



ACADÉMIE DE LYON

PROJET DE VALIDATION DE LA FORMATION ISN 2015/2016

BATAILLE NAVALE
IMPLÉMENTATION, RÉOLUTION AUTOMATIQUE ET
APPROCHES PÉDAGOGIQUES

Frédéric Muller

Lycée Arbez Carme

maths.muller@gmail.com

13 MARS 2016

Le code source \LaTeX de ce rapport est disponible sur la page du projet
<https://github.com/Abunix/pyBatNav>

L'ensemble du projet, ainsi que ce rapport, sont sous licence



<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Violence... is the last refuge of the incompetent.

Isaac Asimov

TABLE DES MATIÈRES

1	Présentation du projet	1
1	Le jeu de la bataille navale	1
2	Objectifs du projet	1
3	Liste des modules du projet	2
4	À propos des annexes	2
2	Quelques fonctions utiles	3
1	Coordonnées des cases	3
2	Constantes de direction	3
3	Paramètres en ligne de commande	3
3	Gestion de la grille	5
1	La classe Bateau	5
2	La classe Grille	5
2.1	Présentation	5
2.2	État de la grille	6
2.3	Gestion des espaces vides	6
2.4	Liste des bateaux possibles sur chaque case	6
2.5	Gestion de la flotte	7
2.6	Fin de la partie	7
4	Gestion des joueurs et de la partie	9
1	La classe Joueur	9
2	La classe Partie	9
5	Algorithme de résolution	11
1	Phase de tir en aveugle	11
1.1	Tirs aléatoires	11
1.2	Optimisations	12
2	Phase de tir ciblé	12
2.1	Premier tir	13
2.2	Deuxième tir	14
2.3	Tirs suivants	15
6	Affichage console	17
1	Preliminaires	17
1.1	Constantes graphiques	17
1.2	Effacer le terminal	17

1.3	Fusion des deux grilles	17
1.4	Autres fonctions d’affichage	18
2	Affichage des grilles	18
2.1	Affichage simple de la grille	18
2.2	Affichage de la grille avec ses propres bateaux en gras	18
3	Modes de jeu	20
7	Interface web	21
1	Serveur web	21
2	HTML et CSS	21
3	Script cgi	22
4	Évolutions possibles	22
8	Point de vue pédagogique	23
9	Conclusion	25
	Annexes	29
A	Étude des algorithmes clés	29
1	Gestion de la grille	29
1.1	Détermination des espaces vides	29
1.2	Détermination des bateaux possibles démarrant sur chaque case et des placements possibles de chaque bateaux	30
1.3	Création de bateaux aléatoires	31
2	Détermination des bateaux coulés	32
3	Algorithme de résolution	33
3.1	Optimisation de la phase de tirs en aveugle	33
3.2	Gestion de la file d’attente	37
3.3	Algorithme complet de résolution	41
4	Déroulement de la partie	42
B	Étude statistique des algorithmes de résolution	43
1	Niveau 1	44
2	Niveau 2	45
3	Niveau 3	46
4	Niveau 4	47
4.1	Échantillons de taille nb_echantillons=10	47
4.2	Échantillons de taille nb_echantillons=100	48
4.3	Échantillons de taille nb_echantillons=1 000	49
4.4	Échantillons de taille nb_echantillons=1 000	50
4.5	Conclusion sur le niveau 4	51
5	Niveau 5	52
6	Niveau 6	53

6.1	Seuil égal à 60	53
6.2	Seuil égal à 70	54
6.3	Conclusion sur le niveau 6	54
7	Conclusions	55
C	Codes des caractères graphiques	57
D	Documentation des modules	59
1	Module bn_grille.py	59
1.1	Classe Bateau	59
1.2	Classe Grille	60
2	Module bn_joueur.py	62
2.1	Classe Joueur	62
2.2	Classe Ordi	64
2.3	Classe Partie	67
3	Module bn_console.py	68
3.1	Classe GrilleC	68
3.2	Classe JoueurC	71
3.3	Classe OrdiC	72
3.4	Classe PartieC	76
3.5	Classe MainConsole	77
4	Module bn_stats.py	78
4.1	Classe Stats	78
E	L'algorithme en action	81

PRÉSENTATION DU PROJET

1 Le jeu de la bataille navale

Le jeu de la bataille navale est un jeu qui se joue à deux joueurs.

Chaque joueur dispose d'une grille sur laquelle il place des bateaux rectangulaires de différentes tailles et essaie, à tour de rôle, de deviner l'emplacement des bateaux de l'adversaire par des tirs successifs, ce dernier annonçant à chaque coup « manqué » ou « touché » (on ne rejoue pas après avoir touché). J'ai pris le parti de ne pas annoncer « coulé » lorsque toutes les cases d'un bateau ont été touchées pour rendre l'algorithme de résolution un petit peu plus intéressant.

Les bateaux peuvent être placés horizontalement ou verticalement et deux bateaux ne peuvent pas se trouver sur des cases adjacentes.

Les paramètres de jeu retenus dans ce projet sont ceux du jeu original, mais ils peuvent être facilement modifiés, à savoir que la grille est un carré 10 cases de côté et la composition de la flotte est la suivante :

- Un bateau de 5 cases
- Un bateau de 4 cases
- Deux bateaux de 3 cases
- Un bateau de 2 cases

Notons tout de suite quelques implications stratégiques de ces règles qui seront utilisées dans le projet :

- Le plus petit bateau étant de taille 2, il suffit de ne tirer que sur une cases sur deux (imaginez les cases noires d'un damier) lors de la recherche d'un bateau.
- Une fois qu'un bateau a été coulé, on peut éliminer de la recherche toutes ses cases adjacentes.
- On peut concevoir ce jeu comme un jeu à un seul joueur jouant sur une grille aléatoire.

2 Objectifs du projet

Mes objectifs ont été les suivants :

- Définir une structure de données pour modéliser la grille de jeu, ainsi que les joueurs.
- Implémenter un algorithme de résolution par l'ordinateur qui soit le plus performant possible (en nombre de coups ainsi qu'en temps de résolution d'un grille) et en faire une étude statistique complète.
- Avoir une interface permettant de jouer contre l'ordinateur, d'une part en mode console et, d'autre part, avec le module tkinter. J'ai également rajouté une inter-

face web en HTML5 utilisant un canvas, une communication via un script cgi et quelques fonctions en Javascript, à titre de démonstration.

3 Liste des modules du projet

Afin de faciliter les développement et la maintenance du projet, celui-ci a été décomposé en un certain nombre de modules :

- `main.py` : le programme principal. Il permet, via un argument `--interface` en ligne de commande de choisir l'interface de jeu (`console` ou `tkinter`).
- `bn_utiles.py` : contient quelques fonctions utiles ainsi que les constantes du projet.
- `bn_grille.py` : gère la grille et les bateaux.
- `bn_joueur.py` : gère les joueurs et implémente l'algorithme de résolution.
- `bn_stats.py` : fournit les outils d'analyse statistique des résultats de l'algorithme de résolution.
- `bn_console.py` : toute l'interface en mode console, et l'étude statistique de l'algorithme de résolution.

L'ensemble du code source du projet a été déposé sur mon compte GitHub à l'adresse <https://github.com/Abunix/pyBatNav>

4 À propos des annexes

Vu l'ampleur du projet et la contrainte de taille du rapport, de nombreux points sont abordés en annexe, à partir de la page 29. On y trouve notamment les principaux algorithmes du programme, l'étude statistique des différents algorithmes de résolution ainsi que des points techniques du code et un exemple de résolution pas à pas.

QUELQUES FONCTIONS UTILES

1 Coordonnées des cases

Les cases de la grille sont codées par des tuples (x,y) , où x est la colonne et y la ligne. Aussi nous utilisons la fonction `alpha(case)` qui, à partir des coordonnées, retourne sa représentation naturelle (par exemple 'B4') et `coord(case_alpha)` qui réalise l'opération réciproque.

Ces deux fonctions utilisent le code ASCII.

2 Constantes de direction

Les constantes suivantes, définies dans le module `bn_utiles.py`, indiquent les différentes directions, et sont utilisées dans tout le projet :

- `DROITE = (1, 0)`
- `GAUCHE = (-1, 0)`
- `BAS = (0, 1)`
- `HAUT = (0, -1)`

Ainsi que :

- `TOUTES_DIR = (1, 1)` (toutes les direction)
- `HORIZONTAL = (1, 0)` (à gauche et à droite)
- `VERTICAL = (0, 1)` (en haut et en bas)

Elles permettent de rendre le code plus clair et plus compact.

3 Paramètres en ligne de commande

Le lancement du programme principal `main.py` admet un paramètre en ligne de commande. Ce paramètre est géré par le module `argparse`. La prototype est le suivant :

```
$ main.py -h
usage: main.py [-h] [--interface INTERFACE]
```

Jeu de bataille navale

optional arguments:

```
-h, --help            show this help message and exit
--interface INTERFACE, -i INTERFACE
                        Choix de l'interface : 'console' ou 'tkinter'
```


GESTION DE LA GRILLE

La gestion de la grille et des bateaux est effectuée dans le module `bn_grille.py`.

1 La classe Bateau

Cette classe, très minimaliste, définit un bateau par sa case de départ `Bateau.start`, sa taille `Bateau.taille` et sa direction `Bateau.sens`. Elle permet de récupérer :

- sa case de fin `Bateau.end`,
- la liste de ses cases occupées `Bateau.cases`,
- la liste de ses cases adjacentes `Bateau.cases_adj`.

2 La classe Grille

2.1 Présentation

Cette classe est au cœur du projet. Elle permet de mémoriser l'état de chaque case de la grille et d'effectuer des opérations comme :

- Gérer la liste des bateaux de la flotte : placer un bateau à une position donnée ou aléatoirement, placer une flotte aléatoire, supprimer un bateau coulé, ou encore garder la trace du plus grand bateau restant à couler.
- Déterminer le nombre de cases vides autour d'une case donnée, dans chacune des directions.
- Déterminer la liste, et le nombre, de bateaux possibles sur chaque case.
- Déterminer lorsque la grille est terminée.

Beaucoup de ces fonctionnalités seront utilisées par l'algorithme de résolution.

Afin de pouvoir faire évoluer les règles, elle prend les paramètres suivants lors de son initialisation :

- `xmax` et `ymax` : les dimensions de la grille
- `taille_bateaux` : la liste des bateaux

Dans la mesure où la grille a deux utilisations différentes (la grille du joueur et la grille de suivi des coups), j'ai d'abord décidé de créer deux classes héritées de `Grille` lors de la conception du projet, `GrilleJoueur(Grille)` et `GrilleSuivi(Grille)`, afin de distinguer leurs méthodes spécifiques. Après coup je me suis rendu compte que cela n'apportait pas d'avantage significatif en terme de qualité de code donc je ne les utiliserai pas, mais elles sont encore présentes dans mon code pour une évolution future du projet.

2.2 État de la grille

L'attribut `Grille.etat` fournit l'état de la grille. C'est un dictionnaire indexé par les tuples $(0,0)$, $(0,1)$, ..., $(9,9)$, dans lesquels la première coordonnée correspond à la colonne de la case et la deuxième à sa ligne.

L'état d'une case peut être :

- 0 : case vide
- 1 : case touchée (ou contenant un bateau)
- -1 : case manquée ou impossible

La méthode `Grille.test_case(self, case)` permet de déterminer si une case est valide et vide, et `Grille.is_touche(self, case)` indique si une case donnée contient ou non un bateau.

Notons également l'utilisation de l'attribut `Grille.vides` qui est la liste des cases vides. Bien entendu, cette classe contient une fonction `Grille.update(self)` de mise à jour de l'état de la grille (liste des cases vides, tailles du plus petit et du plus grand bateau restant).

Enfin, la méthode `Grille.adjacent(self, case)` renvoie la liste de cases adjacentes à une case donnée.

2.3 Gestion des espaces vides

La méthode `Grille.get_max_space(self, case, direction, sens)` renvoie le nombre de cases vides dans une direction donnée. Grâce aux constantes de direction, un seul calcul est nécessaire pour englober tous les cas (horizontal et vertical). L'algorithme est donné en annexe A, page 29.

Si le paramètre `sens=1`, la détermination se fait dans les deux sens (espace libre total horizontal ou vertical).

La méthode `Grille.elimine_cases(self)` parcourt toutes les cases vides et élimine celles dans lesquelles le plus petit bateau ne peut pas rentrer en mettant leur état à -1.

2.4 Liste des bateaux possibles sur chaque case

La méthode `Grille.get_possibles(self)` renvoie d'une part la liste des bateaux possibles démarrant sur chaque case (ainsi que leurs directions) et, d'autre part, la liste des positions (et directions) de départ possibles pour chaque bateau. Pour ce faire on procède en deux temps :

- Dans un premier temps, on parcourt la liste des cases vides et pour chacune de ces cases on détermine, pour chaque bateau et chaque direction (droite et bas), s'il peut démarrer sur cette case. Cela fournit le dictionnaire `Grille.possibles_cases` indexé par les cases et dont les éléments sont une liste de tuples de la forme $(taille, direction)$.

Par exemple : $\{(0,0):[(5,(1,0)), (5,(0,1)), \dots], (0,1): \dots\}$

- Dans un deuxième temps, on "retourne" ce dictionnaire pour obtenir le dictionnaire `Grille.possibles` indexé par les tailles des bateaux et dont les éléments sont une liste de tuples de la forme (case, direction).
Par exemple: `{5: [((0,0), (1,0)), ((0,0), (0,1)), ((1,0), (1,0)), ...], 4: ...}`

Cette méthode va servir à faire plusieurs choses :

- Créer une flotte aléatoire grâce au dictionnaire `Grille.possibles`, dans la méthode `Grille.init_bateaux_alea(self)`.
- Déterminer la case optimale dans l'algorithme de résolution au niveau 5.
- Optimiser la file d'attente dans les algorithmes de résolution.
- Déterminer tous les arrangements possibles de bateaux sur la grille.

L'algorithme complet de cette méthode est donné en annexe A, page 30.

2.5 Gestion de la flotte

La classe `Grille` contient toutes les méthodes nécessaires pour gérer la flotte de bateaux. Les méthodes `Grille.get_taille_max(self)` et `Grille.get_taille_min(self)` mettent à jour respectivement la taille maximum et la taille minimum des bateaux restant à trouver. La méthode `Grille.rem_bateau(self, taille)` permet de supprimer de la liste `Grille.taille_bateaux` un bateau coulé.

2.5.1 Ajout d'un bateau

La méthode `Grille.add_bateau(self, bateau)` permet d'ajouter un bateau (instance de la classe `Bateau`) après avoir testé sa validité via la méthode `Grille.test_bateau(self, bateau)`, et marque ses cases adjacentes comme impossibles.

2.5.2 Création d'un flotte aléatoire

La méthode `Grille.add_bateau_alea(self, taille)` permet d'ajouter un bateau aléatoire de taille donnée sur la grille.

Pour créer une flotte aléatoire, on utilise la méthode `Grille.init_bateaux_alea(self)` dont l'algorithme est donné en annexe A, page 31.

2.6 Fin de la partie

L'attribut `Grille.somme_taille`, initialisé dès le départ avec la classe `Grille`, contient le nombre total de cases à toucher. La méthode `Grille.fini(self)` compare donc ce nombre avec le nombre de cases touchée dans `Grille.etat` pour déterminer si la grille a été résolue.

GESTION DES JOUEURS ET DE LA PARTIE

La gestion des joueurs et du déroulement de la partie se font dans le module `bn_joueur.py` mais les classes `Joueur` et `Partie` sont très minimales et ne constituent majoritairement qu'un squelette pour la suite. Elles seront largement héritées, que ce soit par la classe `Ordi` qui implémente l'algorithme de résolution, que pour les différentes interfaces (console et graphique).

