Rapport Bataille Navale

Zhile Zhang 21201131

Jiawen Zhang 21117173

Introduction:

On a 4 parties pour ce projet.

L'objectif de ce petit projet est d'étudier les jeux de bataille navale d'un point de vue probabiliste. Nous étudions d'abord la combinatoire du jeu, puis nous proposons une méthode d'optimisation et enfin nous l'intégrons pour le rendre plus réaliste.

Présentation du répertoire :

- Grille.py et test_Grille.py: la class Grille contient des fonction de première partie et les tester.
- Partie2.py et test_Partie2.py: la class Partie2 contient des fonction de première partie et les tester.
- Batatille.py: la class Bataille contient des réalisation des trois fonction `play`, `victoire`, `reset` et les tester.
- Jouer.py et test_Bataille_Jouer.py：la class Jouer contient des fonction qui réalise les stratégies dans la troisièmement partie.
- Sous_marin.py et test_Sous_marin.py: la class Sou_marin contient des fonction de la dernière partie et les tester.

Les quatre parties:

1. Modélisation et fonctions simples

```
Dans la première partie, on fait la modélisation préliminaire et l'écriture des fonctions de base. Nous utiliserons
 souvent les variables suivantes.
```

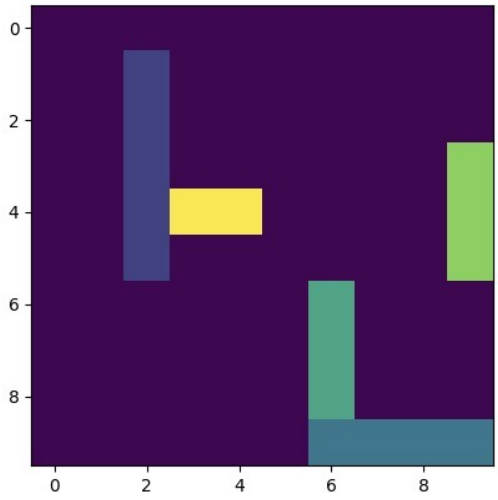
Classe Grille:

- ***Grille :*** La zone de jeu, à l'intérieur de laquelle tous les navires doivent se trouver. Elle est initialisée comme une matrice 10*10 vide.
- ***Bateau :*** Il existe cinq types au total. Chaque bateau sera codé par un identifiant entier: 1 pour le porte-avions, 2 pour le croiseur, 3 pour le contre-torpilleurs, 4 pour le sous-marin, 5 pour le torpilleur.
- ***Position :*** La tête d'un certain bateau. C'est une coordonnée constituée de deux entiers.
- ***Direction :*** Un entier qui représente la direction d'un bateau(1 pour horizontale et 2 pour verticale)

Les fonctions nécessaires :

- `__init__` *:* On crée la classe Grille et on l'initialise une grille de taille 10*10.
- `longBateau` **:** On retourne la longueur du bateau pour simplifier les fonctions suivantes.

- `peut_placer` **:** On crée un tableau (tmp) prenant les dimensions d'un bateau avec toutes ses cases à 0. En partant de l'index "position" on compare horizontalement et verticalement les cases de la grille avec tmp. Si elles sont égales, cela signifie que ces cases ne sont pas occupées et qu'on peut donc placer un bateau sur ces cases.
- `place` **:** Dans chaque direction, en utilisant la fonction précédent `peut_placer`, on peut savoir si cette position peut être placée et si oui, détecter si elle peut être placée à gauche ou à droite.
- `place_alea` **:** On randomise la direction et les coordonnées et les applique dans la fonction `place`.
- `affiche` **:** On utilise `pyplot.imshow(grille)` du module `matplotlib.pyplot`

