Bataille navale

Étude statistique du jeu Bataille Navale

Boudrouss Réda n°28712638

Zhenyao Lin n°28708274

Sorbonne Université
France

20 octobre 2022

Contents

Introduction

L'objectif de ce projet est de mener une étude statistique sur le jeu “bataille navale”. Ce jeu consiste en une grille de dix par dix cases sur laquelle sont placés cinq bateaux de taille respective cinq, quatre, trois, trois et deux cases. À chaque coup, le jouer doit tirer sur une case, révélant ainsi son état : occupé par un bateau ou vide, ainsi le coup a raté, touché ou coulé.

Quelle est la meilleure statégie qui va nous permettre de gagner en minimisant le nombre de coup ?

Description du code

Le code de notre projet s'organise comme suit :

```
Bataille/
├── rapport/
└── src/
```

```
│
├── data/
├── Game/
│   ├── Bateau.py
│   ├── Engine.py
│   └── EngineStats.py
├── Player/
│   ├── AbstractPlayer.py
│   ├── HeuristicPlayer.py
│   ├── HumanPlayer.py
│   ├── MCPlayer.py
│   ├── ProbPlayer.py
│   └── RandomPlayer.py
├── utils/
│   └── constants.py
├── main.py
└── partie4.py
```

Le dossier `rapport/` est le dossier qui contient tous les fichiers qui constituent le présent rapport.

Le dossier `src/` contient l'essentiel du code et de nos différentes simulations.

Le fichier `main.py` permet d'exécuter le code. Pour lancer le programe il faut se déplacer dans `src`avec `cd src/` et ensuite lancer `main.py` avec `python main.py`.

Le dossier `data/` contient différentes données telles que les résultats des joueurs, des images ou des ressources nécessaires à la bonne excécution du programme.

Le dossier `utils/` contient des fonctions utiles au projet. Il contient aussi, et surtout, le fichier `constants.py` qui possède toutes les configurations du jeu et les conventions de programmation. N'hésitez pas à y changer certains paramètres.

Le dossier `Game` contient le code de la partie logique du jeu. C'est une sorte de "game engine" (d'où le nom `Engine.py`). `Engine.py` pour le jeu en lui même et `Bateau.py` pour le code de l'objet "Bateau" qui nous permet de simplifier et centraliser notre code pour le bateau. `EngineStats.py` a les mêmes caractéritiques que `Engine.py` mais avec quelques fonctions assez techniques en plus. C'est ici que vous trouverez certaines des fonctions requises dans le sujet.

Le fichier `partie4.py` contient le code nécessaire demandé pour la partie 4.

Nous avons fait le choix de connaître les bateaux en amont de leur placement, nous les avons alors indexés avec ce que nous appelons leur `type`. C'est le numéro du bateau dans une liste triée par taille. Par défaux nous avons :

Bateau	Taille	Type
Porte avion	5	1
Croiseur	4	2
Contre-torpilleurs	3	3
Sous-marin	3	4
Torpilleur	2	5

Combinatoire du jeu

Dans un premier temps, intéressons nous à la combinatoire du jeu “bataille navale”. Est-il possible de déterminer le nombre de potentielles grilles ?

Approche naïve

Un majorant naïf serait $A_{100}^{17} \approx 2 \times 10^{33}$. En effet nous avons un échiquier de taille 10 x 10, soit 100 cases, et nous devons choisir en tout $5 + 4 + 3 + 3 + 2 = 17$ cases. En ignorant toutes les règles, le nombre maximal de plateaux possibles est donc toutes les manières différentes de poser les 17 cases parmis les 100 disponibles, soit donc un arrangement de 17 parmi 100.

Cependant, ce nombre ne prend pas en compte le fait que les cases d’un même bateau doivent être adjacentes. Il est possible aussi de calculer un autre majorant un peu plus précis en comptabilisant manuellement le nombre de façon possible de poser un bateau de taille n dans un 10x10. Voici les résultats obtenus :

Bateau	Taille	nb
Porte avion	5	120
Croiseur	4	140
Contre-torpilleurs	3	160
Sous-marin	3	160
Torpilleur	2	180

En calculant le produit de ces nombres, on obtient le majorant du nombre de configuration maximal théorique : 7.74×10^{10}. Cependant ce majorant inclut les plateaux où les bateaux se supperposent.

Vérifions ces valeurs avec notre implémentation du jeu. Les fonctions pour ce faire se trouvent dans `src/Game/EngineStats.py`.