1 La classe `Joueur`

Lors de son initialisation, on peut donner un nom au joueur et on initialise sa grille de jeu `Joueur.grille_joueur`, la grille de l'adversaire `Joueur.grille_adverse` ainsi que sa grille de suivi des coups `Joueur.grille_suivi`.

On en profite aussi pour initialiser quelques variables d'état comme la liste des coups déjà joués, le nombre de coups joués et les cases des bateaux coulés détectés.

Les méthodes principales de cette classe sont `Joueur.tire(self, case)` qui permet de tirer sur une case et d'avoir en retour le résultat du coup (y compris si le coup n'est pas valide) et `Joueur.check_coules(self)` qui détermine les bateaux coulés dont l'algorithme est donné en annexe A, page 32.

Notons l'attribut `Joueur.messages` qui est une liste contenant différents messages d'information (comme par exemple "A2 : Touché", ou encore les messages indiquant comment l'algorithme résout la grille). Lors de l'affichage des messages, il suffit de vider cette liste grâce à des `pop(0)` successifs dans la méthode `Joueur.affiche_messages(self)` en affichant chaque élément pour avoir un suivi.

2 La classe `Partie`

Ici encore, un squelette et des méthodes très générales pour une classe qui sera héritée dans les interfaces.

Elle se contente de définir l'adversaire, de placer les bateaux du joueur et de récupérer les paramètres de l'adversaire (sa grille et le coup qu'il vient de jouer).

Notons l'instruction `isinstance(self.adversaire, Ordi)` qui permet de savoir si l'adversaire est l'ordinateur.

À la base je voulais faire un mode de jeu en réseau et c'est ici que se seraient trouvées les instructions de communication.

L'algorithme décrivant le déroulement de la partie est donné en annexe A, page 42.

ALGORITHME DE RÉOLUTION

L'algorithme de résolution est implémenté dans la classe `Ordi(Joueur)` du module `bn_joueur.py`, dans la méthode `Ordi.coup_suivant(self)`. L'attribut `Ordi.case_courante` contient la case qui est entrain d'être jouée.

Il fonctionne en deux temps : dans un premier temps une phase de tirs en aveugle et, une fois qu'une case a été touchée, une phase de tirs ciblés jusqu'à ce que le bateau soit coulé.

Différents algorithmes de résolution sont implémentés et choisis avec l'attribut `Ordi.niveau` :

- `Ordi.niveau=1` : Uniquement des tirs aléatoires en aveugle et pas de tirs ciblés.
- `Ordi.niveau=2` : Tirs aléatoires et phase de tirs ciblés.
- `Ordi.niveau=3` : Tirs aléatoires sur les cases noires et phase de tirs ciblés.
- `Ordi.niveau=4` : Détermination de la case optimale par des échantillons et phase de tirs ciblés.
- `Ordi.niveau=5` : Détermination de la case optimale par le nombre de bateaux possibles sur chaque case et phase de tirs ciblés.
- `Ordi.niveau=6` : Idem niveau 5, mais dès que le nombre de cases vides passe en-dessous de `Ordi.seuil`, l'algorithme énumère toutes les répartitions de bateaux possibles.

Une étude statistique de chacun de ces algorithmes est donnée en annexe B, page 43 et un exemple de résolution pas à pas (avec le niveau 5) est donné dans l'annexe E, page 81. L'algorithme complet de résolution est donné en annexe A, page 41.

1 Phase de tir en aveugle

Lors de cette phase, l'algorithme va commencer par éliminer les zones dans lesquelles le plus petit bateau restant ne peut pas rentrer, puis choisir une case aléatoire parmi les cases vides. Le choix de cette case va, en grande partie, déterminer les performances de l'algorithme.

1.1 Tirs aléatoires

1.1.1 Méthode naïve

La méthode la plus naïve est de choisir une case de manière uniforme dans la liste des cases vides avec l'instruction `random.choice(liste)`.

Cette méthode est celle qui peut être attendue d'un élève (qui peut se contenter de ça pour l'ensemble de son algorithme de résolution) et la seule qui permette de résoudre la grille uniquement en faisant des tirs en aveugle.

1.1.2 Méthode des cases noires

Dans la mesure où le plus petit bateau est de taille 2, si on imagine la grille comme un damier, un bateau recouvre obligatoirement une case noire. On peut donc se contenter de viser uniquement parmi ces cases.

Une case (x, y) est une case noire si, et seulement si, $(x + y) \% 2 == 0$.

1.2 Optimisations

1.2.1 Méthode par échantillonnage

Dans la méthode suivante, on va essayer d'optimiser les tirs en aveugle en estimant la probabilité de chaque case de contenir un bateau.

Pour ce faire, on va créer un échantillon de n répartitions aléatoires de bateaux sur les cases vides restantes et on va compter, sur chacune, le nombre de fois où elle a été occupée ce qui va nous donner une densité de probabilités pour chaque case. Cette partie est gérée par la méthode `Grille.case_max_echantillons(self, nb_echantillons)` dont l'algorithme est donné en annexe A, page 33.

Les performances en nombre d'essais sont satisfaisantes, mais le temps de calcul beaucoup trop élevé.

1.2.2 Méthode par énumération de tous les cas

On peut essayer de déterminer tous les arrangements de bateaux possibles sur la grille à chaque coup, de manière récursive. Cette approche semble optimale mais malheureusement, vu le nombre de configurations, cette approche est inutilisable sur une grille standard. Elle est néanmoins implémentée dans la méthode `Grille.case_max_all(self)`.

1.2.3 Méthodes par comptage

Dans cette méthode, on va compter pour chaque case le nombre de bateaux possibles contenant celle-ci et tirer sur celle qui en contient le plus.

La méthode `Grille.case_max(self)` renvoie la case optimale, ainsi que le nombre de bateaux qu'elle contient. L'algorithme est donné en annexe A, page 34.

Bien que locale (on ne regarde pas la répartition de tous les bateaux ensemble sur la grille), cette méthode est très efficace.

2 Phase de tir ciblé

Lors de cette phase, on va utiliser une file d'attente dans la liste `Ordi.queue` qui va contenir la liste des prochaines cases à viser. On va également garder la trace des cases touchées sur ce bateau dans la liste `Ordi.liste_touchees`, ainsi que sa première case touchée dans `Ordi.case_touchee`.

2.1 Premier tir

2.1.1 Création de la liste des cases adjacentes

Lors du premier tir touché après la phase de tir en aveugle, l'attribut `Ordi.case_touchee` reçoit la case courante qui sera également ajoutée à `Ordi.liste_touchees`. Puis on va remplir la file d'attente avec les cases adjacentes à la case touchée.

L'algorithme va également vérifier si le plus petit bateau ne peut pas rentrer dans une direction. Imaginons, par exemple, que le plus petit bateau restant soit de taille 3 et qu'on vienne de toucher la case (3,0) dans la configuration suivante :

	0	1	2	3	4	5	6	7	8	9
0				X						
1										
2				O						

Le bateau de taille 3 ne rentre pas verticalement

Il ne sert alors à rien de mettre la case (3,1) dans la file d'attente car le plus petit bateau de taille 3 ne peut pas rentrer verticalement en (3,0).

La construction de la liste des cases adjacentes possibles est effectuée par la méthode `Ordi.add_adjacentes_premiere(self)` dont l'algorithme est donné en annexe A, page 37.

2.1.2 Optimisation de la file d'attente

Pour l'optimisation de cette phase, on va classer la file d'attente en ordre décroissant de nombre de bateaux possibles avec la méthode `Ordi.shuffle_queue(self)`.

Cette optimisation est un petit peu délicate. Une fois qu'une case a été touchée, l'algorithme va tester ses 4 (au maximum) cases adjacentes et les ranger en ordre décroissant du nombre de bateaux possibles. C'est le rôle de la méthode `Grille.case_max_touchee(self, case_touchee)`, dont l'algorithme complet est donné en annexe A, page 38.

Notons (x, y) les coordonnées de la case touchée et intéressons nous au nombre de bateaux possibles sur les cases adjacentes horizontales (pour les verticales, c'est exactement la même chose). Pour chaque taille de bateau à couler possible contenant `case_touchee` il faudra distinguer trois cas :

- 1) Le bateau est à gauche de `case_touchee` et se termine sur cette case. Dans ce cas on augmente de 1 le nombre de possibilités de la case à gauche $(x-1, y)$
- 2) Le bateau est à cheval sur `case_touchee`. Dans ce cas on augmente de 1 le nombre de possibilités de la case à gauche $(x-1, y)$ et de celle à droite $(x+1, y)$
- 3) Le bateau est à droite de `case_touchee` et commence sur cette case. Dans ce cas on augmente de 1 le nombre de possibilités de la case à droite $(x+1, y)$

Exemple :

Imaginons que, sur une grille vierge, on vienne de toucher la case de coordonnées (5,0) et regardons le nombre de façons de placer le bateau de taille 4 à gauche et à droite :

- 1) Le bateau rentre à gauche de la case (5,0) :

	0	1	2	3	4	5	6	7	8	9
0			X	X	X	X				

La case (4,0) est augmentée de 1

- 2) Le bateau est à cheval sur la case (5,0) (2 possibilités) :

	0	1	2	3	4	5	6	7	8	9
0				X	X	X	X			

	0	1	2	3	4	5	6	7	8	9
0					X	X	X	X		

Les cases (4,0) et (6,0) sont augmentées de 2

- 3) Le bateau rentre à droite de la case (5,0) :

	0	1	2	3	4	5	6	7	8	9
0						X	X	X	X	

La case (6,0) est augmentée de 1

Au final, la case (4,0) admet 3 bateaux horizontaux de taille 4 et idem pour la case (6,0).

Si la case (3,0) avait été jouée et manquée nous aurions obtenu 1 bateau horizontal de taille 4 possible sur la case (4,0) et 2 sur la case (6,0) :

	0	1	2	3	4	5	6	7	8	9
0				O		X				

Une fois que le compte des bateaux possibles a été effectué sur chacune des cases adjacentes, on crée une liste `probas_liste` contenant des tuples de la forme `(case, probas[case])` que l'on ordonne en ordre décroissant de possibilités grâce à l'instruction `sorted(probas_liste, key=lambda proba: proba[1], reverse = True)` et que l'on retourne.

2.2 Deuxième tir

Lors du deuxième tir (c'est à dire sur la première case de la file d'attente), on peut soit toucher, soit manquer.

2.2.1 Deuxième tir touché

Si on touche alors, grâce à la méthode `Ordi.update_queue_touche(self)`, on détermine la direction du bateau (horizontal ou vertical) en comparant les coordonnées de `Ordi.case_courante` et `Ordi.case_touchee` et on enlève les cases de la file d'attente qui ne sont pas dans la bonne direction. On ajoute enfin la case à l'extrémité de la configuration créée à la file d'attente et on met à jour la liste `Ordi.liste_touchees`. L'algorithme de cette partie est donné en annexe A, page 39.

2.2.2 Deuxième tir manqué

Si on manque, alors on a peut-être bloqué une direction. La méthode `Ordi.update_queue_manque(self)` se charge de cette vérification et élimine la case en face de la case jouée si besoin de la file d'attente. Regardons un exemple. Imaginons que le plus petit bateau à trouver soit de taille 4 et que la première case touchée soit la case (3,0). Nous venons de manquer la case (4,0). Alors le bateau de taille 4 ne rentre plus horizontalement et on peut éliminer la case (2,0) :

	0	1	2	3	4	5	6	7	8	9
0	O			X						

À ce niveau, le bateau de taille 4 rentre horizontalement.

	0	1	2	3	4	5	6	7	8	9
0	O			X	O					

Après ce coup, le bateau de taille 4 ne rentre plus horizontalement et on peut éliminer la case (2,0).

Pour ce faire, on détermine la direction dans laquelle on vient de tirer en comparant les coordonnées de `Ordi.case_courante` et `Ordi.case_touchee` et on regarde si le plus petit bateau rentre dans la direction en face. L'algorithme de cette partie est donné en annexe A, page 40.

2.3 Tirs suivants

Une fois que la direction du bateau est déterminée, à chaque fois qu'on touche une case, on ajoute à la file d'attente sa case adjacente dans la bonne direction si elle est vide.

Enfin on s'arrête lorsque la file d'attente est vide (on a manqué les deux extrémités) ou lorsque la taille du bateau touché est égale à la plus grande taille du bateau sur la grille avec `Ordi.test_plus_grand(self)` et, dans ce cas, on vide la file d'attente.

Au prochain tour, on sait qu'un bateau vient d'être coulé lorsque la file d'attente est vide et `Ordi.liste_touchees` ne l'est pas. Dans ce cas on marque ses cases adjacentes comme impossibles et on l'enlève de la liste des bateaux à chercher. On n'a plus alors qu'à repartir dans une phase de tir en aveugle jusqu'à ce qu'on ait terminé la grille.

AFFICHAGE CONSOLE

Le module `bn_console.py` implémente l'interface en mode console.

L'idée de cette interface est de rendre hommage au style de jeu des années 80 en essayant d'en garder au maximum l'esprit.

Il est fortement conseillé de mettre la fenêtre de terminal en plein écran afin d'avoir un affichage complet des grilles.

1 Préliminaires

1.1 Constantes graphiques

Pour afficher les grilles j'utilise des caractères graphiques en unicode (famille Box Drawing de codes U2500 à U257F). Ceux-ci donnent tous les outils afin de fabriquer des grilles, y compris avec des caractères gras. Pour des raisons de commodité, le code de chacun des caractères utilisés est stocké dans une constante (par exemple `CAR_CX=u'\u253C'` correspond à la croix centrale `+`). La liste complète des codes des caractères utilisés est donnée en annexe C page 57.

1.2 Effacer le terminal

Le module `os` permet d'une part d'accéder à la version du système d'exploitation avec `os.name` et, d'autre part, de lancer des commandes système avec `os.system(commande)`. La combinaison de ces deux commandes permet facilement de pouvoir effacer l'écran en utilisant la commande `cls` sous Windows et `clear` sous Linux.

1.3 Fusion des deux grilles

Lors d'une partie contre un adversaire, il faut pouvoir afficher côte à côte la grille de suivi du joueur ainsi que sa propre grille avec, au fur et à mesure, les coups joués par l'adversaire. Afin de réaliser cette opération on utilise la fonction `fusion(chaine1, chaine2)`. Celle-ci prend en entrée deux chaînes de caractères et retourne la chaîne fusionnée de la façon suivante : chaque chaîne est convertie en liste en prenant comme séparateur le caractère de retour de ligne `'\n'` grâce à la méthode `String.split('\n')`. Ensuite, en prenant à tour de rôle les éléments de chacune des listes et en insérant un caractère de trait vertical entre les deux on crée la chaîne fusionnée.

Le résultat peut être vu plus bas en page 20.

1.4 Autres fonctions d’affichage

La fonction `centre(chaine, longueur)` centre la chaîne sur un espace de longueur donnée en insérant le nombre d’espaces nécessaires. Cette fonction sera utilisée pour l’affichage des noms des joueurs.

La fonction `boite(texte, prefixe, longueur)` permet d’encadrer le texte dans une boîte de longueur donnée, chaque ligne étant précédée d’un préfixe. Cette fonction sera utilisée pour afficher la liste des messages pour chaque joueur à chaque tour, les préfixe servant à identifier l’auteur du message.