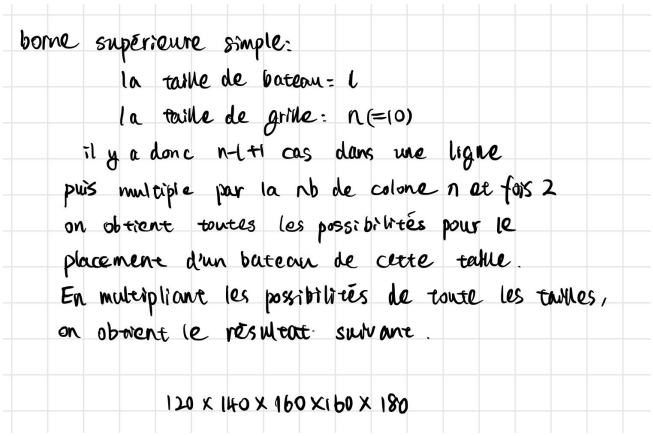


- `eq` **:** On utilise `array_equal` du module `numpy`
- `genere_grille` **:** Créer cinq bateaux aléatoires sur la grille en utilisant `place_alea`.

2. Combinatoire du jeu

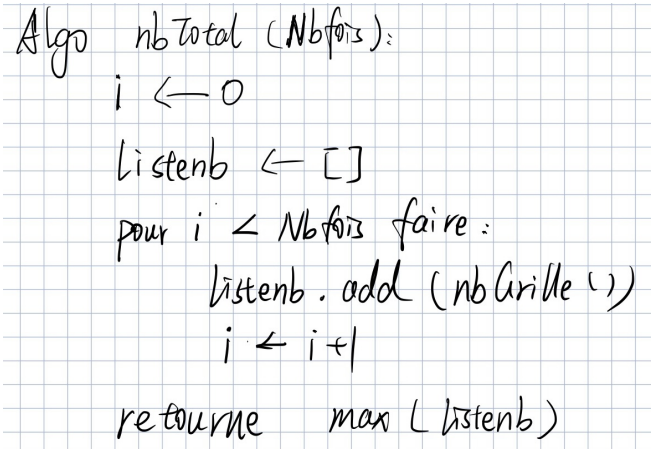
```
Dans cette partie, nous étudions le nombre de mailles possibles dans différentes condition.
```

Classe Partie2:


borne supérieure simple:
la taille de bateau = l
la taille de grille: n(=10)
il y a donc n-l+1 cas dans une ligne
puis multiplié par la nb de colone n et fois 2
on obtient toutes les possibilités pour le placement d'un bateau de cette taille.
En multipliant les possibilités de toute les tailles, on obtient le résultat suivant.

120 x 140 x 160 x 160 x 180

2. Pour calculer le nombre de façons de placer un bateau donné sur une grille vide, on parcourt toute la grille et si une position peut être placée(`peut_placer`) alors nb plus 1, nomée `nbUnBateaux`. Le résultat final est la valeur de nb.
3. Nous pouvons calculer le nombre de grilles pour une liste complète de navires de cette manière, mais elle est moins précise et présente une erreur plus importante. Pour implémenter cette fonction, on applique la fonction `nbUnBateaux` et la bouclons 5 fois, c'est le résultat d'une liste de cinq bateaux.
4. On crée une grille G2 générée aléatoirement par la fonction `genere_grille`, *et la compare avec la grille donn*ée G1. Dans chaque boule si G1 et G2 ne sont pas égaux(`eq`), randomiser la grille de G2 et ajouter un au compteur. Se termine à égalité et renvoie le compteur.
5. Nous appelons la fonction `nbGrille` N fois, pouvons obtenir N nombre de grilles générées, et nous prenons la plus grande valeur parmi N, qui est la valeur la plus proche le dénombrement du nombre de grilles possibles.



3. Modélisation probabiliste du jeu

```
Dans cette partie, on cree une classe Bataille qui contient une grille aléatoie et des méthodes pour concevoir la stratégie de jeu：
```

Classe Bataille:

- `__init__`**:** Initialiser un object Grille et une grille comprenant la liste des bateaux de manière aléatoire.
- `play` (Tirer sur la grille*): Si une case d'un bateau est touché, changer la case de touche en -2, sinon -1*
- `victoire`**:** Vérifier que tous les points du navire ont été touchés
- `reset`**:** Réinitialisation de la grille

Classe Jouer:

- `__init__`**:** On initialiser un object Bataille et une grille m qui a la même taille que la grille avec la liste des bateaux du jeu. m est une grille utilise dans la version probabiliste simplifiée pour enregistrer la case de chaque tir et de l'utiliser pour les prochaines statistiques de probabilité.