La fonction `EngineStats.nb_placer(type)` fait exactement ce que nous voulons, elle parcourt chaque case du plateau et vérifie avec `Engine.peut_placer(type)` si le bateau donné en paramètre peut être placé à la case, si oui elle ajoute 1. Nous obtenons bien les résultats théoriques avec cette fonction.

La fonction `EngineStats.nb_placerL(types)` utilise la fonction précédente pour calculer les plateaux possibles avec la méthode utilisée précédemment (donc elle inclut les plateaux où les bateaux se superposent). Et on retrouve le résultat théorique obtenu avant. La fonction est assez simple, nous faisons juste le produit de chacun des résultats.

```python
def nb_placerL(types: list[int])->int:
    nb = 1
    for type in types:
        nb *= EngineStats.nb_placer(type)
    return nb
```

Est-il possible d’obtenir une approximation encore plus précise du nombre de plateaux possibles ? Essayons la méthode brute.

Brute force

La fonction naïve et brute `EngineStats.nb_placerL_brute(types)` peut en théorie nous donner le nombre exact de plateaux possibles. Elle procèdes ainsi :

- Tous les bateaux sont posés à la première position possible en faisant bien attention de pas les superposer.
- Une fois tous les bateaux positionnés, elle incrémente le compteur de positions possibles et place le dernier bateau à toutes les positions possibles en incrémentant le compteur de positions possibles à chaque fois.
- Une fois que le dernier bateau a écoulé toutes ses positions, on place l'avant-dernier bateau à sa prochaine position viable.
- Et ainsi de suite jusqu'à ce que tous les bateaux aient pris toutes leurs positions possibles.

Voici les valeurs qu'on a pu obtenir :

Bateaux	`nb_placerL_brute()`
1	120
1, 2	14400
1, 2, 3	1850736

Cette fonction marche très bien pour des petites listes de bateaux ou pour des petites grilles, mais pour notre cas le temps d'exécution est énorme, même en optimisant avec les symétries cela prendrait toujours trop de temps, notre fonction se rapproche d'une complexité $O\left((3n)^m\right)$ avec n la taille de la dimension du plateau (supposé carré) et m le nombre de bateau.

Python n'est pas le langage pour de tels calculs, on sait que notre résultat doit être aux alentours de 10^{10}. Le bout de code simpliste suivant prend déjà trop longtemps à s'excécuter :

```python
i = 0
while i < 1e10:
    i+=1
```

N'y a-t-il pas une autre méthode qui ne nécessite pas de faire tant de calculs ?

Approche Aléatoire

Étudions dans un premier temps le lien entre le nombre de grilles et la probabilité d'en tirer une aléatoirement.

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

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

Avec ω l'événement élémentaire "tirer une grille donnée", Ω l'ensemble des grilles possibles et g le nombre de grilles.

Bien évidement on suppose ici que toutes les grilles sont équiprobables.

Nous pouvons donc explorer la formule suivante pour déduire d'une approximation du nombre totale de grille :

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

Notre fonction `EngineStats.nb_alea(grille)` génére de manière aléatoire des plateaux jusqu'à tomber sur celui donné en paramètre et retourne le nombre d'itérations effectuées. Malheuresement cette fonction prend trop de temps à s'exécuter pour les 5 bateaux selon notre implémentation.

Modélisation probabiliste du jeu

Nous allons maintenant modéliser différentes stratégies de jeu que nous analyserons.

Implémentation des Joueurs

Tous les joueurs doivent hériter de la classe `AbstractPlayer` et override les fonctions `play`, `reset`, et `name` pour être considérés comme Joueur. La classe `AbstractPlayer` s'occupe de plusieurs chose, telles que les interactions avec le game engine, détecter si le jeu est fini, gérer le plateau qui récapitule toute la vision qu'a le joueur à l'instant T et surtout la boucle principale. En clair elle regroupe tout le code centrale et nécessaire à un joueur.

Joueur Aléatoire

Le joueur aléatoire est sans doute la stratégie la plus naïve et la plus simple, mais jusqu'à quel point est-elle mauvaise ?

Étude probabilistique

Soit notre grille contenant $N = 100$ cases, $m = 17$ le nombre de cases occupées. La probabilité que le jeu se termine en n actions est alors :

$$P(n) = \frac{C_{n-m}^{N-m}}{C_1^n 00}$$

avec $17 \leq n \leq 100$. Ce qui nous donne comme espérence ≈ 95.4. C'est bien ce que l'on observe sur la *Figure 1*.