Notons enfin qu’afin de pouvoir réutiliser le code de ce module dans d’autres contextes (comme une interface en `tkinter`), la fonction `print(*args)` a été encapsulée dans une fonction `info(*args)` de sorte qu’il suffit de surcharger cette dernière pour envoyer l’affichage ailleurs (par exemple dans une boîte de texte dans une fenêtre graphique)

2 Affichage des grilles

La classe `GrilleC(Grille)` hérite de la classe `Grille` en ajoutant uniquement les fonctions d’affichage.

2.1 Affichage simple de la grille

La méthode `GrilleC.make_chaine(self)` crée la chaîne de caractères de la grille simple. En plus des coins, chaque case utilise 3 caractère horizontaux (ce qui permet de centrer un symbole) et 1 caractère vertical.

Pour cet affichage, on crée les lignes les unes après les autres (dans deux boucles imbriquées) en marquant les cases suivant les valeurs de `Grille.etat`. La seule subtilité provient des deux premières et de la dernière ligne (à cause des coins).

2.2 Affichage de la grille avec ses propres bateaux en gras

Cette partie est beaucoup plus délicate. L’idée est d’afficher une grille en entourant ses propres bateaux et en marquant les coups joués par l’adversaire (sa grille de suivi). C’est le rôle de la fonction `GrilleC.make_chaine_adverse(self, grille)`, où `grille` est soit sa propre grille de jeu, soit celle de l’adversaire si on veut tricher (pour les tests bien sûr...) ou en fin de partie, si on a perdu, pour avoir la solution. Par convention, comme `grille` est une grille de jeu, nous allons noter dans les explications les seuls états possibles par 1 si la case est occupée par un bateau et 0 sinon (ou si la case est hors grille).

Les contraintes que nous nous fixons sont les suivantes :

- Les bords des bateaux doivent être en gras
- Les séparations à l’intérieur d’un bateau doivent être en clair
- Lorsque deux bateaux se touchent par un coin, il faut bien sûr que ces coins soit en gras (soit une croix en gras)

Le bord de chaque case de la grille se fera sur la ligne du bas, la séparation verticale de gauche et le coin en bas à gauche. Le cas de la ligne horizontale juste sous les lettres sera fait à part, ainsi que la dernière ligne horizontale et la dernière ligne verticale de la grille (à cause du bord).

- 1) Première ligne, sous les lettres des colonnes : pour chaque case de la ligne 0 on va tester son état, ainsi que l'état de la case de gauche (pour savoir si on est en début ou en fin de bateau, ou au milieu d'un bateau). On obtient les configurations suivantes (la case testée est celle de droite) :

$\begin{array}{c} \\ \hline 0 \end{array}$	$\begin{array}{c} \\ \hline 1 \end{array}$	$\begin{array}{c} \\ \hline 1 \end{array}$	$\begin{array}{c} \\ \hline 0 \end{array}$	$\begin{array}{c} \\ \hline 0 \end{array}$	$\begin{array}{c} \\ \hline 1 \end{array}$
0	1	1	0	0	1

- 2) Lignes suivantes, jusqu'à l'avant dernière : ici c'est plus délicat car il faut tester, en plus de celle de gauche (pour la séparation verticale), la case en-dessous (pour la séparation horizontale) et celle en-dessous à gauche (pour le coin) pour obtenir les 11 configurations suivantes (la case testée est celle en haut à droite) :

$\begin{array}{c} 1 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 1 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 1 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 1 \end{array}$
0	0	0	1	0	1

$\begin{array}{c} 1 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 1 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 1 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$
0	1	0	1	1	0	1	0

Les 5 autres configurations de 4 cases restantes sont impossibles (bateaux collés par un côté).

- 3) Enfin, pour la dernière colonne va juste tester la case en-dessous, et pour la dernière ligne, on va juste tester celle de gauche. Pour la case tout en bas à droite il faudra juste finir en mettant un coin.

$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 1 \end{array}$	$\begin{array}{c} 1 \\ \\ \hline 1 \end{array}$	$\begin{array}{c} 1 \\ \\ \hline 0 \end{array}$
0	1	1	0
$\begin{array}{c} 1 \\ \hline 1 \end{array}$	$\begin{array}{c} 1 \\ \\ \hline 0 \end{array}$	$\begin{array}{c} 0 \\ \hline 1 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 0 \end{array}$
1	0	0	0
$\begin{array}{c} 1 \\ \\ \hline 1 \end{array}$	$\begin{array}{c} 0 \\ \\ \hline 1 \end{array}$		
1	0		

Au final, cela permet de construire toute la grille, dans tous les cas de figure possibles. Le résultat est visible sur la grille de droite ci-dessous.

Notons que cette technique pourrait servir dans d'autres jeux, comme un Sudoku.

3 Modes de jeu

Après un écran d'introduction et un menu sommaire, le programme propose 3 modes de jeu, dans lesquels on peut choisir le niveau de l'ordinateur (et si on veut tricher), ainsi que le test statistique des algorithmes :

- 1) Jeu en solo sur une grille aléatoire.
- 2) Résolution d'une grille aléatoire par l'ordinateur, avec mise en évidence des bateaux qu'il doit trouver (voir annexe E page 81).
- 3) Jeu contre l'ordinateur (suivant l'algorithme de déroulement de partie en annexe A, page 42).

L'affichage sera alors le suivant :

Toto											
	A	B	C	D	E	F	G	H	I	J	
0								○	×	×	
1									○	○	
2	○										
3											
4							○				
5		○						○			
6			×								
7											
8			○								
9						○				○	

<Toto> Coup (Entrée pour un coup aléatoire) :

HAL											
	A	B	C	D	E	F	G	H	I	J	
0											
1											
2							○				
3				○				○			
4					○				×		
5						○					
6			○	○	○	○	○				
7		○	×	×	×	○					
8			○	○	○						
9											

Notons que chaque fois qu'un coup est demandé au joueur, le programme teste la validité de sa réponse (chaîne qui n'est pas une case, case hors grille ou encore une case déjà jouée).

- 4) Tests statistiques des algorithmes de résolution avec choix des paramètres de la grille.

INTERFACE WEB

Le dossier `interface_web` contient les fichiers pour une interface web. Celle-ci est réalisée en HTML5 et CCS3 et utilise des canvas dont la gestion est faite en Javascript. La communication avec les modules du projet se fait via le script cgi `bn_cgi.py`. Un serveur minimal est également implémenté dans le fichier `serveur.py`.

Un problème rencontré a été la persistance des données de jeu entre les lancements successifs du scripts (les différentes pages). Pour résoudre ce problème, au premier lancement d'une partie, le script crée un ID aléatoire et va sauvegarder les données de la partie dans le fichier `./sessions/session_ID` grâce au module built-in `shelve`. Cet ID est stocké ensuite dans le code source de la page HTML en tant que variable Javascript. Il ne restera alors plus qu'à recharger ces données lors des coups suivants.

1 Serveur web

La partie serveur est gérée par le fichier `serveur.py`. Celle-ci crée un serveur web, grâce au sous-module `HTTPServer` du module `http.server`, sur le port 8000 accessible uniquement en local (`localhost` ou `127.0.0.1`). La gestion des scripts cgi est faite par le sous module `CGIHTTPRequestHandler` du module `http.server`.

Le deuxième rôle de `serveur.py` est d'effacer les anciens fichiers de session au cas où il en resterait.

2 HTML et CSS

Le fichier `index.html` présente rapidement le projet et permet de lancer les trois modes de jeu : jeu solo, résolution automatique seule et partie contre l'ordinateur. Son style CSS est dans le fichier `./css/style_index.css`.

Lors du lancement d'une partie, le code de la page est généré par le script `bn_cgi.py` et le style CSS de la page est dans les fichiers `./css/style_solo.css` ou `./css/style_duos.css`, suivant s'il faut afficher une ou deux grilles. Ces deux styles héritent de `./css/style_base.css` qui contient leurs éléments communs.

Le dessin de la grille est effectué par un canvas en HTML5. Cet objet permet de dessiner grâce à des fonctions en Javascript. La récupération des coordonnées de la souris est également gérée en Javascript.

3 Script cgi

Un script cgi permet de générer du code source HTML pour l'envoyer au serveur en fonction des données qu'il reçoit. Ici, ces données sont :

- id : l'identificateur de la partie, égal à 0 au premier lancement.
- mode : le mode de jeu (0 : partie solo, 1 : partie contre l'ordinateur, 2 : résolution automatique).
- jx et jy : les coordonnées de la case cliquée par le joueur.

Une URL sera donc de la forme :

`http://localhost:8000/cgi-bin/bn_cgi.py?id=638229159&mode=1&jx=2&jy=6`

Le script va alors récupérer ces paramètres et lancer une action de jeu appropriée :

- Si l'id vaut 0, on vient de lancer le script pour la première fois. On crée donc un id aléatoire et on crée les acteurs du jeu (joueurs et grilles) que l'on sauvegarde dans un fichier de session.

Sinon, on récupère les données du jeu dans le fichier de session correspondant.

Cet id est sauvegardé dans le code HTML de la page dans une variable Javascript `sessionID`.

- mode va déterminer l'action à entreprendre : faire jouer l'ordinateur ou le joueur, et également déterminer la feuille de style.

Une fois ces paramètres déterminés, et les objets de jeu créés, le script va créer le source HTML et les fonctions Javascript d'affichage du canvas. Grâce à la variable `sessionID`, on pourra faire des liens dynamiques vers la page suivante de la partie.

4 Évolutions possibles

Grâce à cette infrastructure, on peut tout à fait envisager de créer une base de données des utilisateurs connectés et organiser des parties à deux joueurs, garder des traces de leurs statistiques (nombre de partie jouées, gagnées, nombre de coups moyens,...).

Par ailleurs, cette partie étant juste expérimentale, je n'ai pas implémenté la possibilité pour un joueur de placer lui même ses bateaux, mais ce ne serait pas difficile à faire (juste un petit peu long).

POINT DE VUE PÉDAGOGIQUE

Bien évidemment, ce projet dépasse largement ce qui est exigible d'un élève (même très bon) de lycée. Certains points peuvent néanmoins être abordés en simplifiant certaines parties et en l'abordant soit comme une série de TP guidés (les élèves doivent coder le contenu des fonctions dont on leur donne le prototype), soit comme projet de fin d'année. De part sa richesse, on peut également aborder ce projet comme trame pour introduire différentes notions de programmation.

On pourra aborder les points suivants :

- La structure de la grille : un bon exemple de codage d'une structure complexe (définir les états des cases, utilisation d'un dictionnaire ou d'une liste double, test des cases valides,...).
- Le placement de bateaux : sûrement la partie la moins évidente, mais oblige à réfléchir sur la façon de définir un bateau.
- L'affichage (simple) de la grille en console : utilisation de boucles imbriquées et de tests pour afficher les bons symboles, et gestion de la mise en page.
- Une interface graphique en `tkinter` en utilisant des boutons ou un canevas pour les cases. Pour les plus en avance, on peut également créer une interface web en HTML5 utilisant un script `cgi`.
- La possibilité pour un joueur de tirer sur une case et retour du résultat.
- Une résolution de la grille par l'ordinateur avec uniquement des tirs aléatoires sur les cases vides (les plus en avance pourront réfléchir à des méthodes plus évoluées, comme la gestion de la file d'attente).
- La détection des bateaux coulés par le joueur.
- La création d'un menu pour le choix de jeu (solo, contre l'ordinateur,...).

À côté de ça de nombreux points plus techniques peuvent être abordés comme :

- Mise en place de modules et utilisation d'un dépôt de code (GIT).
- Introduction à la programmation orientée objet (POO).
- Gestion de la validité des données entrées par l'utilisateur en mode console, gestion des exceptions.
- Gestion de l'encodage Unicode.
- Lancement de commandes systèmes (pour effacer le terminal).
- Mise en place d'outils d'analyse statistique (indicateurs et graphiques) pour la résolution automatique.

CONCLUSION

Ce travail a été très stimulant et m'a pris de nombreuses heures (environ 200), mais j'y ai pris beaucoup de plaisir. Il a d'abord fallu mettre en place des structures de données pour avoir un projet minimal. Ensuite est venu le temps de coder l'algorithme de résolution et son optimisation, les outils statistiques, la construction de l'interface console et enfin l'interface web. Lors de ces différentes phases, de nombreux problèmes sont survenus et leurs résolutions m'ont permis de progresser.

Certes il reste des points à développer, comme la mise en place d'une architecture de jeu en réseau, ou une interface en `tkinter` qui devait être réalisée par mon binôme, mais je pense que ce projet est déjà bien abouti.

La rédaction du rapport en \LaTeX a également été très plaisante, avec quelques figures réalisées avec `tikz`, l'affichage des caractères unicodes, ou encore la rapatriement automatique des docstrings.

ANNEXES

ÉTUDE DES ALGORITHMES CLÉS

Dans cette partie, une description commentée des algorithmes clés du projet. Les algorithmes vraiment simples (comme par exemple tirer sur une case) ne sont pas mis.

Le fait de rédiger cette partie a permis de prendre du recul sur ces algorithmes et de faire quelques améliorations (formaliser les choses est toujours bénéfique).

1 Gestion de la grille

1.1 Détermination des espaces vides

`Grille.get_max_space(self, case, direction, sens):`

```

Si direction == TOUTES_DIRECTIONS :
    On renvoie le maximum de get_max_space(case, HORIZONTAL, sens=1)
    et get_max_space(case, VERTICAL, sens=1)
direction[0] → dh
direction[1] → dv
1 → m
case[0] → x
case[1] → y
Tant que la case (x+dh, y+dv) est vide et est dans la grille :
    m+1 → m
    x+dh → x
    y+dv → y
Si sens == 1 (on regarde dans l'autre sens):
    case[0] → x
    case[1] → y
    Tant que la case (x-dh, y-dv) est vide et est dans la grille :
        m+1 → m
        x-dh → x
        y-dv → y
Retourner m

```

Cet algorithme est très simple. Il se contente de compter dans chaque direction le nombre de cases vides et, éventuellement dans les deux sens.

1.2 Détermination des bateaux possibles démarrant sur chaque case et des placements possibles de chaque bateaux

Grille.get_possibles(self):

```

On met à jour la liste des cases vides
# Récupération des bateaux possibles sur chaque case
possibles_case est un dictionnaire indexé par les cases
Les éléments de possibles_case sont des listes vides
On récupère la liste unique des bateaux restants dans tmp_taille_bateaux
Pour chaque case vide :
    Pour chaque direction dans [DROITE, BAS] :
        tmax est l'espace maximum dans direction
        On ajoute à possible_case[case] les tuples (taille, direction)
        pour les tailles dans tmp_taille_bateaux, si taille <= tmax

# Récupération des position possibles pour chaque bateau
possibles est un dictionnaire indexé par les tailles de bateaux
Les éléments de possibles sont des listes vides
Pour chaque case index dans possible_case :
    Pour chaque couple (taille, direction) dans possible_case[case] :
        On ajoute (taille, direction) à possibles[taille]

```

Cet algorithme est un des points clés du projet. Dans un premier temps on récupère la liste de tous les bateaux possibles démarrant sur chaque case, ainsi que leurs directions possibles. Dans un deuxième temps on récupère, pour chaque bateau, ses placements possibles.

Les dictionnaires créés ont cette allure :

- possibles_case={ (0,0):[(5,(1,0)), (5,(0,1)),...], (0,1):... }
- possibles={5:[((0,0), (1,0)), ((0,0), (0,1)), ((1,0), (1,0)),...], 4:... }

Afin d'éviter de répéter les mêmes calculs, la liste taille_bateaux des bateaux restants sur la grille est transformée en une liste tmp_taille_bateaux qui ne contient que ses éléments uniques. Pour cela on la convertit en ensemble puis on revient à une liste avec l'instruction

```
tmp_taille_bateaux = list(set(taille_bateaux))
```

À des fins de tests, cette fonction accepte un argument tri qui, s'il est Vrai trie les résultats. Ce tri est uniquement nécessaire pour les tests et est mis à Faux par défaut (cela prend un petit peu plus de temps de trier à chaque fois).