1)Version aléatoire:

```
Cette stratégie consiste à frapper les coordonnées au hasard (sans les répéter).

Supposons que n actions permettent de gagner la bataille, alors cela revient à prendre Y cases sur 100 avec tous les
bateaux se trouvant sur ces Y cases, car 17 ≤ Y ≤100. On peut dériver la probabilité que le jeu peut fini dans Y
 actions.
```

Y la valeur qui compte le nombre de coups jusqu'à ce qu'on ait touché tous les bateaux:

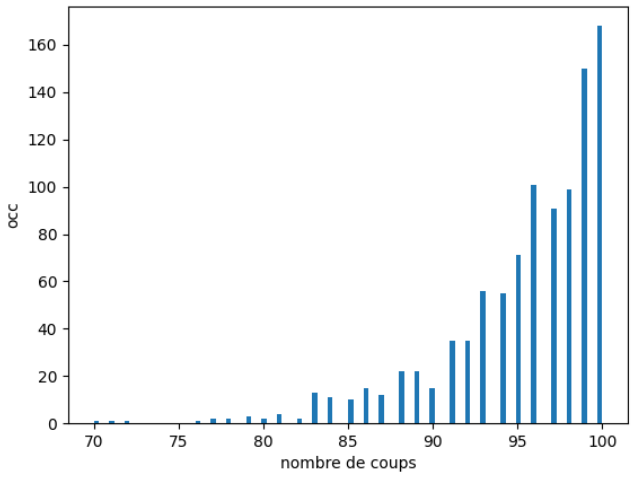
$Y \epsilon [[17, 100]]$ et $\mathbb{P}(Y = k) = \frac{\binom{N-17}{k-17}}{\binom{N}{k}}$

alors $\mathbb{E}(Y) = \sum_{17 \leq k \leq 100} k * \mathbb{P}(Y = k)$ > 100

La fonction pour cette version:

- `aleatiore`**:** Lorsque la partie n'est pas gagnée, nous tirons à des coordonnées aléatoires et si une case à cette coordonnée n'est pas touchée, la fonction de jeu est appliquée pour modifier la valeur et le compteur est ajouté d'une unité. Renvoie la valeur du compteur.

- On teste cette version 1000 fois et on obtient donc un échantillon de 1000 coups. On trace le graphique suivant:

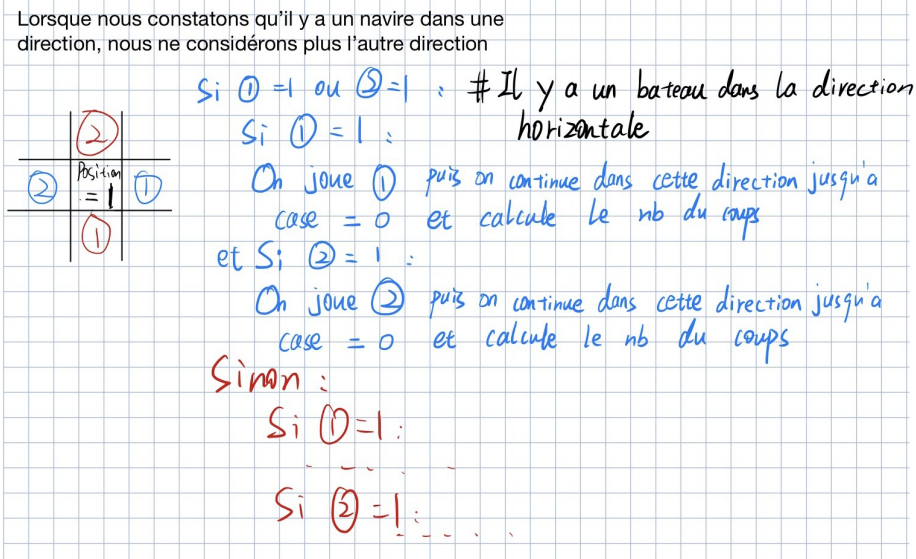



```
La moyenne sur 1000 fois play est de 95.322 coups pour gagne qui est proche de la valeur théorique.
```