Implémentation

Notre stratégie aléatoire est implémentée dans le fichier `Player/RandomPlayer.py` avec une fonction `play` qui contient l'algorithme suivant :

- choisir aléatoirement une position dans l'ensemble des coups non-joués
- jouer cette position et la retirer de l'ensemble

Joueur Heuristique

Un peu de stratégie, nous jouons toujours aléatoirement dans la phase de détections mais cette fois nous usons des informations que nous donnes le jeu. Si une case est touchée, les cases des bateaux étant liées, il y a très probablement une case occupée adjacente.

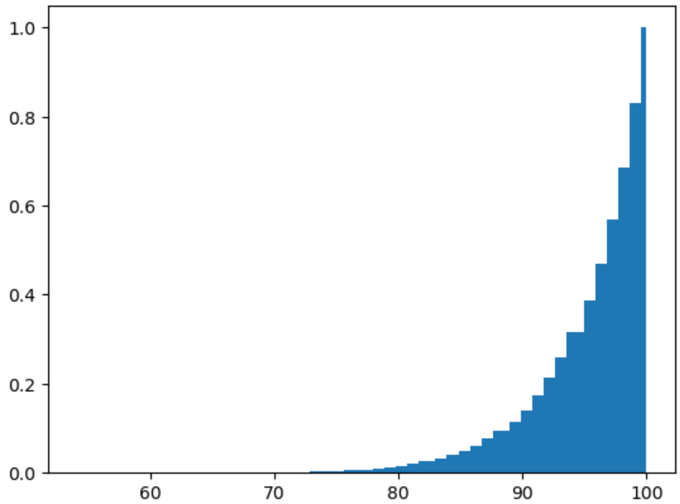


Figure 1: Probabilité pour le joueur aléatoire de gagner avec au plus n coup

Étude probabiliste

Grâce à notre stratégie heuristique, avec l'ajout simple du mode "hunt" au mode aléatoire, nous avons réussi à passer de 95 coups en moyennes à 64 coups. C'est bien ce que l'on observe sur la *Figure 2*.

Implémentation

Notre stratégie heuristique est implémentée dans le fichier `Player/HeuristicPlayer.py`. Il contient 2 mode de jeu

1. Mode "Hunt"
   - ajoute les cases adjacentes de la dernière case touchée à la queue queueCoups
   - tant qu'il y a des éléments dans cette queue et que le jeu n'est pas terminé :
     - joue le dernier élément de la queue
     - si un bateau est touché, ajoute les cases adjacentes à la queue.
   - si il n'y plus d'élément dans la queue, passe en mode Aléatoire.
2. Mode Aléatoire (hérité de RandomPlayer)
   - tant que le dernier coup n'a pas touché:
     - joue aléatoirement un coup dans l'ensemble des coups disponibles

Joueur Probabiliste Simple

Essayons cette fois-ci de prendre en compte l'information "coulé" qui nous donne les cases du bateau coulé. Nous pouvons donc déduire la liste des bateaux coulés et celle de ceux restants.