1.3 Création de bateaux aléatoires

1.3.1 Ajout d'un bateau aléatoire

```
Grille.add_bateau_alea(self, taille):
```

```
Récupérer les placements de départ possibles des bateaux
  (dans le dictionnaire possibles)
Si possibles[taille] est vide :
  Retourner Faux
Sinon :
  Choisir un tuple (case, direction) au hasard dans possibles[taille]
  Ajouter la bateau (taille, case, direction)
  Retourner Vrai
```

Si possibles[taille] est vide, cela signifie qu'on ne peut pas placer ce bateau et donc on renvoie Faux pour éviter une situation de blocage par la suite, sinon on renvoie Vrai pour indiquer que tout s'est bien passé.

1.3.2 Création d'une flotte aléatoire

```
Grille.init_alea(self):
```

```
0 → nb_bateaux
Tant que nb_bateaux < nombre de bateaux à placer :
  0 → nb_bateaux
  On crée une copie temporaire de la grille dans gtmp
  Pour chaque taille de bateau à placer :
    On essaie de placer un bateau aléatoire de cette taille dans gtmp
    Si pas possible :
      On casse la boucle et on recommence tout
      (pour éviter une situation de blocage)
    Sinon :
      nb_bateaux+1 → nb_bateaux
      On enlève taille de la liste des bateaux restants à placer
  Enfin on copie l'état de gtmp dans notre grille
```

Pour placer chaque bateau, on utilise la fonction précédente avec sa valeur de sortie (vrai ou faux) qui permet de savoir si le placement a été possible.

2 Détermination des bateaux coulés

Joueur.check_coules(self):
 xmax et ymax sont les dimensions de la grille et dimensions=(xmax, ymax).

```

coules est une variable globale contenant
  la liste de case marquées comme coulées
checked est une liste vide
Pour chaque case dans les cases jouées triées en ordre lexicographique :
  liste_touchees est une liste vide
  Si case a été marquée comme touchée :
    Si case est dans checked ou dans coules :
      On continue la boucle (on ignore cette case)
    Vrai→case_isolee
    Pour d dans [DROITE, BAS] :
      Si la case (case[0]+d[0],case[1]+d[1]) est hors de la grille :
        On continue la boucle
      Si la case (case[0]+d[0],case[1]+d[1]) est marquée touchée :
        d→direction
        Faux→case_isolee
        On casse la boucle
    Si case_isolee est Vraie :
      On continue la boucle (on ignore cette case)
    0→k
    Tant que case[0]+k*direction[0] < xmax
      et case[1]+k*direction[1] < ymax
      et (case[0]+k*direction[0],case[1]+k*direction[1])
      est marquée touchée :
        On ajoute la case (case[0]+k*direction[0],case[1]+k*direction[1])
        aux listes checked et liste_touchees
        k += 1
    Si (
      (case[direction[1]]== 0
        ou (case[0]-direction[0],case[1]-direction[1]) est manquée)
      ou (case[direction[1]]+k== dimensions[direction[1]]
        ou (case[0]+k*direction[0],case[1]+k*direction[1]) est manquée)
      ) ou len(liste_touchees)==taille_max :
      Enlever le bateau de taille len(liste_touchees)
      des bateaux restants
      Éliminer les cases adjacentes à celles de liste_touchees
      Ajouter les cases de liste_touchees à la liste coules

```


On parcourt les cases jouées de haut en bas et de gauche à droite. Quand on tombe sur une case touchée, on vérifie d'abord qu'on n'a pas déjà traité cette case, puis on détermine la direction d'un bateau éventuel (si pas de direction c'est que cette case est isolée et on l'ignore).

Ensuite on détermine le nombre de cases adjacentes touchées (grâce à la direction)

Enfin on décide que le bateau est coulé si ses extrémités sont sur les bords de la grille ou ont été marquées comme manquées, ou si sa taille est celle de plus grand bateau restant à trouver. On marque alors ses cases adjacentes comme impossibles et on l'enlève de la liste des bateaux à chercher.

Les listes coules et checked permettent de garder une trace des cases déjà traitées. La liste coules est global (un attribut de la classe Joueur). Elle garde la trace des cases déjà marquées comme appartenant à des bateaux coulés. La liste checked est locale et sert pour le parcours des cases jouées. Sans ces listes, la deuxième case touchée d'un bateau serait de nouveau traitée ce qui créerait une incohérence.

Cet algorithme n'est utilisé que si le joueur est un joueur humain. En effet, l'algorithme de résolution automatique gère une file d'attente et possède sa propre méthode pour déterminer qu'un bateau est coulé.

3 Algorithme de résolution

3.1 Optimisation de la phase de tirs en aveugle

3.1.1 Détermination des probabilités par échantillonnage

`Grille.case_max_echantillons(self, nb_echantillons):`

```

probas est un dictionnaire indexé sur les cases
Pour chaque case, 0→probas[case]
On répète nb_echantillons fois :
    grille_tmp reçoit une copie temporaire de la grille du suivi
    On crée une flotte aléatoire sur grille_tmp
    Pour chaque case vide dans la grille de suivi originale :
        Si la case contient un bateau dans grille_tmp :
            probas[case]+1→probas[case]
    Pour chaque case, probas[case]/nb_echantillons→probas[case]
    case_max est la case qui a la plus grande probabilité pmax
    On retourne (case_max, pmax)

```

C'est l'algorithme de niveau 4 qui crée une estimation de la distribution de probabilité avec des échantillons de flottes aléatoires. Aucune difficulté particulière n'est à signaler.

3.1.2 Détermination du nombre de bateaux possibles sur chaque case

Grille.case_max(self) :

```

probas est un dictionnaire indexé sur les cases
Pour chaque case, 0→probas[case]
On récupère la liste placements possibles pour chaque bateau
    (dans le dictionnaire possibles)
Pour chaque taille de bateau restant :
    Pour chaque (case, direction) dans possibles[taille] :
        #La probabilité de chaque case occupée par le bateau est augmentée de 1
        Pour k allant de 0 à taille :
            probas[case+k*direction] est augmentée de 1
case_max est la case qui a le plus de possibilités pmax
On retourne (case_max, pmax)

```

C'est l'algorithme de niveau 5 qui détermine, pour chaque case, le nombre de bateaux possibles.

Dans la mesure où la liste possibles[taille] ne donne que les points de départ de chaque bateau, il est nécessaire de parcourir toutes ses cases occupées dans la dernière boucle. case+k*direction signifie (case[0]+k*direction[0], case[1]+k*direction[1]).

3.1.3 Énumération de tous les cas

Grille.case_max_all(self):

```
gtmp est une copie temporaire de la grille
probas_all est un dictionnaire indexé par les cases
Pour chaque case, 0→probas_all[case]
Créer toutes les répartitions de bateaux sur gtmp
Récupérer la case qui en contient le plus
```

Grille.make_all(self, gtmp):

```
# En entrée : une grille gtmp
S'il n'y a plus de bateaux dans la liste de gtmp :
    Mettre à jour les probabilités avec les cases occupées sur gtmp
    Sortir de la fonction
Récupérer les positions de bateaux possibles sur gtmp
taille est la taille du premier bateau restant
Pour (case, direction) dans possibles[taille] :
    gtmp2 est une copie temporaire de gtmp
    Ajouter le bateau (taille, case, direction) à gtmp2
    Enlever ce bateau de la liste de ceux de gtmp2
    Créer toutes les répartitions de bateaux sur gtmp2
```

Cet algorithme est récursif. On utilise une grille temporaire gtmp sur laquelle on va faire tous les arrangements possibles. Pour chaque placement possible d'un bateau, on crée une autre grille temporaire gtmp2 sur laquelle on va recommencer à faire tous les arrangements possibles avec les bateaux restants.

Le problème de cet algorithme est qu'il est exponentiel. Avec une grille vide de 10×10 , on obtient les résultats suivants :

- taille_bateaux=[5,4] : 11 744 répartitions, 0,8 secondes
- taille_bateaux=[5,4,3] : 1 064 728 répartitions, 70 secondes
- taille_bateaux=[5,4,3,2] : 101 286 480 répartitions, 6566 secondes

À chaque fois qu'on ajoute un bateau, le nombre de répartitions possibles, ainsi que le temps de calcul pour les déterminer, est multiplié par 100.

Pour une liste de bateaux taille_bateaux=[5,4,3,3,2] on s'attend donc à 10^{10} répartitions et un temps de calcul d'environ 8 jours (pour un seul coup!).

Certes, sur une grille vide, on pourrait jouer avec les symétries (et diviser le calcul par 8) mais celles-ci sont tout de suite brisées dès qu'une case a été jouée.

Même si cette méthode est inutilisable en tant que telle, elle peut être intéressante sur des petites grilles, ou avec peu de bateaux, et donc peut être utilisée en fin de partie. C'est le rôle du niveau 6 dans lequel on utilise cette méthode dès que le nombre de cases vides descend en-dessous d'un certain seuil.

Pour information voici, sur la page suivante, le nombre de répartitions sur chaque case pour taille_bateaux=[5,4,3,2] :

	A	B	C	D	E	F	G	H	I	J
0	8138842	11060807	13598504	15286936	16111642	16111642	15286936	13598504	11060807	8138842
1	11060807	12059986	13448547	14238869	14511549	14511549	14238869	13448547	12059986	11060807
2	13598504	13448547	14505458	15088255	15244715	15244715	15088255	14505458	13448547	13598504
3	15286936	14238869	15088255	15542880	15653410	15653410	15542880	15088255	14238869	15286936
4	16111642	14511549	15244715	15653410	15769046	15769046	15653410	15244715	14511549	16111642
5	16111642	14511549	15244715	15653410	15769046	15769046	15653410	15244715	14511549	16111642
6	15286936	14238869	15088255	15542880	15653410	15653410	15542880	15088255	14238869	15286936
7	13598504	13448547	14505458	15088255	15244715	15244715	15088255	14505458	13448547	13598504
8	11060807	12059986	13448547	14238869	14511549	14511549	14238869	13448547	12059986	11060807
9	8138842	11060807	13598504	15286936	16111642	16111642	15286936	13598504	11060807	8138842

*Nombre de bateaux sur chaque case pour une grille vide avec $taille_bateaux=[5, 4, 3, 2]$
à partir d'une liste exhaustive de toutes les répartitions possibles*

3.2 Gestion de la file d'attente

3.2.1 Création de la file d'attente

```
Ordi.add_adjacentes_premiere(self):
```

```

adjacent est la liste des cases vides adjacentes à case_touchee
taille_min est la taille minimale des bateaux restants
Pour direction dans [HORIZONTAL, VERTICAL] :
    direction[0] → k
    Si l'espace maximum dans direction sur case touché >= taille_min :
        Pour case dans adjacent :
            Si case[k]==case_touchee[k] :
                Ajouter case à la file d'attente
            Sinon afficher que cette direction ne convient pas

```

Pas de grande difficulté sur celui-ci. On se contente de regarder si le plus petit bateau rentre dans chaque direction et on ajoute à la file d'attente les cases correspondantes.

Les tests des deux directions sont regroupés dans une boucle afin de rendre le code plus compact.

3.2.2 Optimisation de la file d'attente

Grille.case_max_touchee(self, case_touchee):
 Pour des raisons de mise en page, notons ct la case touchée.

```

On marque temporairement case_touchee comme vide
On récupère la liste placements possibles pour chaque bateau
  (dans le dictionnaire possibles)
probas est un dictionnaire indexé sur les cases
Pour chaque case, 0→probas[case]
Pour chaque taille de bateau restant :
  Pour chaque direction dans [HORIZONTAL, VERTICAL] :
    #Bateau qui se termine sur case_touchee
    Si ct[direction[1]]-(taille-1)*direction[direction[1]]>=0
      et ((ct[0]-(taille-1)*direction[0],
          ct[1]-(taille-1)*direction[1]), direction)
      est dans possibles :
      probas[(ct[0]-direction[0],ct[1]-direction[1])] += 1
    #Bateau à cheval sur case_touchee
    Pour k allant de 1 à taille-2 :
      Si ct[direction[1]]-k*direction[direction[1]]>=0
        et ((ct[0]-k*direction[0], ct[1]-k*direction[1]),
            direction) est dans possibles :
        probas[(ct[0]-direction[0],ct[1]-direction[1])] += 1
        probas[(ct[0]+direction[0],ct[1]+direction[1])] += 1
    #Bateau qui démarre sur case_touchee
    Si ((ct[0], ct[1]), direction) est dans possibles :
      probas[(ct[0]+direction[0],ct[1]+direction[1])] += 1
On remet l'état de case_touchee à touché
On trie probas dans l'ordre décroissant du nombre de possibilités
On renvoie cette liste

```

Le fait de boucler sur les directions HORIZONTAL et VERTICAL, c'est-à-dire dans [(1, 0), (0, 1)], permet de condenser les calculs sur ces deux directions dans une seule boucle. ct[direction[1]] et direction[direction[1]] permettent d'obtenir respectivement ct[0] et direction[0] dans le cas horizontal, et ct[1] et direction[1] dans le cas vertical.

Notons que dans cet algorithme on ne s'intéresse qu'aux cases adjacentes à case_touchee.

Le tri final de la liste se fait de la façon suivante : on commence par convertir le dictionnaire probas en une liste de tuple (case, probas[case]) dans probas_liste, puis on l'ordonne avec l'instruction

```
sorted(probas_liste, key=lambda proba: proba[1], reverse = True)
```

3.2.3 Mise à jour après le deuxième coup touché

```
Ordi.update_queue_touche(self):
```

```
Si case_courante[1] == case_touchee[1] :
    direction=HORIZONTAL
Sinon :
    direction=VERTICAL
Si on vient de découvrir la direction du bateau (len(liste_touchee)==1):
    On affiche cette direction
    On enlève les cases
        (case_touchee[0]-direction[1], case_touchee[1]-direction[0])
        et (case_touchee[0]+direction[1], case_touchee[1]+direction[0])
        de la file d'attente
nv_case est la case
    (case_courante[0] + direction[0]*signe(case_courante[0]-case_touchee[0]),
     case_courante[1] + direction[1]*signe(case_courante[1]-case_touchee[1]))
Si nv_case est dans la grille et est vide :
    On ajoute nv_case à la file d'attente
On ajoute case_courante à liste_touches
```

La liste `liste_touches` permet de savoir combien de cases ont été touchées sur ce bateau. Si c'est la deuxième, on met à jour les file d'attente avec les case dans la bonne direction.

`signe(x)` renvoie 1 si $x > 0$ et -1 si $x < 0$.

`signe(case_courante[i]-case_touchee[i])`, pour $i \in \{0,1\}$, permet de déterminer le sens dans lequel on vient de toucher la nouvelle case, ce qui permet d'ajouter la case en bout de configuration à la file d'attente (si elle est valide et vide).

3.2.4 Mise à jour après le deuxième coup manqué

Ordi.update_queue_manque(self) :

```

taille_min est la plus petite taille de bateau restant
delta =
    (case_courante[0]-case_touchee[0],
     case_courante[1]-case_touchee[1])
direction =
    (abs(case_courante[0]-case_touchee[0]),
     abs(case_courante[1]-case_touchee[1]))
case_face =
    (case_touchee[0]-delta[0],
     case_touchee[1]-delta[1])
Si l'espace vide sur case_face dans direction < taille_min-1 :
    Enlever case_face de la file d'attente

```

- delta est l'écart entre la première case touchée du bateau et la case sur laquelle on vient de jouer
- direction est la direction dans laquelle on vient de jouer
- case_face est la case en face de celle qu'on vient de jouer

Dans la mesure où il y a déjà eu une case touchée sur ce bateau, on teste s'il y a assez de place sur case_face pour faire rentrer un bateau de taille `taille_min-1` dans la direction `direction`.