2)Version heuristique:

```
Dans cette stratégie, nous commençons toujours par cibler les cases de manière aléatoire comme précédemment, mais
 nous divisons en deux cas différents le prochain ciblage :
Si le ciblage précédent échoue, les nouvelles coordonnées sont à nouveau choisies au hasard.
S'il réussit, les 4 cases autour de la case touchée sont sélectionnées à tour d rôle pour déterminer sa direction et
ainsi pouvoir le couler.
```

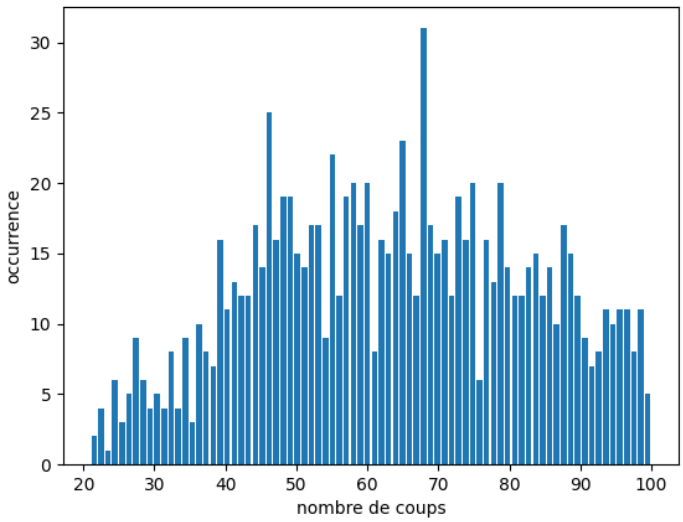
La explication suivant correspond à la fonction `FourDirection(x,y,grille,type)`



La fonction pour cette version:

- `heuristique`: Lorsque la partie n'est pas gagnée, nous tirons sur une coordonnée aléatoire et si la case à cette coordonnée n'est pas touchée, la fonction de jeu est appliquée pour modifier la valeur ; si la grille est vide, le compteur est incrémenté d'une unité et s'il y a un navire à l'intérieur, la même opération est effectuée et la valeur explorée par la fonction `FourDirection` est ajoutée.Renvoie la valeur du compteur.

- On teste cette version 1000 fois et on obtient donc un échantillon de 1000 coups. On trace le graphique suivant:




```
La moyenne sur 1000 fois play est de 65.33 coups pour gagne qui est beaucoup plus faible qu'auparavant.
```

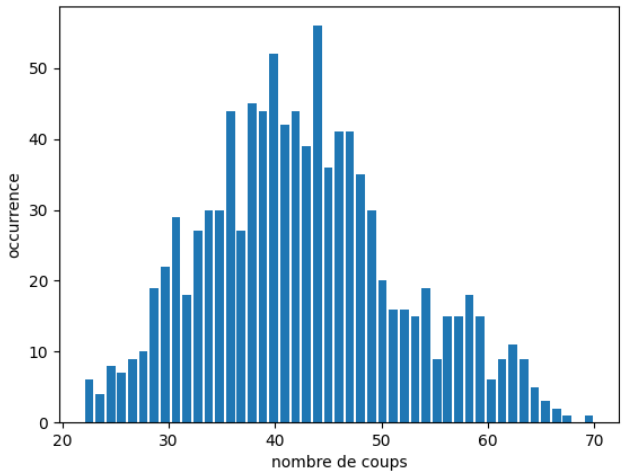
3)Version probabiliste simplifiée :

```
Nous comparons la taille des navires avec les points disponibles dans la grille pour arriver à la grille où les
 navires ont le plus de chances d'apparaître. Nous choisissons un navire au hasard dans la liste des navires, puis
 nous parcourons la grille en calculant la probabilité que chaque case contienne le navire et en sélectionnant les
coordonnées de la case ayant la plus forte probabilité. On détermine ensuite si cette case a un navire et on la
 compte.
```

La fonction pour cette version:

- `PrbSimple`**:** Lorsque la partie n'est pas gagnée, nous choisissons un navire au hasard dans la liste des navires, puis parcourir la grille et calculer la probabilité d'avoir ce navire dans chaque case. Choisissez les coordonnées de la case avec la plus grande probabilité.Si la case à cette coordonnée est vide, appliquez la fonction play et changez la valeur de la case correspondant en m(une grille vide) à -2. Si la valeur de cette case est 1 ou 2 ou 3 ou 4 ou 5, alors changez la valeur de la cellule correspondante dans m en 1 et explorer les environs de ce navire. Nous allons ensuite retirer ce navire de la liste des navires. Renvoie la valeur du compteur.