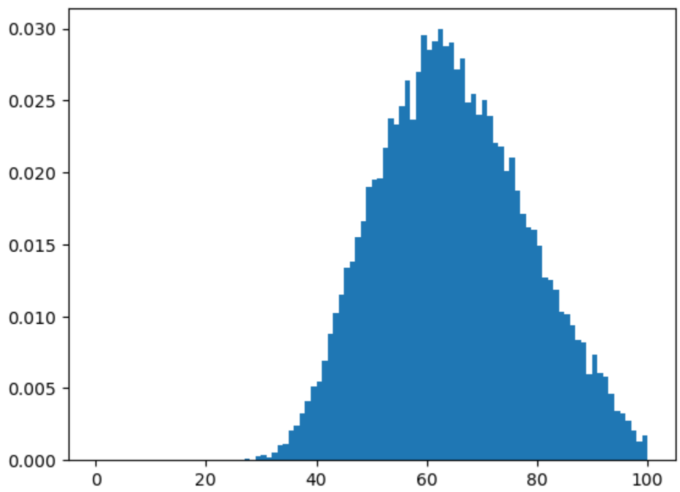
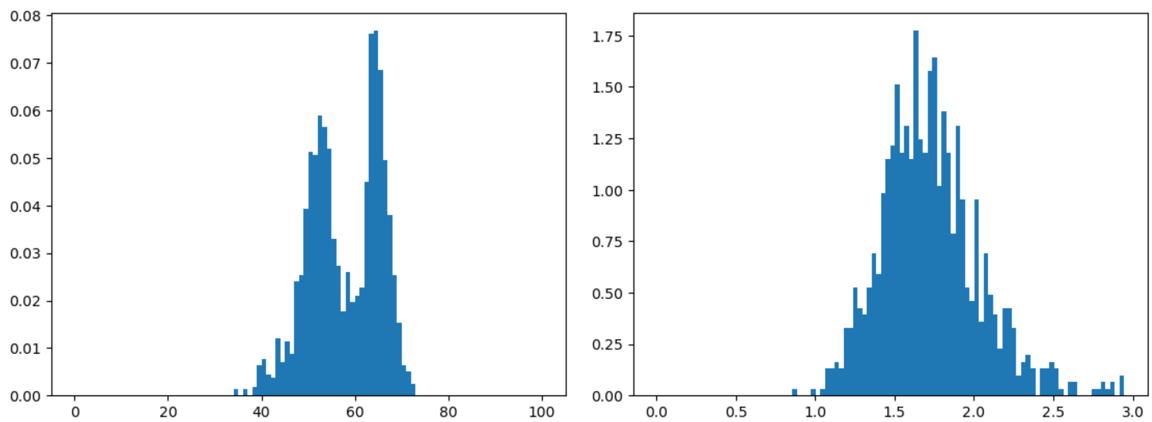


Figure 2: Probabilité pour le joeur Heuristique de gagner avec précisément n coup

Étude probabilistique

La stratégie probabilist simple prend environ 57 coups en moyenne pour terminer. Cependant on constate qu'elle prend beaucoup plus de temps à s'exécuter contrairement aux deux autres stratégies précédentes (en moyenne 1.7 secondes contre significativement moins qu'une seconde) car elle nécessite beaucoup plus de calculs en amont.



à gauche le probablité de gagner avec exactement n coups et à droite un historigramme du temps d'exécution en secondes.

Implémentation

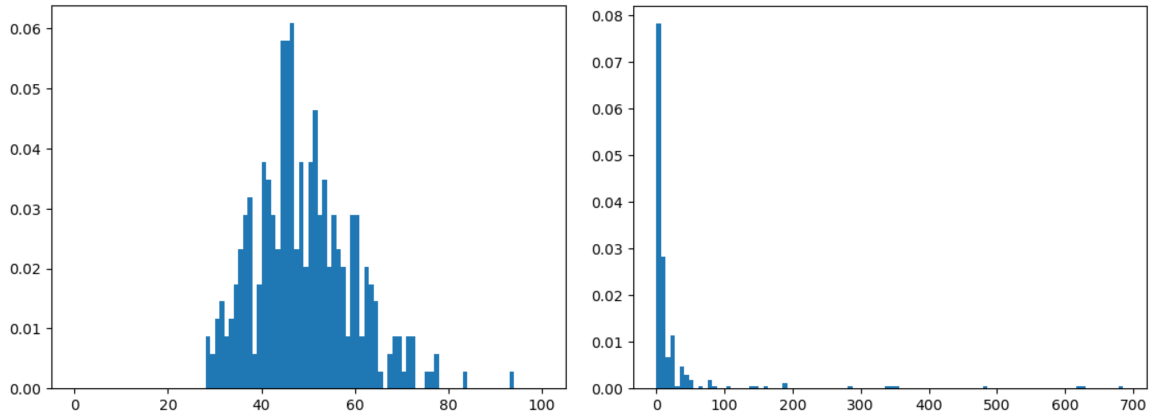
Notre stratégie probabiliste simple est implémentée dans le fichier `Player/HeuristicPlayer.py`. Avant chaque coup, pour chaque position que peut occuper un bateau et qui n'a pas déjà été jouée, il ajoute 1 si le bateau peut être joué. Joue ensuite la case avec le nombre maximal.

Joueur Monte-Carlo

Notre stratégie précédente avait un assez gros défaut, on n'éliminait pas les plateaux où les bateaux pouvaient se superposer. Avec les algorithmes de Monte-Carlo, on joue avec de l'aléatoire certes, mais l'erreur de l'algorithme est minime, négligeable.