3.3 Algorithme complet de résolution

Voici l'algorithme complet de résolution de la grille par l'ordinateur :

```

La file d'attente est une liste vide
liste_touches est une liste vide
Tant que le grille n'est pas résolue :
  Si la file d'attente est vide (tir en aveugle) :
    Si liste_touches n'est pas vide :
      On enlève le bateau de taille len(liste_touches)
      On élimine les cases adjacentes à celles de liste_touches
      On vide liste_touches
    On élimine les zones trop petites
    case_courante reçoit une case en aveugle (suivant le niveau)
  Sinon (tir ciblé) :
    case_courante reçoit le premier élément de la file d'attente
    On enlève cette case de la file d'attente
  On tire sur case_courante
  Si on a touché :
    Si liste_touches est vide (1ère case du bateau) :
      On ajoute case_courante dans liste_touches
      case_courante → case_touchee
      On ajoute ses cases adjacentes dans la file d'attente
      (en testant également les directions impossibles éventuelles)
    Sinon :
      Si len(liste_touches) == 1 (2ème case du bateau):
        On détecte la direction du bateau
        On met à jour la file d'attente
        (avec la case adjacente à case_courante dans la bonne direction
         si elle est vide)
      Si le bateau touché est le plus grand restant :
        On vide la file d'attente
    Sinon :
      Si len(liste_touches) == 1 :
        On met à jour la file d'attente
        (on élimine éventuellement la case en face de case_touchee)
  On affiche le nombre de coups

```

4 Déroulement de la partie

Une partie à deux joueurs se déroule selon l'algorithme suivant (le joueur porte le numéro 0 et son adversaire le numéro 1) :

```
Placement des bateaux du joueur
Récupération des bateaux de l'adversaire
Aléa(0,1)→joueur_en_cours (celui qui commence)
Tant qu'aucun joueur n'a fini :
    Si joueur_en_cours == 0 et l'adversaire n'a pas fini :
        Le joueur joue un coup
        Mise à jour de l'affichage des grilles
        Affichage des messages du joueur (résultat du coup)
    Sinon :
        Récupération du coup de l'adversaire
        Mise à jour de l'affichage des grilles
        Affichage des messages de l'adversaire
    (joueur_en_cours+1)%2→joueur_en_cours (changement de joueur)
Affichage des grilles avec la solution
Affichage du gagnant
```

ÉTUDE STATISTIQUE DES ALGORITHMES DE RÉOLUTION

Le module `bn_stats.py` fournit, dans la classe `Stats`, les outils pour analyser statistiquement une distribution de valeurs (calculs des indicateurs statistiques classiques, représentation en histogramme et diagramme en boîte grâce aux modules `numpy` et `matplotlib`).

Cette classe fournit également les outils pour sauvegarder et charger la liste des résultats bruts dans un fichier texte pour des analyses plus poussées futures.

Chaque graphique fait apparaître, outre l'histogramme de distribution des fréquences, les indicateurs suivants :

- moyenne et écart-type,
- minimum, Q_1 , médiane, Q_3 et maximum dans le diagramme en boîte,
- le mode, noté à la base de la barre correspondante,
- le temps moyen de résolution.

La méthode de test est la suivante : on crée une liste `distrib` de longueur `xmax*ymax+1` qui est initialisée avec des valeurs nulles.

On répète n fois la résolution sur une grille aléatoire, chaque fois différente et, en notant k le nombre de coups de la résolution, on incrémente `distrib[k]` de 1.

Le calcul du temps se fait en lançant un chronomètre grâce à la fonction `time()` du module `time`, qui renvoie le nombre de seconde écoulées depuis `epoch` (le 1^{er} janvier 1970). Donc en sauvegardant cette valeur au début de la simulation dans une variable `start` et en calculant `time()-start` à la fin de la simulation, on obtient le temps total écoulé. Afin de chronométrer uniquement le temps de résolution (et non de la création de la grille), ce chronomètre est mis à jour uniquement lors de la résolution effective.

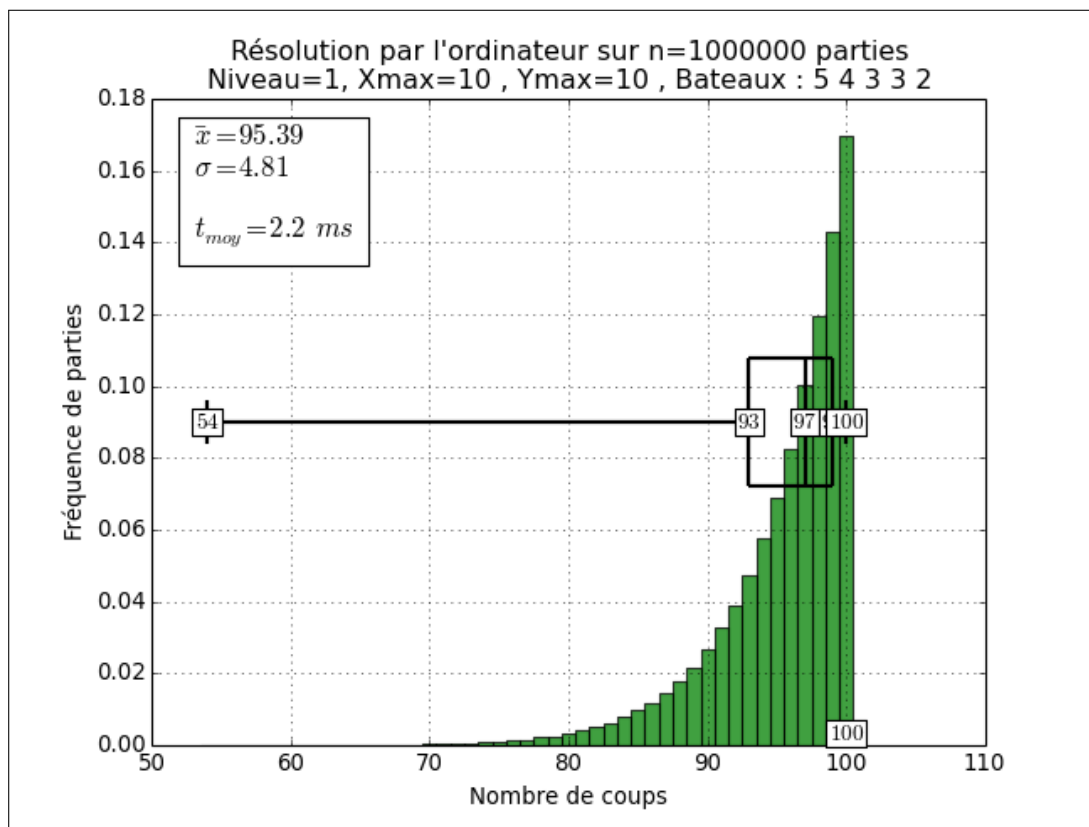
L'algorithme de test affiche une estimation du temps restant (ainsi que de la date et l'heure de la fin de la simulation) basé sur le temps de la première résolution (donc très approximative), ainsi que l'avancement par tranches de 10%.

Afin d'obtenir des résultats comparables, tous les temps ont été mesurés sur le même ordinateur disposant d'un processeur Intel i7-4800-MQ cadencé à 2,7 GHz en mode monoprocesseur, de 16 Go de mémoire vive et tournant sous un système Linux 64 bits (Xubuntu 14.04).

1 Niveau 1

Dans ce niveau tous les tirs sont aléatoires uniformément sur les cases vides, et il n'y a pas de phase de tirs ciblés.

On obtient les résultats suivants sur un échantillon de $n = 1\,000\,000$ parties :

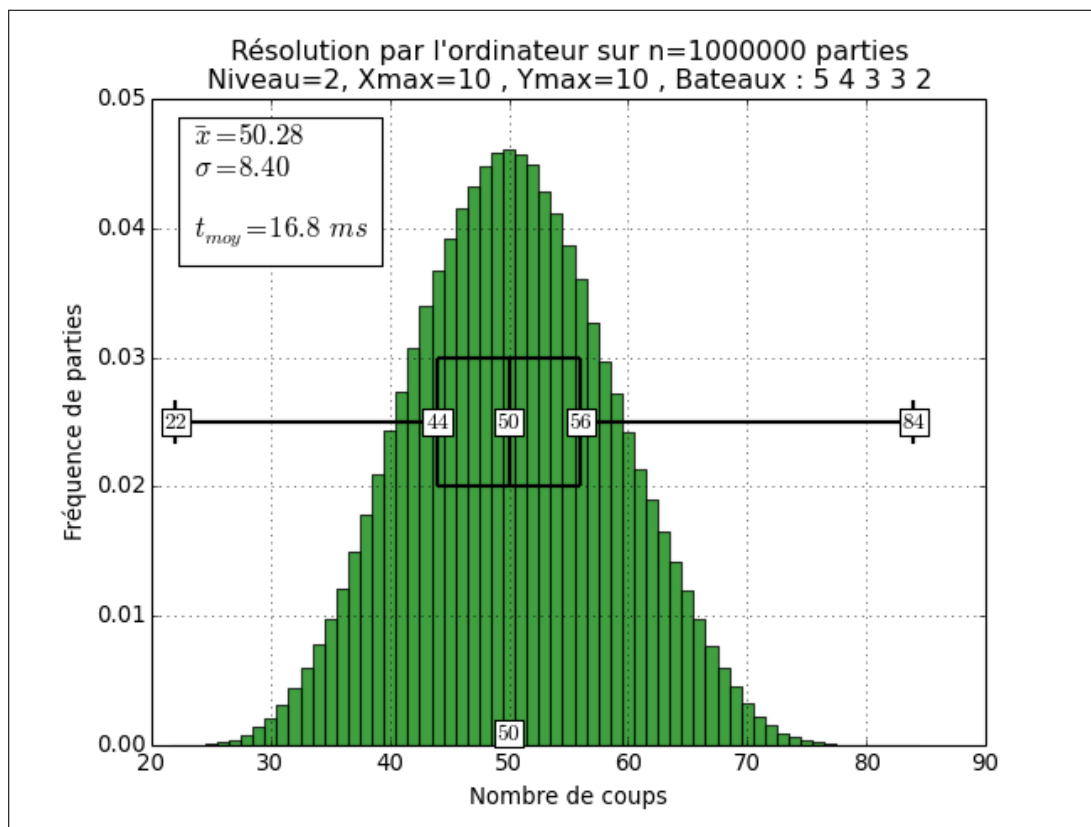


Comme on pouvait s'y attendre, les résultats sont catastrophiques. Par contre la résolution est quasi immédiate (2 ms par partie en moyenne)

2 Niveau 2

Au niveau 2, la phase de tirs en aveugle est aléatoire uniformément sur les cases vide, mais on ajoute la phase de tirs ciblés lorsqu'on touche un bateau.

Les résultats sur $n = 1\,000\,000$ parties sont les suivants :

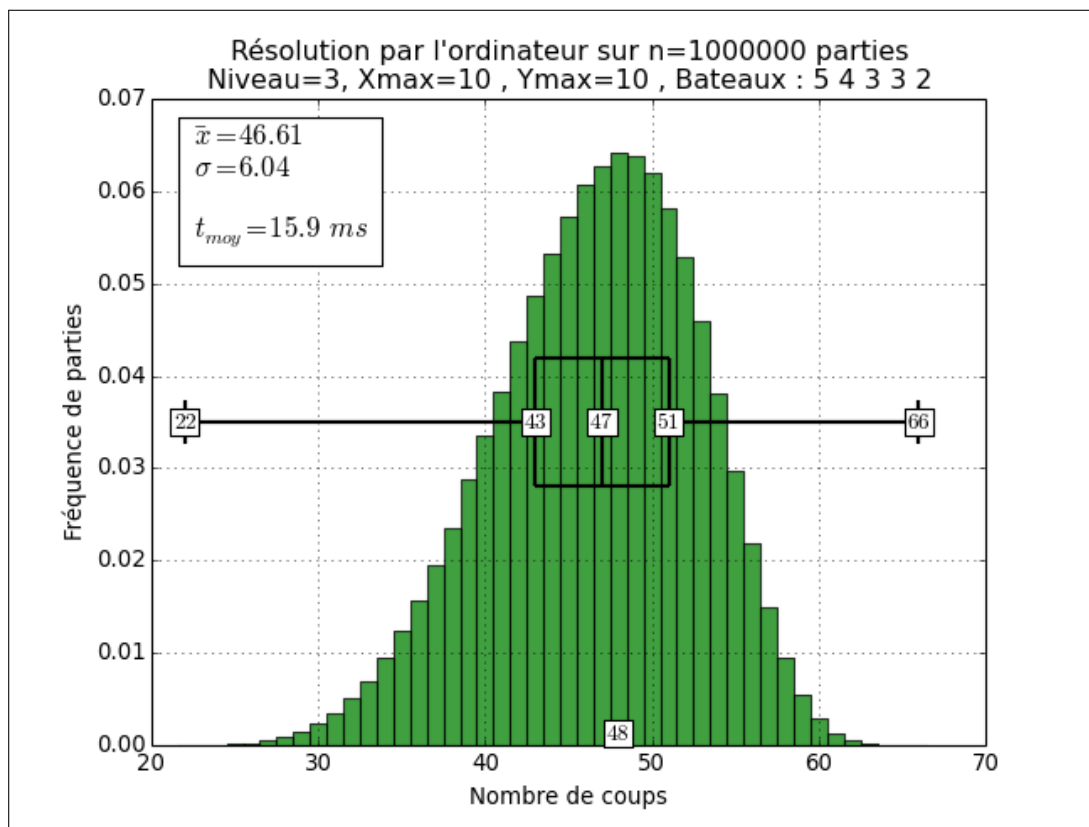


Le fait de gérer la phase de tirs ciblés change radicalement les résultats. La forme de la courbe de distribution est d'ailleurs totalement différente de la précédente. Par contre la résolutions prend 8 fois plus de temps.

3 Niveau 3

Au niveau 3, la phase de tirs en aveugle est encore aléatoire mais uniquement sur les cases noires.

Les résultats sur $n = 100\,000$ parties sont les suivants :



On note une amélioration significative des performances, autant pour la moyenne que pour le mode et la médiane, sans pour autant sacrifier en temps de résolution, au contraire puisqu'il faut moins de coups en aveugle pour finir la grille. Les résultats sont également plus homogènes ($\sigma = 6,04$ au lieu de $8,40$ précédemment).

4 Niveau 4

Au niveau 4 la détermination de la case optimale durant la phase de tirs en aveugle se fait en regardant, à chaque coup, un échantillon de taille `nb_echantillons` de distributions de bateau aléatoires.

La valeur de `nb_echantillons` va jouer sur les performances en nombre de coups (plus on fait d'échantillons et plus la distribution de probabilités obtenue est conforme à la vraie distribution de probabilité) mais aussi sur le temps de résolution. En effet ce dernier sera linéaire en `nb_echantillons`.

Vu le temps de calcul, les simulations suivantes portent sur un nombre plus faible de parties (la contrainte fixée est que le calcul ne doit pas durer plus d'une nuit).

