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
```

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.

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.

```
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

Tout les joueurs doivent hériter de la classe `AbstractPlayer` et override les fonctions `play`, `reset`, et `name` pour être considiré comme Joueur. La class `AbstractPlayer` s'occupe d'énormement de chose, tel que les intéractions avec le game engine, détecter si le jeu et fini ou pas, gérer le plateau qui récapitule toute la vision qu'à le joueur pour l'instant et surtout la boucle principale. En clair elle regroupe tout le code centrale et nécessaire à un joueur.

Joueur Aléatoire

Étude probabilistique

Soit notre grille contenant $N = 100$ cases, $m = 17$ le nombre de cases occupé paré. La probabilité que le jeu se termine en n action 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. Et c'est bien ce que l'on observe :

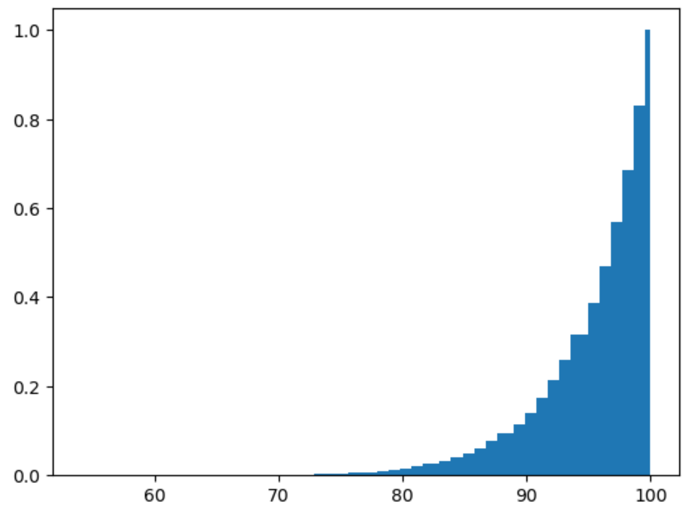


Figure 1: Probabilité de gagner avec au plus n coup

Implémentation

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

- choisi aléatoirement une position dans l'ensemble des coups non-joué

- Joue cette position et la retire de l'ensemble

Joueur Heuristique

Étude probabilistique

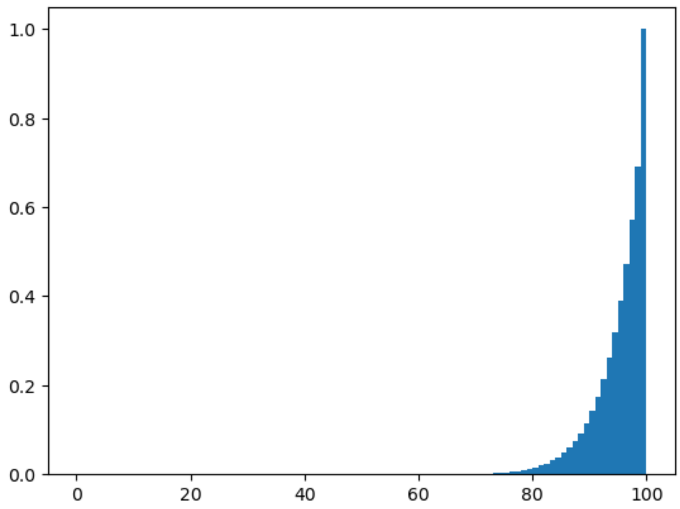


Figure 2: Probabilité de gagner avec au plus n coup

Implémentation

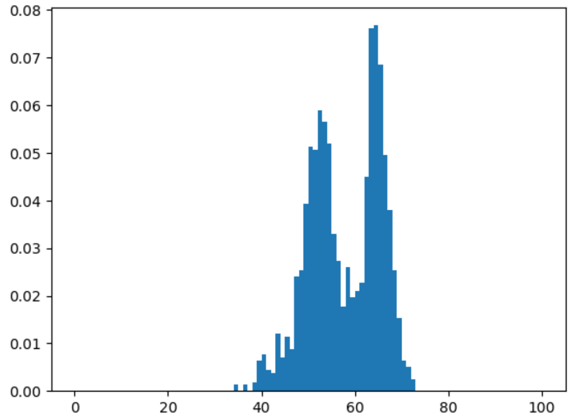
Notre stratégie heuristique est implémenté dans le fichier `Player/HeuristicPlayer.py`. Il contient 2 mode de jeu

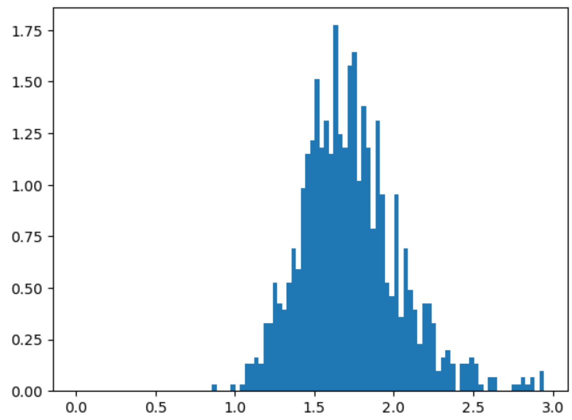
1. Mode "Hunt"
   - Ajoutes les cases adjacentes du derniers à la queue queueCoups
   - tant qu'il y a des éléments dans cette queue et que le jeu n'est pas terminé :
     - Joue le dernier élément de la queue
     - si c'est touché, ajoute les cases adjacentes à la queue.

   -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'est pas un touché:
     - joue aléatoirement un coup dans l'ensemble des coups disponible

Joueur Probabiliste Simple

Étude probabilistique





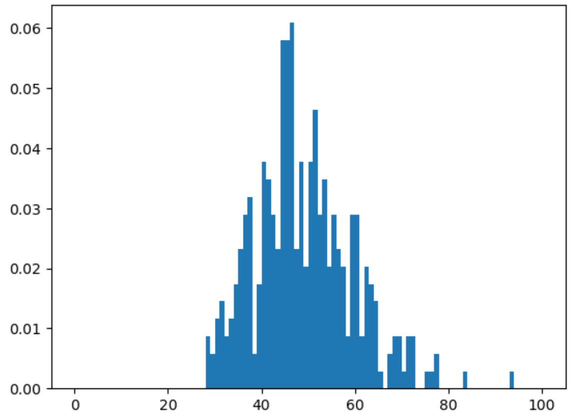
à gauche le probablité de gagner avec exactement n coup et à droite un historigrame du temps d'exécution.

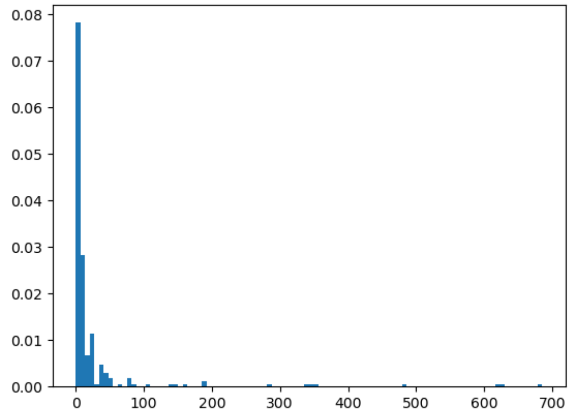
Implémentation

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

Joueur Monte Carlo

Étude probabilistique





à gauche le probablité de gagner avec exactement n coup et à droite un historigrame du temps d'exécution.

Implémentation

Conclusion