résultats expérimentaux. En rouge : résultats théoriques.

De même, la distribution trouvée expérimentalement est très proche de celle calculée.

## 2.2 Version heuristique

Pour jouer plus intelligemment, nous pensons à une stratégie qui consisterait à jouer aléatoirement jusqu'à toucher une case bateau. Dans ce cas, on explore les cases adjacentes. Cette stratégie n'exploite pas la notion de probabilité. Nous verrons comment elle se situe par rapport à une qui au contraire ferait appel aux probabilités pour décider des coups à jouer.

### Implémentation du joueur heuristique

Nous avons conçu une classe `JoueurHeuristique`. Ce joueur tient à jour une liste de coups réussis et une liste de coups à suivre. Initialement, ces deux listes sont vides. Quand il est amené à jouer, si la liste *prochains_coups* est vide, le joueur tire un coup aléatoirement. Sinon, il joue la première case de *prochains_coups* et la retire de la liste. S'il touche un bateau, cette case touchée est ajoutée à *coups_reussis* et si l'une des cases adjacentes est dans *coups_reussis*, la case adjacente contraire est ajoutée à *prochains_coups*. Si la case touchée n'a pas de case adjacente dans *coups_reussis*, alors toutes les cases adjacentes sont ajoutées à *prochains_coups*.

**Exemple**

*Tour 1 :* initialement, `coups_reussis = []` et `prochains_coups = []`.

Comme `prochains_coups` est vide, le joueur tire au hasard `(5, 4)`. Touché ! `coups_reussis = [(5, 4)]`. Aucune case adjacente n'est dans `coups_reussis`, par conséquent elles sont toutes ajoutées à `prochains_coups`.

*Tour 2 :*

`coups_reussis = [(5, 4)]`,

`prochains_coups = [(5, 3), (4, 4), (5, 5), (6, 4)]`.

Le joueur tire `(5, 3)`. Touché ! Comme `(5, 4)` est dans `coups_reussis`, on s'aperçoit qu'il y a de grande chances pour qu'on ait touché un bateau horizontal; `(5, 2)` est ajouté à `prochains_coups`.

*etc.*

**Comparaison de la distribution correspondant au nombre de coups pour terminer une partie**

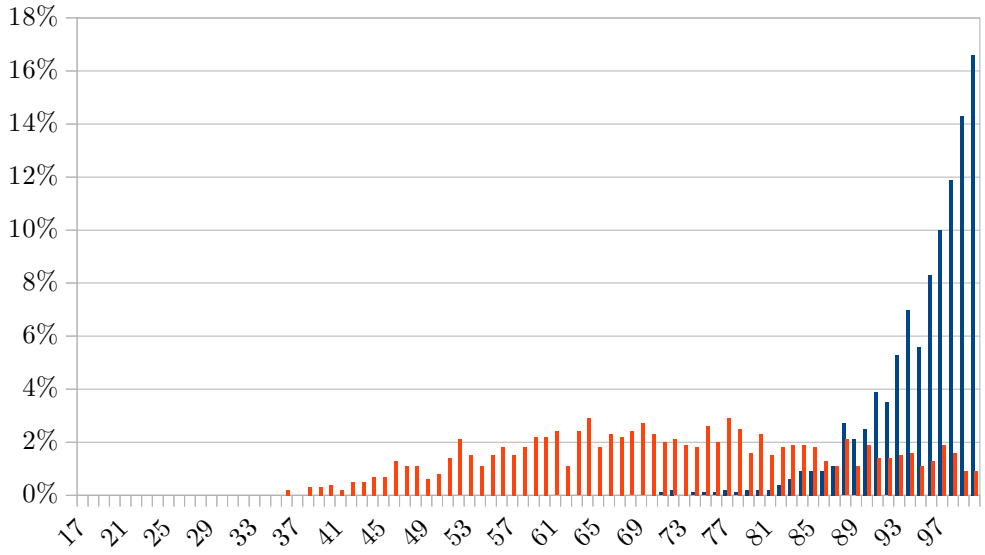


FIGURE 4 : Distribution du nombre de coups pour terminer une partie. En rouge, résultats expérimentaux obtenus avec la stratégie heuristique. En bleu, les résultats espérimentaux obtenus avec la stratégie aléatoire.

On observe que la stratégie heuristique est bien plus efficace que l'aléatoire, avec une espérance de $\simeq 71{,}63$ (contre $\simeq 95{,}48$ ).

## 2.3 Version probabiliste simplifiée

Dans cette stratégie, nous allons faire appel à notre connaissance des probabilités pour tirer le plus précisemment possible. La version précédente ne prend pas en compte le type de bateaux restant ni si le positionnement du bateau est admissible : or, certaines positions ne seront pas possibles en fonction de la localisation de la case touchée (par exemple à coté 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).