4.1 Échantillons de taille `nb_echantillons=10`

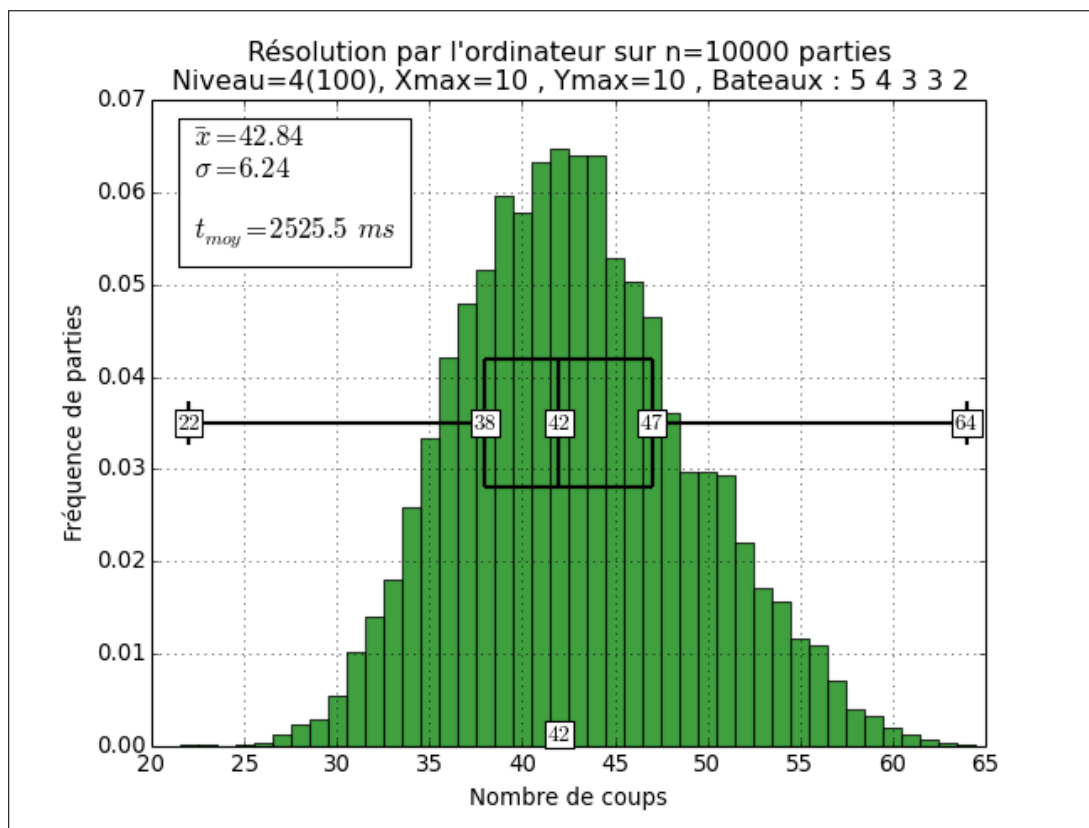
Avec des échantillons de taille 10 nous obtenons les résultats suivants sur $n = 100\,000$ parties :



Avec une taille modeste des échantillons, nous obtenons une nette amélioration des performances avec une moyenne de 44,29, pour un temps moyen de 0,3 secondes par partie.

4.2 Échantillons de taille nb_echantillons=100

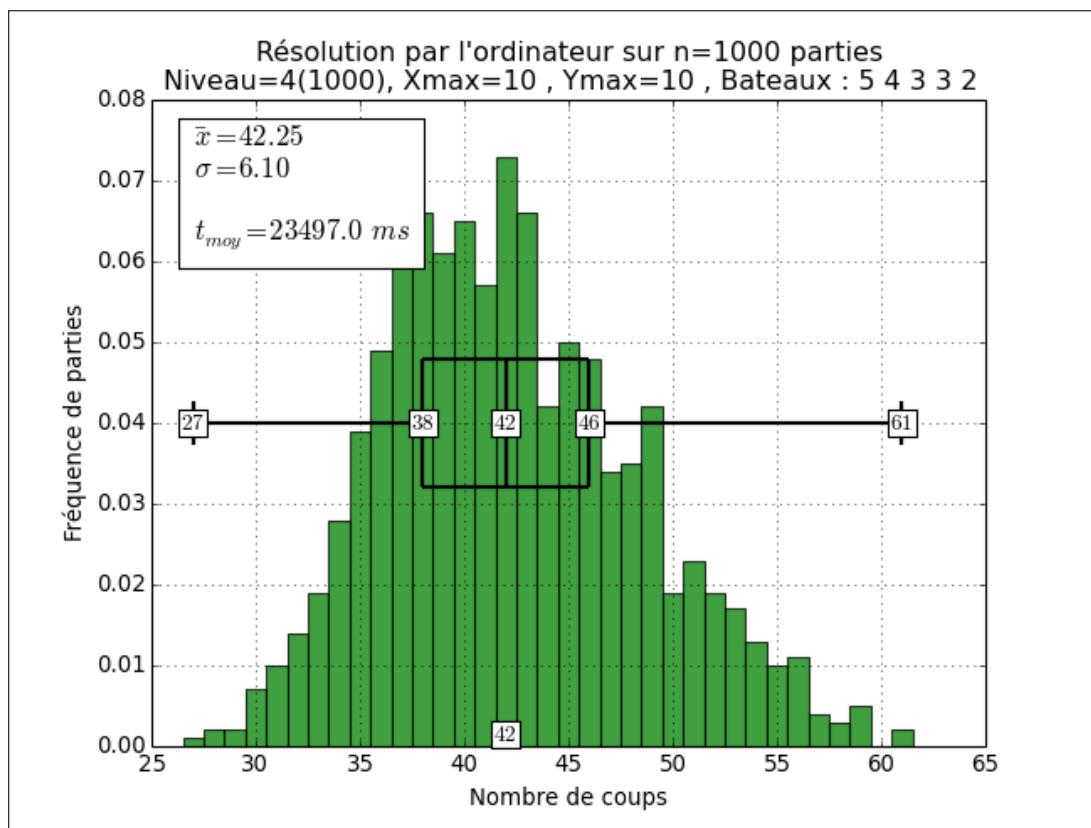
Avec des échantillons de taille 100 nous obtenons les résultats suivants sur $n = 10000$ parties :



Les résultats sont très bons, avec une moyenne de 42,84 coups par partie, mais le temps de calcul commence à devenir long (2,5 secondes par partie).

4.3 Échantillons de taille nb_echantillons=1 000

Nous prévoyons un temps de résolution moyen d'approximativement 30 secondes par partie. Aussi nous ne ferons qu'une simulation sur $n = 1\,000$ parties :

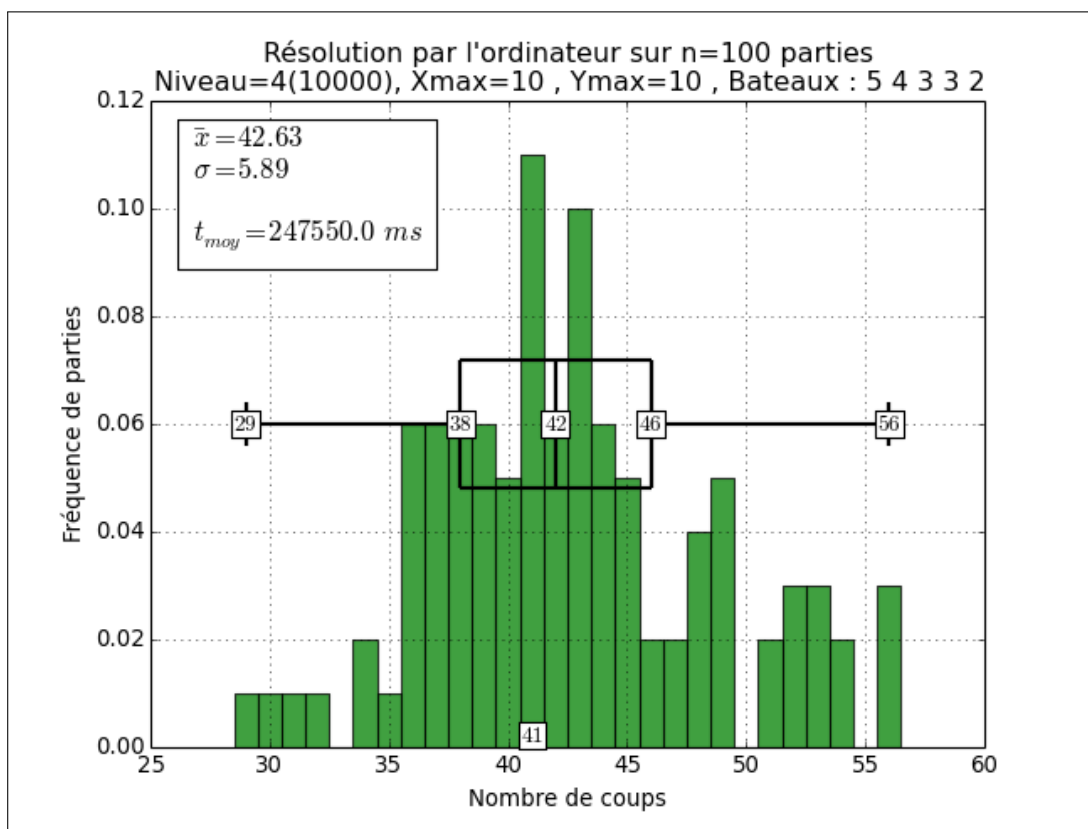


La forme de la distribution est moins régulière que les précédentes à cause du nombre plus réduit de parties simulées.

Le gain en nombre de coups moyen par partie, s'il est présent, est très limité compte tenu de l'augmentation linéaire du temps de calcul.

4.4 Échantillons de taille $\text{nb_echantillons}=1\,000$

Juste pour essayer, voici une simulation sur $n = 100$ partie :



Les résultats sont très décevants. On n'a fait que $n = 100$ parties ce qui explique peut-être cela mais on note même une régression ce qui est quand même étonnant.

4.5 Conclusion sur le niveau 4

Comme on pouvait s'y attendre le temps de résolution est approximativement linéaire en nb_échantillons.

Les résultats en nombre de coups sont très corrects mais le coût en temps de calcul est rédhibitoire. Par ailleurs, il a été impossible de créer de grosses simulations avec un nb_échantillons élevé ce qui biaise un peu les comparaisons.

Voici un tableau récapitulatif des résultats obtenus :

Numéro de la simulation	1	2	3	4
Taille des échantillons pour l'algorithme	10	100	1 000	10 000
Nombre de parties n	100 000	10 000	1 000	100
Nombre de coups moyens \bar{x}	44,29	42,84	42,25	42,63
Écart-type ¹ σ	6,62	6,24	6,10	5,89
Temps moyen par partie (en secondes)	0,29	2,5	23,5	247

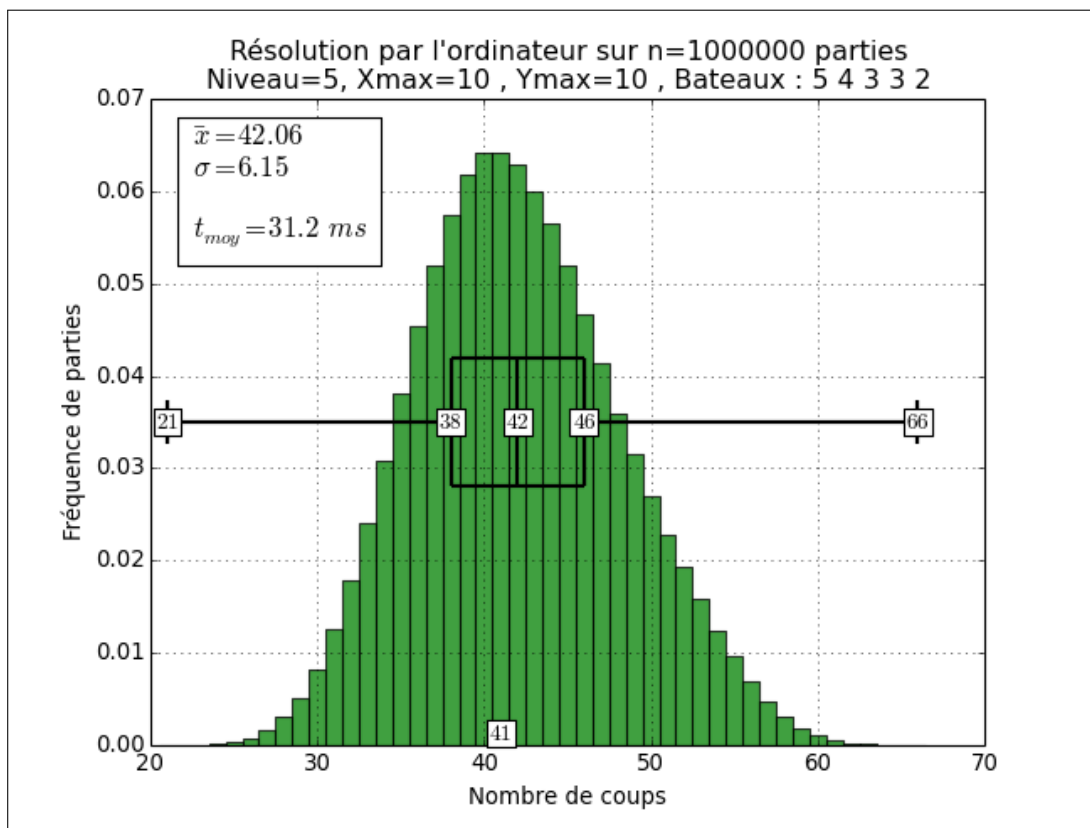
n°	nb_échantillons	Intervalle de confiance ² de μ
1	10	[44,25 ; 44,33]
2	100	[42,72 ; 42,96]
3	1 000	[41,87 ; 42,63]
4	10 000	[41,48 ; 43,78]

1. Vu la taille importante des échantillons nous prendrons $\sigma_{\text{estimé}} = \sigma_{\text{échantillon}}$

2. Au seuil de 95% : $\left[\bar{x} - 1,96 \frac{\sigma}{\sqrt{n}} ; \bar{x} + 1,96 \frac{\sigma}{\sqrt{n}} \right]$

5 Niveau 5

Les résultats obtenues sur un échantillon de $n = 1\,000\,000$ parties sont les suivants :



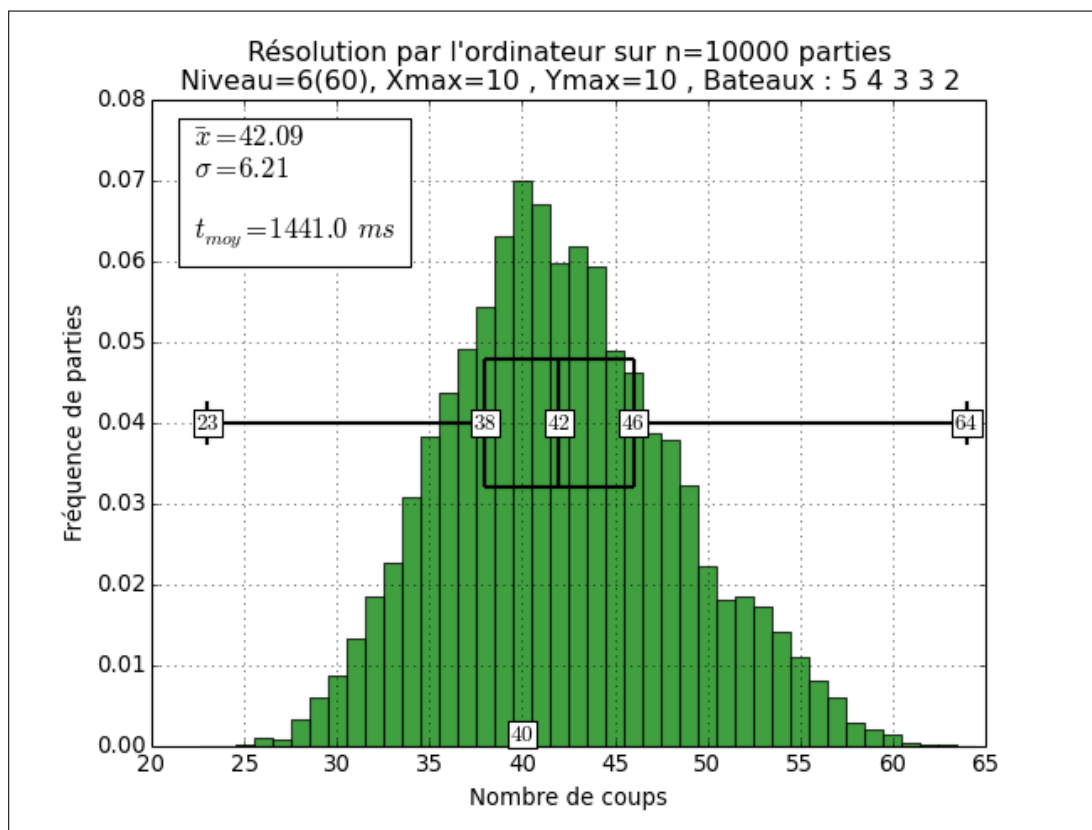
Les performances sont excellentes avec une moyenne de 42,06 coups pour un temps de résolution moyen de seulement 31,2 ms par partie.