- On teste cette version 1000 fois et on obtient donc un échantillon de 1000 coups. On trace le graphique suivant:




```
La moyenne sur 1000 fois play est de 44,249 coups pour gagner qui est plus que l'optimisation précédente.
```

Mais cette hypothèse d'indépendance est fausse, car après avoir sélectionné un navire et trouvé la case la plus probable avec ce navire, lors du tir sur cette case, la case de frappe peut appartenir à d'autres navires, ce qui entraînera un nombre final le nombre du coups inférieur à la valeur réelle.

4. Senseur imparfait : à la recherche de l'USS Scorpion

```
Dans cette partie, on utilise la approche bayésienne pour simuler la recherched'un objet dans une région.
```

Interprétation des variables:

- ***Ps:*** La probabilité que le capteur détecte une objet dans une région.
- ***πi:*** La probabilité a priori qui indique les chances que l'objet s'y trouve pour chaque case.
- ***Yi:*** Pour la cellule où se trouve l'objet , la valeur est 1 et 0 ailleurs.
- ***Zi:*** le résultat d'une recherche en case i, valant 1 en cas de détection et 0 sinon.

1. Ps est la probabilité que le capteur détecte l'objet, ce qui signifie que dans la cellule où se trouve l'objet($Yi = 1$), le capteur le recherche ($Zi = 1$)

   donc formulation: $p_s = \mathbb{P}(Zi = 1|Yi = 1)$
2. Selon le titre, nous pouvons savoir que la loi de Yi:

   $Yi \backsim \mathcal{B}(\pi_i)$ et $\sum \pi_i = 1$

   Et pour $P(Zi|Yi)$:

   $\mathbb{P}(Zi = 1|Yi = 1) = p_s$

   $\mathbb{P}(Zi = 0|Yi = 1) = 1 - p_s$

   $\mathbb{P}(Zi = 0|Yi = 0) = 1$

   $\mathbb{P}(Zi = 1|Yi = 0) = 0$
3. `le sous-marin se trouve en case k(Probabilité: πk) et un sondage`

   est effectué à cette case mais ne détecte pas le sous-marin(Probabilité: $1 - p_s$)

   Donc, la probabilité on cherché est la formule suivant: $\mathbb{P}(Zi = 0 \wedge Yi = 1)$, avec

   $\mathbb{P}(Zi = 0 \wedge Yi = 1) = \mathbb{P}(Yi = 1) * \mathbb{P}(Zi = 0|Yi = 1) = \pi_i * (1 - p_s)$
4. Après un sondage infructueux en case k à l'aide de Y_k et Z_k:

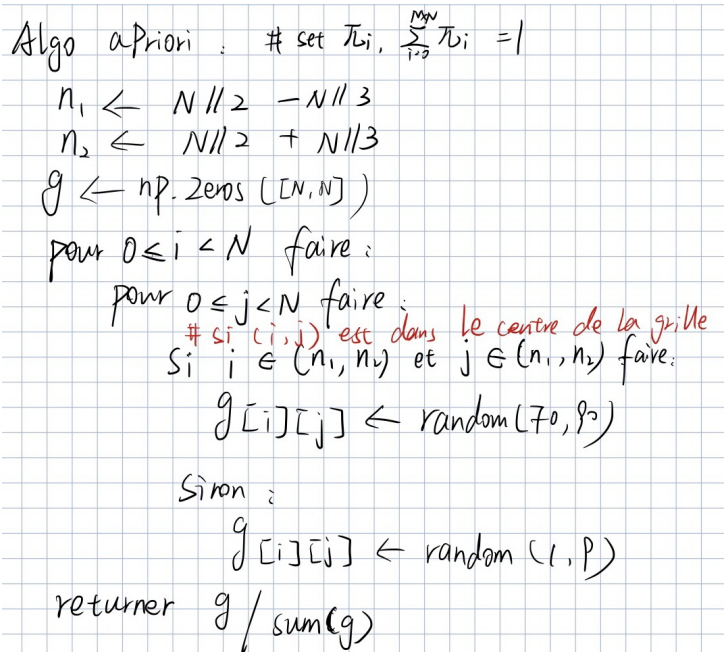
   Pour i = k, par la question précédente, on obtient π_k mise à jour est $\pi_k * (1 - p_s)$