### Implémentation du joueur probabiliste simplifié

Notre classe `JoueurProba` est initialisée avec une liste des bateaux restants *restants* et une grille vide nommée *probas*. Pour chaque bateau restant (les bateaux coulés ne sont pas retenus), pour toute case `(i,j)` de la grille de jeu :

- Si la case `(i,j)` est une case vide touchée, *probas*`(i,j)` est mis à `0` (impossible qu'un bateau se trouve là) ;
- Sinon, si `(i,j)` est un bateau touché, une variable *bonus* est initialisée à `1` et *probas*`(i,j)` est mis à `0` (inutile de toucher une case déjà touchée). Par la suite, nous vérifions d'abord si on peut poser un bateau à `(i,j)` verticalement :
  - S'il n'est pas possible de placer le bateau verticalement, on passe directement à la vérification horizontale ;
    - Sinon, pour chaque case `(i',j')` que le bateau peut occuper, si celle-ci est un bateau touché alors bonus est augmenté de 1;
    - Ensuite, *probas*`(i',j')` est augmenté de $1 +$ *bonus* $\times 2$ (grâce à *bonus*, le programme est plus enclin à donner de grandes probabilités aux cases contigues et successives aux cases déjà victorieuses).
  - On procède de même horizontalement.

Finalement, on joue la case à plus forte « probabilité ». Au tour suivant, *probas* sera réinitialisé à une grille vide (`0` partout).

**Comparaison de la distribution correspondant au nombre de coups pour terminer une partie**

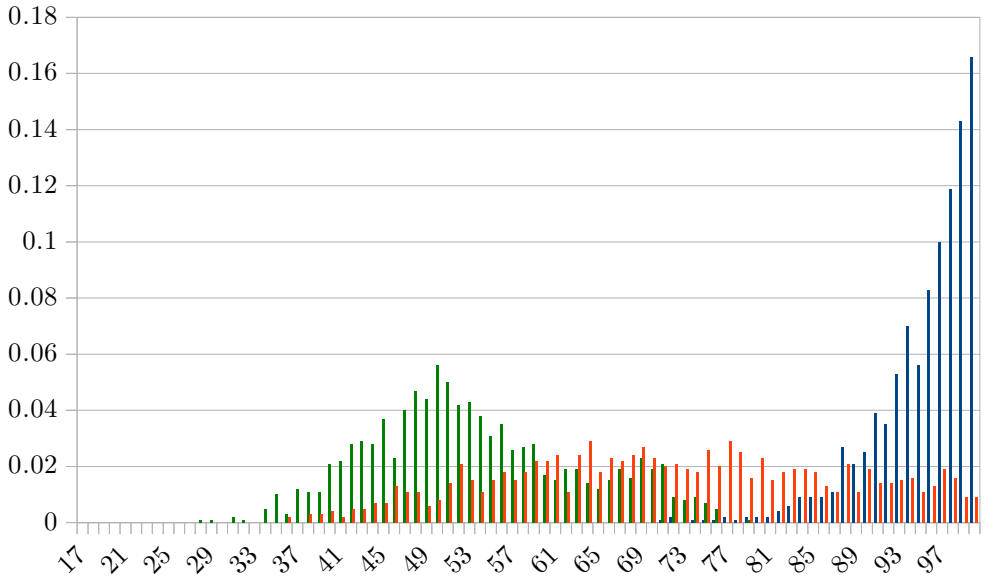


FIGURE 5: Distribution du nombre de coups pour terminer une partie.
En vert : résultats expérimentaux obtenus avec la stratégie probabiliste simplifiée.
En rouge : stratégie heuristique.
En bleu : stratégie aléatoire.

La stratégie probabiliste simplifiée gagne en moyenne en $\simeq 53{,}34$ tours, loin devant la stratégie heuristique ( $\simeq 71{,}63$ ) et aléatoire ( $\simeq 95{,}48$ ) : c'est la puissance des probabilités.

# Conclusion

Le jeu de la bataille navale possède une dimension combinatoire importante. Comme nous l'avons vu, jouer au hasard ne suffit pas pour gagner en un nombre de tours significativement réduit, justement car le nombre de grilles uniques possibles est si grand. Le joueur moyen, qui se contenterait de jouer au hasard jusqu'à toucher un bateau et dont il déduierait facilement alors la position, est finalement assez médiocre face à un adversaire qui aurait la patience de calculer minutieusement la probabilité de toucher un bateau à tel ou tel endroit.

Nous pourrions approfondir nos recherches et répondre à certaines questions pertinentes. Par exemple, pourquoi le joueur probabiliste simplifié semble-t-il toujours commencer ses parties de la même manière, en traçant des lignes diagonales au centre

de la grille ? Une stratégie heuristique tirant partie de cette technique pourrait-elle battre cette dernière ? Comment évoluent nos algorithmes avec des grilles plus grandes ?

D'autre part, nous pourrions faire l'étude d'une version de Monte-Carlo qui règlerait le problème d'indépendance des bateaux que notre version probabiliste simplifiée soulève.