Étude probabilistique

La stratégie Monte-Carlo réussit en moyenne en 48 coups. C'est significativement mieux que la stratégie probabiliste. Cependant Monte-Carlo prend en moyenne 30 secondes à s'exécuter. Les calculs peuvent même prendre jusqu'à plus de 10 minutes sur certains plateaux. Typiquement, cela concerne les plateaux où les bateaux sont assez proches l'un de l'autre.



à gauche la probablité de gagner avec exactement n coups et à droite un historigramme du temps d'exécution en secondes.

Implémentation

Une grande partie du code nécessaire à implémenter cette stratégie se trouve dans `src/Player/MCPlayer.py`. Nous y avons implémenter l'algorithme suivant :

- Pour chaque itération, vérifie si les `<nbGen>` anciens plateaux générés sont toujours valides avec la fonction `EngineStats.verify_from_mask()`.
  - si ils sont toujours valides, il les prend en compte pour calculer la probabilité d'apparition de bateaux dans chaque case
  - sinon on retire ceux qui ne marche pas et on en génere des nouveaux avec `self.generate_plateau()`
    - si cela prend plus de `<nbMax>` génération pour en générer, on abandonne cette essai et retourne un tableau vide.
  - pour chaque tableau généré toujours valide, les ajoute dans un tableau temporaire pour récupérer la case avec le plus d'occurence de bateau et joue cette case.
  - si toutes les générations ont été des échecs, joue une case aléatoire

Senseur Imparfait

Introduction

Nous essayons d'implémenter un algorithme d'approche bayésienne pour la recherche d'un objet perdu en mer avec un senseur de fiabilité $p_s < 1$.

Étude

Introduisont 4 variables :

- $y_i \in 0, 1$: est la varialbe aléatoire qui vaut 1 pour la case i qui contient l'objet qu'on cherche et 0 dans les autres.
- $z_i \in 0, 1$: est la variable aléatoire qui vaut 1 en cas de detection et 0 sinon
- $\pi_i \in [0, 1]$: la probabilité a priori de la contenance de l'objet recherché
- $p_s \in [0, 1]$: la probabilité que le senseur détecte l'objet.

Nous avons alors :

$$p(z_i = 1|y_i = 1) = ps$$

$$p(z_i = 0|y_i = 0) = 1$$

$$p(z_i = 0|y_i = 1) = 1 - ps$$

$$p(z_i = 1|y_i = 0) = 0$$

Nous pouvons donc déduire que les deux événements sont indépendants l'un de l'autre. Ainsi la probabilité que le senseur ne détecte pas l'objet présent dans une case k est $\pi_k(1 - p_s)$.

Dans ce cas il suffit de changer la valeur de π_k, nous proposons alors l'algorithme suivant :

- On choisit la cellule avec la plus grande valeur de π_i
- En cas de détection, le programme se termine.
- Sinon, la valeur de π_i devient $\pi_i(1 - p_s)$ et on reprend le programme à l'étape 1.

Le résultat de l'exécution de notre algorithme se trouve dans la *Figure 3.* Pour le code, il se trouve dans le fichier `src/partie4.py`

Conclusion

Dans un premier temps nous avons tenté de prédire les positions possibles d'un batteau en usant de combinatoire et dans un second temps nous avons essayé d'établir des stratégies, de les comparer et de les critiquer. Cependant nous sommes quand même arrivés à la conlusion que le jeu de la bataille navale possède une assez grande dimension combinatoire. Un joueur qui joue aléatoirement ne vaut rien face à un joueur qui calcule minutieusement les positions des bateaux et se sert de toutes les informations données par le jeu. Nous sommes assez confiant que notre algorithme de Monte-Carlo, sur une grande puissance de calcul, peut être un très bon joueur.

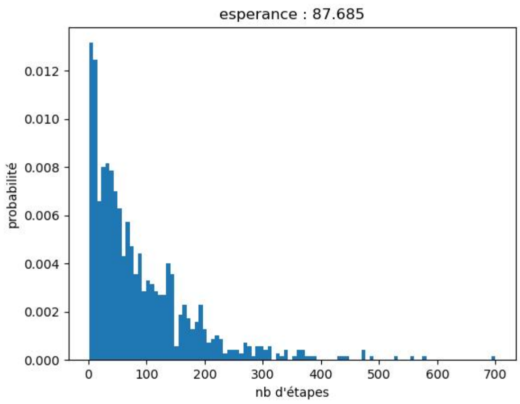


Figure 3: probabilité de détection avec exactement n étapes