6 Niveau 6

Au niveau 6, tant que le nombre de cases vides est inférieur à seuil, on procède comme au niveau 5 en regardant localement le nombre de bateaux possibles sur chaque case. Dès que le nombre de cases vides passe en-dessous de seuil, on crée la liste de tous les arrangements possibles de bateaux sur la grille (ce qui prend un temps conséquent, d'où l'obligation d'utiliser un seuil).

6.1 Seuil égal à 60

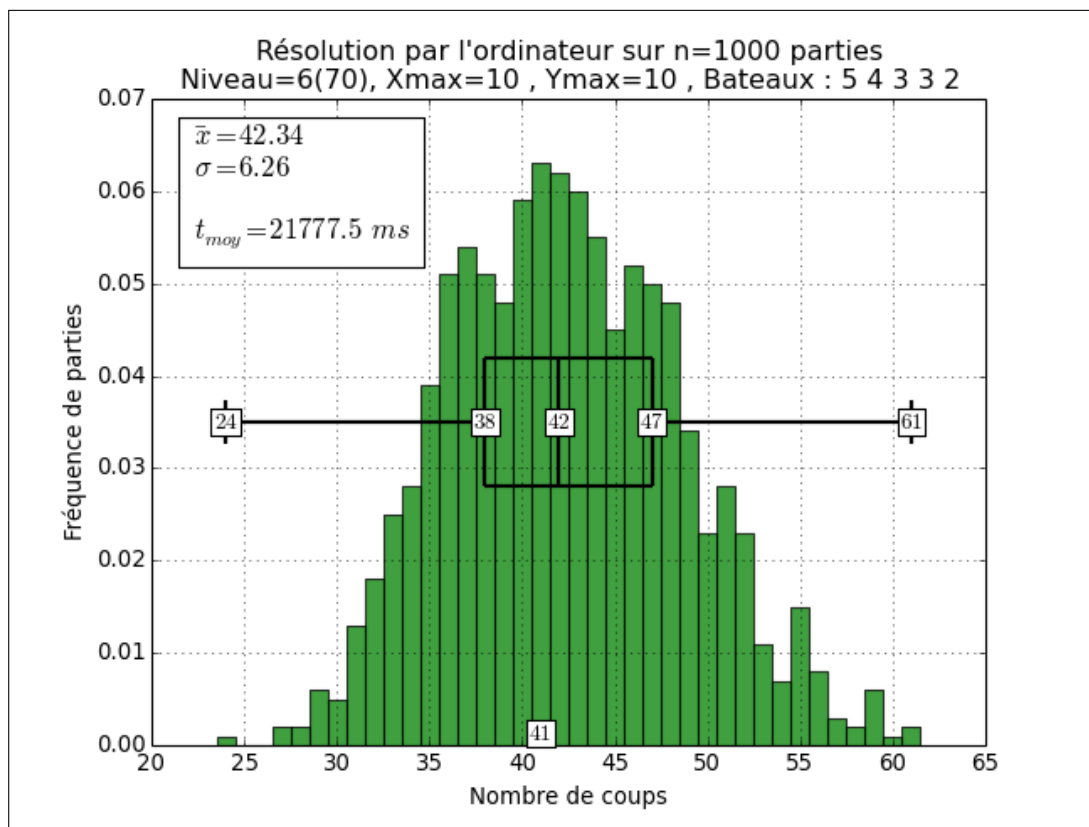
Avec un seuil de 60, les résultats obtenues sur un échantillon de $n = 10000$ parties sont les suivants :



Pour l'instant, avec un seuil de 60, nous restons sur notre faim. Il n'y a aucune amélioration par rapport au niveau 5.

6.2 Seuil égal à 70

Avec un seuil de 70, les résultats obtenues sur un échantillon de $n = 1\,000$ parties sont les suivants :



À notre grande surprise, la moyenne est un petit peu moins bonne alors qu'on aurait pu imaginer le contraire.

6.3 Conclusion sur le niveau 6

Comparons les intervalles de confiance à 95% de μ pour les trois dernières simulations :

n°	Niveau	n	\bar{x}	σ	Intervalle de confiance
1	5	1 000 000	42,06	6,15	[42,05 ; 42,07]
2	6(60)	10 000	42,09	6,21	[41,97 ; 42,21]
3	6(70)	1 000	42,34	6,26	[41,95 ; 42,73]

C'est un petit peu décevant. On aurait pu prévoir une amélioration du fait de regarder la grille de façon globale. Ce n'est malheureusement pas le cas. Le niveau 5 reste définitivement le meilleur.

7 Conclusions

Après avoir testé ces différents algorithmes nous pouvons tirer les conclusions suivantes :

- Le niveau 1 est très médiocre (voire mauvais) mais a l'avantage de la simplicité. On n'a pas à gérer de file d'attente ce qui est pratique dans le cadre d'un projet élève.
- Le niveau 2 n'a que peu d'intérêt. Il permet uniquement d'introduire la phase de tirs ciblés.
- Le niveau 3 est, quant à lui, très intéressant. On a une optimisation simple de la phase de tirs en aveugle et des performances tout à fait correctes.
- Le niveau 4 est intéressant dans le fait qu'il introduit un début d'intelligence artificielle. L'algorithme teste différentes combinaisons avant de prendre sa décision.
- Le niveau 5 est de loin le meilleur. L'optimisation est somme toute assez simple pour des résultats qui sont les meilleurs de tous.
- Le niveau 6 est plus une preuve de concept qu'un algorithme réellement utile en pratique. Les temps de calcul sont pharamineux, ce qui oblige de ne l'utiliser qu'à partir d'un certain moment, et les résultats sont assez décevants par rapport au niveau 5, auquel il n'apporte en fait rien. Il serait toutefois intéressant de le tester (avec un seuil de 100, c'est-à-dire en n'utilisant jamais le niveau 5) en disposant d'une grande puissance de calcul (et de trouver un moyen de paralléliser ceux-ci).

En tout état de cause il semble qu'une moyenne de 42 coups par partie constitue une borne que l'on ne pourra pas franchir.

CODES DES CARACTÈRES GRAPHIQUES

Voici la liste des caractères graphiques utilisés dans l’affichage en console. La plupart de ces caractères font partie de la famille Unicode Box Drawing, consultable sur la page <http://www.unicode.org/charts/PDF/U2500.pdf>.

```
# Caractères simples pour la grille
# -----
# Traits
CAR_H = u'\u2500'      # Trait Horizontal : -
CAR_V = u'\u2502'      # Trait Vertical : |
# Coins
CAR_CHG = u'\u250C'    # Coin Haut Gauche : ┌
CAR_CHD = u'\u2510'    # Coin Haut Droite : ┐
CAR_CBG = u'\u2514'    # Coin Bas Gauche : └
CAR_CBD = u'\u2518'    # Coin Bas Droite : ┘
# T
CAR_TH = u'\u252C'     # T Haut : ┴
CAR_TB = u'\u2534'     # T Bas : ┬
CAR_TG = u'\u251C'     # T Gauche : ├
CAR_TD = u'\u2524'     # T Droite : ┤
# +
CAR_CX = u'\u253C'     # Croix Centrale : ┼

# Caractères en gras pour les bateaux
# -----
# Traits
CAR_GH = u'\u2501'     # Trait Gras Horizontal : -
CAR_GV = u'\u2503'     # Trait Gras Vertical : |
# T
CAR_GTB = u'\u2537'    # T Gras Bas : ┴
CAR_GTD = u'\u2528'    # T Gras Droite : ┤
CAR_GTDH = u'\u252A'   # T Droite Haut : ┤
CAR_GTDB = u'\u2529'   # T Droite Bas : ┤
CAR_GTBG = u'\u253A'   # T Bas Gauche : ┴
CAR_GTBD = u'\u2539'   # T Bas Droite : ┴

# Coins
CAR_GCBD = u'\u251B'   # Coin Gras Bas Droite : ┘
```

ANNEXE C : CODES DES CARACTÈRES GRAPHIQUES

+

CAR_GCXHG = u'\u2546'

Croix Gras Haut Gauche : 

CAR_GCXHD = u'\u2545'

Croix Gras Haut Droite : 

CAR_GCXBG = u'\u2544'

Croix Gras Bas Gauche : 

CAR_GCXBD = u'\u2543'

Croix Gras Bas Droite : 


CAR_GCX = u'\u254B'

Croix Gras Centrale : 

CAR_GCXH = u'\u253F'

Croix Gras Horizontal : 

CAR_GCXV = u'\u2542'

Croix Gras Vertical : 

Touché / Manqué

CAR_TOUCH = u'\u2716'

Touché : ×

CAR_MANQ = u'\u25EF'

Manqué : ○

DOCUMENTATION DES MODULES

Les documentations ci-dessous ont été obtenues automatiquement grâce à la commande `pydoc3 <module.Classe>` et sont générés par les docstrings à l'intérieur de chaque objet. Leur intégration automatique en \LaTeX se fait avec :

```
\immediate\write18{pydoc3 module.Classe > ./docstrings/module.Classe}
\verbatiminput{./docstrings/module.Classe}
```

1 Module `bn_grille.py`

1.1 Classe Bateau

Help on class Bateau in bn_grille:

```
bn_grille.Bateau = class Bateau(builtins.object)
| Classe pour définir les bateaux
|
| Methods defined here:
|
| __init__(self, taille, start, sens)
|     Un bateau est défini par :
|         - Sa taille
|         - Son point de départ (start)
|         - Son sens (DROITE, GAUCHE, BAS, HAUT)
|     On récupère les cases occupées par le bateau
|     ainsi que les cases adjacentes
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)
```

1.2 Classe Grille

Help on class Grille in bn_grille:

```
bn_grille.Grille = class Grille(builtins.object)
| Classe pour définir la grille de jeu
|
| Methods defined here:
|
| __init__(self, xmax=10, ymax=10, taille_bateaux=[5, 4, 3, 3, 2])
|     Initialisation de la grille de jeu
|     Les cases sont numérotées de 0 à (x|y)max-1
|
| add_bateau(self, bateau)
|     Ajoute un bateau dans la grille
|     et met à jour les états des cases adjacentes
|
| add_bateau_alea(self, taille)
|     Ajoute un bateau aléatoire de taille donnée
|
| adjacent(self, case)
|     Retourne la liste des cases vides adjacentes à case
|     dans l'ordre : DROITE, GAUCHE, HAUT, BAS
|
| affiche(self)
|     Affiche la grille
|
| case_max(self, affiche=False)
|     Détermine la case qui contient le plus de bateaux et
|     regardant sur chaque case le nombre de bateaux possibles
|
| case_max_all(self, affiche_all=False)
|     Détermination de la case optimale par énumération de
|     toutes les répartitions possibles de bateaux
|
| case_max_echantillons(self, nb_echantillons=100, affiche=False)
|     Calcul des probabilités sur chaque case vide de contenir
|     un bateau. Retourne la case la plus probable en essayant
|     différents arrangements des bateaux restants
|
| case_max_touchee(self, case_touchee)
|     Retourne le nombre de bateaux possibles
|     sur chaque case adjacentes à case (qui vient d'être touchée)
```

```

|
| copie_grille_tmp(self)
|     Crée une copie temporaire de la grille
|
| elimine_cases(self)
|     Élimine les cases dans lesquelles
|     le plus petit bateau ne peut pas rentrer
|
| elimine_cases_joueur(self)
|     Élimine les cases dans lesquelles
|     le plus petit bateau ne peut pas rentrer
|     Version pour le joueur
|
| fini(self)
|     Renvoie True si tous les bateaux ont été coulés
|
| get_max_space(self, case, direction=(1, 1), sens=1)
|     Renvoie la plus grande place possible sur cette case
|     dans une direction
|
| get_possibles(self, affiche=False, tri=False)
|     Crée la liste des bateaux possibles démarrant sur chaque case
|     ainsi que la liste des cases et directions possibles pour
|     chaque bateau
|
| get_taille_max(self)
|     Met à jour la taille du bateau le plus grand restant
|
| get_taille_min(self)
|     Met à jour la taille du bateau le plus petit restant
|
| init_bateaux_alea(self)
|     Initialise une grille avec des bateaux aléatoires
|
| is_touche(self, case)
|     Teste si la case contient un bateau
|
| make_all(self, gtmp, affiche_all=False)
|     Crée toutes les répartitions possibles de bateaux
|
| reinit(self)
|     Réinitialisation de la grille
|

```

```

| rem_bateau(self, taille)
|     Enlève le bateau de taille taille de la liste
|
| test_bateau(self, bateau)
|     Test si le bateau est valide (rentre bien) dans la grille
|
| test_case(self, case)
|     Teste si une case est valide (dans la grille) et vide
|
| update(self)
|     Met à jour les paramètres de la grille
|
| update_vides(self)
|     Met à jour la liste des cases vides
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)

```

2 Module bn_joueur.py

2.1 Classe Joueur

Help on class Joueur in bn_joueur:

```

bn_joueur.Joueur = class Joueur(builtins.object)
| Classe pour définir le joueur
|
| Methods defined here:
|
| __init__(self, nom='Joueur')
|     Initialisation du joueur
|
| add_bateau(self, taille, start, direction)
|     Ajoute un bateau sur la grille du joueur
|

```

```

| add_message(self, texte)
|     Ajoute le message texte à la file des messages
|
| affiche_messages(self, affiche=True)
|     Affiche la liste des messages
|
| case_aleatoire(self)
|     Retourne une case aléatoire parmi les cases vides
|     (tire sur les cases noires tant qu'il en reste)
|
| check_coules(self)
|     Vérifie les bateaux coulés sur la grille
|     et les enlève de la liste des bateaux à chercher
|     en marquant leurs case adjacentes comme impossibles
|
| clean_grille(self)
|     Élimine de la grille les cases impossibles
|
| elimine_adjacentes(self, cases)
|     Élimine les cases adjacents à un bateau coulé
|
| elimine_petites(self)
|     Élimine les cases dans lesquelles le plus petit bateau
|     ne peut pas rentrer
|
| jeu_solo(self)
|     Lance une partie solo sur une grille aléatoire
|
| joue_coup(self)
|     Joue un coup
|
| rem_bateau(self, taille)
|     Enlève le dernier bateau coulé
|
| tire(self, case)
|     Tire sur la case (x,y)
|     Renvoie True si la case est touchée,
|     False si non touché ou case invalide
|
| tire_aleatoire(self)
|     Tire sur une case aléatoire
|
| -----