   pour i ≠ k, on les change pas.

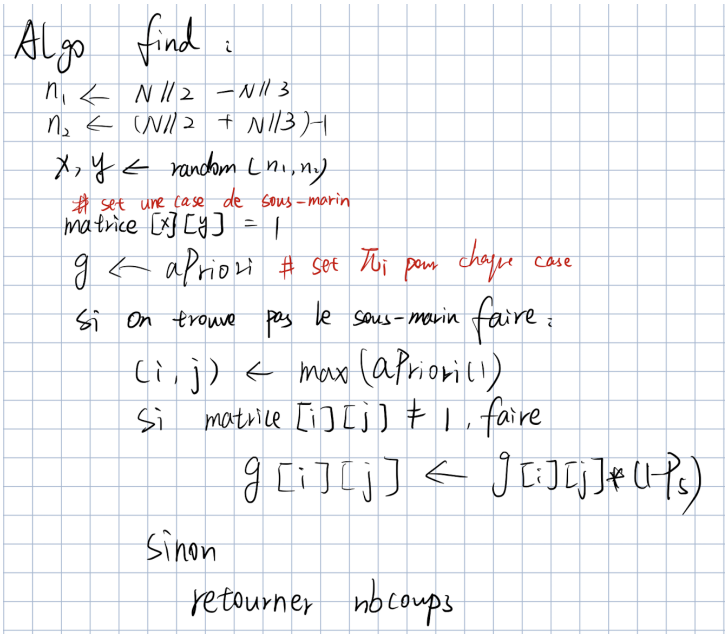
Classe Sous_marin:

- `_ _init_ _(N,Ps)`**:** On crée une grille de la taille N*N, et initialise la valeur de Ps
- `aPriori`**:** On set les valeurs de π_i

```
La mer et la terre peu profondes, c'est-à-dire les bords de la matrice, sont peu susceptibles d'avoir du
 sous-marin, donc les valeurs de toutes les cases de bord sont aléatoires entre (1, 9).
La mer profonde est que la probabilité du centre de la grille est relativement grande, et la valeur est
 aléatoire entre (70, 90).
Puisque toutes les valeurs de pi s'additionnent à 1, nous divisons finalementchaque case par la somme des
 valeurs de toutes les cases pour avoir pi et s'assurer que la somme de pi est 1.
```


Algo apriori : # set π_i, $\sum_{i=0}^{N^2} \pi_i = 1$
$n_1 \leftarrow N//2 - N//3$
$n_2 \leftarrow N//2 + N//3$
$g \leftarrow np.zeros([N,N])$
pour $0 \leq i < N$ faire :
pour $0 \leq j < N$ faire :
si (i,j) est dans le centre de la grille
Si $i \in (n_1, n_2)$ et $j \in (n_1, n_2)$ faire :
$g[i][j] \leftarrow random(70, 90)$
Sinon :
$g[i][j] \leftarrow random(1, P)$
retourner $g / sum(g)$

- `find` :


Algo find :
$n_1 \leftarrow N//2 - N//3$
$n_2 \leftarrow (N//2 + N//3) - 1$
$x, y \leftarrow random(n_1, n_2)$
# set une case de sous-marin
matrice[x][y] = 1
$g \leftarrow apriori$ # set π_i pour chaque case
Si on trouve pas le sous-marin faire :
$(i, j) \leftarrow max(apriori(1))$
Si matrice[i][j] $\neq 1$, faire
$g[i][j] \leftarrow g[i][j] * (1 - P_s)$
Sinon
retourner nbcoups