```

```

| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)

```

2.2 Classe Ordi

Help on class Ordi in bn_joueur:

```

bn_joueur.Ordi = class Ordi(Joueur)
| Algorithme de résolution
|
| Method resolution order:
|     Ordi
|     Joueur
|     builtins.object
|
| Methods defined here:
|
| __init__(self, nom='HAL', niveau=5, nb_echantillons=100, seuil=20)
|
| add_adjacentes_premiere(self)
|     Ajoute les cases adjacentes possibles à
|     la premiere case touchée dans la file d'attente
|
| add_queue(self, case)
|     Ajoute la case à la file d'attente
|
| affiche_bateaux(self)
|     Affiche la liste des bateaux restant à couler
|
| affiche_queue(self)
|     Affiche le contenu de la file d'attente
|
| affiche_suivi(self)
|     Affiche la grille de suivi des coups
|
| coup_suivant(self)
|     Fait jouer à l'ordinateur le coup suivant

```



```

|
|  elimine_adjacentes(self)
|      Élimine les cases adjacents à un bateau coulé
|
|  elimine_petites(self)
|      Élimine les cases dans lesquelles le plus petit bateau
|      ne peut pas rentrer
|
|  make_case_aleatoire(self)
|      Choisit une case aléatoire (suivant l'algorithme choisi)
|
|  pop_queue(self)
|      Récupère, en l'enlevant, le premier élément de la queue
|
|  rem_bateau(self)
|      Enlève le dernier bateau coulé
|
|  rem_queue(self, case)
|      Enlève la case de la file d'attente
|
|  resolution(self)
|      Lance la résolution de la grille par l'ordinateur
|
|  shuffle_queue(self)
|      Mélange les cases de la file d'attente en les triant
|      par ordre décroissant des bateaux possibles
|
|  test_plus_grand(self)
|      Renvoie True si on a touché autant de cases que
|      le plus grand bateau restant
|
|  tire_case_courante(self)
|      Tire sur la case courante
|
|  update_queue_manque(self)
|      Met à jour la file d'attente en éliminant une direction
|      impossible, après avoir manqué la case en face
|
|  update_queue_touche(self)
|      Met à jour la file d'attente en enlevant les cases
|      qui ne sont pas dans la bonne direction après avoir
|      touché une 2ème fois
|

```

```

|  vide_queue(self)
|      Vide la file d'attente
|
|  -----
|  Methods inherited from Joueur:
|
|  add_bateau(self, taille, start, direction)
|      Ajoute un bateau sur la grille du joueur
|
|  add_message(self, texte)
|      Ajoute le message texte à la file des messages
|
|  affiche_messages(self, affiche=True)
|      Affiche la liste des messages
|
|  case_aleatoire(self)
|      Retourne une case aléatoire parmi les cases vides
|      (tire sur les cases noires tant qu'il en reste)
|
|  check_coules(self)
|      Vérifie les bateaux coulés sur la grille
|      et les enlève de la liste des bateaux à chercher
|      en marquant leurs case adjacentes comme impossibles
|
|  clean_grille(self)
|      Élimine de la grille les cases impossibles
|
|  jeu_solo(self)
|      Lance une partie solo sur une grille aléatoire
|
|  joue_coup(self)
|      Joue un coup
|
|  tire(self, case)
|      Tire sur la case (x,y)
|      Renvoie True si la case est touchée,
|      False si non touché ou case invalide
|
|  tire_aleatoire(self)
|      Tire sur une case aléatoire
|
|  -----
|  Data descriptors inherited from Joueur:

```

```

|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)

```

2.3 Classe Partie

Help on class Partie in bn_joueur:

```

bn_joueur.Partie = class Partie(builtins.object)
|  Gère le déroulement de la partie
|
|  Methods defined here:
|
|  __init__(self, joueur=<bn_joueur.Joueur object at 0x7f715c3926a0>, adversaire=<b
|
|  add_bateau_joueur(self, taille)
|      Ajoute un bateau pour le joueur
|
|  get_bateaux_adverse(self)
|      Récupère la liste des bateaux adverses
|
|  get_coup_adverse(self)
|      Récupère le coup de l'adversaire
|
|  lance_partie(self)
|      Lance une partie à deux joueurs
|
|  place_bateaux_joueur(self)
|      Place tous les bateaux du joueur
|
|  -----
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)

```

3 Module bn_console.py

3.1 Classe GrilleC

Help on class GrilleC in bn_console:

```
bn_console.GrilleC = class GrilleC(bn_grille.Grille)
|   Affichage de la grille en mode console
|
|   Method resolution order:
|       GrilleC
|       bn_grille.Grille
|       builtins.object
|
|   Methods defined here:
|
|       __init__(self, xmax=10, ymax=10, taille_bateaux=[5, 4, 3, 3, 2])
|
|       affiche(self)
|           Affiche une grille
|
|       affiche_adverse(self, grille=None)
|           Affiche la grille de suivi de l'adversaire
|           en entourant nos propres bateaux
|
|       make_chaine(self)
|           Crée la grille avec des caractères graphiques
|
|       make_chaine_adverse(self, grille=None)
|           Crée la grille avec des caractères graphiques en entourant
|           en gras les bateaux de la grille passée en paramètre
|
|       -----
|       Methods inherited from bn_grille.Grille:
|
|       add_bateau(self, bateau)
|           Ajoute un bateau dans la grille
|           et met à jour les états des cases adjacentes
|
|       add_bateau_alea(self, taille)
|           Ajoute un bateau aléatoire de taille donnée
|
|       adjacent(self, case)
```

```

|     Retourne la liste des cases vides adjacentes à case
|     dans l'ordre : DROITE, GAUCHE, HAUT, BAS
|
|     case_max(self, affiche=False)
|         Détermine la case qui contient le plus de bateaux et
|         regardant sur chaque case le nombre de bateaux possibles
|
|     case_max_all(self, affiche_all=False)
|         Détermination de la case optimale par énumération de
|         toutes les répartitions possibles de bateaux
|
|     case_max_echantillons(self, nb_echantillons=100, affiche=False)
|         Calcul des probabilités sur chaque case vide de contenir
|         un bateau. Retourne la case la plus probable en essayant
|         différents arrangements des bateaux restants
|
|     case_max_touchee(self, case_touchee)
|         Retourne le nombre de bateaux possibles
|         sur chaque case adjacentes à case (qui vient d'être touchée)
|
|     copie_grille_tmp(self)
|         Crée une copie temporaire de la grille
|
|     elimine_cases(self)
|         Élimine les cases dans lesquelles
|         le plus petit bateau ne peut pas rentrer
|
|     elimine_cases_joueur(self)
|         Élimine les cases dans lesquelles
|         le plus petit bateau ne peut pas rentrer
|         Version pour le joueur
|
|     fini(self)
|         Renvoie True si tous les bateaux ont été coulés
|
|     get_max_space(self, case, direction=(1, 1), sens=1)
|         Renvoie la plus grande place possible sur cette case
|         dans une direction
|
|     get_possibles(self, affiche=False, tri=False)
|         Crée la liste des bateaux possibles démarrant sur chaque case
|         ainsi que la liste des cases et directions possibles pour
|         chaque bateau

```

```

|
|  get_taille_max(self)
|      Met à jour la taille du bateau le plus grand restant
|
|  get_taille_min(self)
|      Met à jour la taille du bateau le plus petit restant
|
|  init_bateaux_alea(self)
|      Initialise une grille avec des bateaux aléatoires
|
|  is_touche(self, case)
|      Teste si la case contient un bateau
|
|  make_all(self, gtmp, affiche_all=False)
|      Crée toutes les répartitions possibles de bateaux
|
|  reinit(self)
|      Réinitialisation de la grille
|
|  rem_bateau(self, taille)
|      Enlève le bateau de taille taille de la liste
|
|  test_bateau(self, bateau)
|      Test si le bateau est valide (rentre bien) dans la grille
|
|  test_case(self, case)
|      Teste si une case est valide (dans la grille) et vide
|
|  update(self)
|      Met à jour les paramètres de la grille
|
|  update_vides(self)
|      Met à jour la liste des cases vides
|
|  -----
|  Data descriptors inherited from bn_grille.Grille:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)

```

3.2 Classe JoueurC

Help on class JoueurC in bn_console:

```
bn_console.JoueurC = class JoueurC(bn_joueur.Joueur)
|   Joueur en mode console
|
|   Method resolution order:
|       JoueurC
|       bn_joueur.Joueur
|       builtins.object
|
|   Methods defined here:
|
|       __init__(self, nom='Joueur')
|
|       affiche_messages(self, affiche=True)
|           Affiche les messages du joueur
|
|       jeu_solo(self, cheat=True)
|           Lance une partie solo sur une grille aléatoire
|
|       joue_coup(self)
|           Joue un coup sur une case
|
|       -----
|
|       Methods inherited from bn_joueur.Joueur:
|
|       add_bateau(self, taille, start, direction)
|           Ajoute un bateau sur la grille du joueur
|
|       add_message(self, texte)
|           Ajoute le message texte à la file des messages
|
|       case_aleatoire(self)
|           Retourne une case aléatoire parmi les cases vides
|           (tire sur les cases noires tant qu'il en reste)
|
|       check_coules(self)
|           Vérifie les bateaux coulés sur la grille
|           et les enlève de la liste des bateaux à chercher
|           en marquant leurs case adjacentes comme impossibles
|
```

```

|  clean_grille(self)
|      Élimine de la grille les cases impossibles
|
|  elimine_adjacentes(self, cases)
|      Élimine les cases adjacents à un bateau coulé
|
|  elimine_petites(self)
|      Élimine les cases dans lesquelles le plus petit bateau
|      ne peut pas rentrer
|
|  rem_bateau(self, taille)
|      Enlève le dernier bateau coulé
|
|  tire(self, case)
|      Tire sur la case (x,y)
|      Renvoie True si la case est touchée,
|      False si non touché ou case invalide
|
|  tire_aleatoire(self)
|      Tire sur une case aléatoire
|
|  -----
|  Data descriptors inherited from bn_joueur.Joueur:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)

```

3.3 Classe OrdiC

Help on class OrdiC in bn_console:

```

bn_console.OrdiC = class OrdiC(JoueurC, bn_joueur.Ordi)
|  Résolution de la grille en mode console
|
|  Method resolution order:
|      OrdiC
|      JoueurC
|      bn_joueur.Ordi
|      bn_joueur.Joueur

```



```

|     builtins.object
|
| Methods defined here:
|
|     __init__(self, nom='HAL', niveau=4, nb_echantillons=100, seuil=20)
|
|     resolution(self, affiche=True, grille=None)
|         Lance la résolution de la grille par l'ordinateur
|
|     resolution_latex(self, affiche=True, grille=None)
|         Lance la résolution de la grille par l'ordinateur
|         avec affichage en LaTeX pour copier-coller dans le rapport
|
| -----
| Methods inherited from JoueurC:
|
|     affiche_messages(self, affiche=True)
|         Affiche les messages du joueur
|
|     jeu_solo(self, cheat=True)
|         Lance une partie solo sur une grille aléatoire
|
|     joue_coup(self)
|         Joue un coup sur une case
|
| -----
| Methods inherited from bn_joueur.Ordi:
|
|     add_adjacentes_premiere(self)
|         Ajoute les cases adjacentes possibles à
|         la premiere case touchée dans la file d'attente
|
|     add_queue(self, case)
|         Ajoute la case à la file d'attente
|
|     affiche_bateaux(self)
|         Affiche la liste des bateaux restant à couler
|
|     affiche_queue(self)
|         Affiche le contenu de la file d'attente
|
|     affiche_suivi(self)
|         Affiche la grille de suivi des coups

```

```

|
| coup_suivant(self)
|     Fait jouer à l'ordinateur le coup suivant
|
| elimine_adjacentes(self)
|     Élimine les cases adjacents à un bateau coulé
|
| elimine_petites(self)
|     Élimine les cases dans lesquelles le plus petit bateau
|     ne peut pas rentrer
|
| make_case_aleatoire(self)
|     Choisit une case aléatoire (suivant l'algorithme choisi)
|
| pop_queue(self)
|     Récupère, en l'enlevant, le premier élément de la queue
|
| rem_bateau(self)
|     Enlève le dernier bateau coulé
|
| rem_queue(self, case)
|     Enlève la case de la file d'attente
|
| shuffle_queue(self)
|     Mélange les cases de la file d'attente en les triant
|     par ordre décroissant des bateaux possibles
|
| test_plus_grand(self)
|     Renvoie True si on a touché autant de cases que
|     le plus grand bateau restant
|
| tire_case_courante(self)
|     Tire sur la case courante
|
| update_queue_manque(self)
|     Met à jour la file d'attente en éliminant une direction
|     impossible, après avoir manqué la case en face
|
| update_queue_touche(self)
|     Met à jour la file d'attente en enlevant les cases
|     qui ne sont pas dans la bonne direction après avoir
|     touché une 2ème fois
|

```

```

| vide_queue(self)
|     Vide la file d'attente
|
| -----
| Methods inherited from bn_joueur.Joueur:
|
| add_bateau(self, taille, start, direction)
|     Ajoute un bateau sur la grille du joueur
|
| add_message(self, texte)
|     Ajoute le message texte à la file des messages
|
| case_aleatoire(self)
|     Retourne une case aléatoire parmi les cases vides
|     (tire sur les cases noires tant qu'il en reste)
|
| check_coules(self)
|     Vérifie les bateaux coulés sur la grille
|     et les enlève de la liste des bateaux à chercher
|     en marquant leurs case adjacentes comme impossibles
|
| clean_grille(self)
|     Élimine de la grille les cases impossibles
|
| tire(self, case)
|     Tire sur la case (x,y)
|     Renvoie True si la case est touchée,
|     False si non touché ou case invalide
|
| tire_aleatoire(self)
|     Tire sur une case aléatoire
|
| -----
| Data descriptors inherited from bn_joueur.Joueur:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)

```

3.4 Classe PartieC

Help on class PartieC in bn_console:

```
bn_console.PartieC = class PartieC(bn_joueur.Partie)
|  Partie à deux joueurs en mode console
|
|  Method resolution order:
|      PartieC
|      bn_joueur.Partie
|      builtins.object
|
|  Methods defined here:
|
|      __init__(self, joueur=<bn_joueur.Joueur object at 0x7fe53a9460f0>, adversaire=<b
|
|      add_bateau_joueur(self, taille)
|          Ajoute un bateau pour le joueur
|
|      affiche_grilles(self, fin=False)
|          Affiche les deux grilles cote à cote,
|          avec les noms des joueurs
|
|      lance_partie(self)
|          Lance une partie à deux joueurs
|
|      place_bateaux_joueur(self)
|          Place tous les bateaux du joueur
|
|      -----
|      Methods inherited from bn_joueur.Partie:
|
|      get_bateaux_adverse(self)
|          Récupère la liste des bateaux adverses
|
|      get_coup_adverse(self)
|          Récupère le coup de l'adversaire
|
|      -----
|      Data descriptors inherited from bn_joueur.Partie:
|
|      __dict__
|          dictionary for instance variables (if defined)
```

```
|
|  __weakref__
|      list of weak references to the object (if defined)
```

3.5 Classe MainConsole

Help on class MainConsole in bn_console:

```
bn_console.MainConsole = class MainConsole(builtins.object)
|  Programme principal en mode console
|
|  Methods defined here:
|
|  __init__(self)
|
|  get_nb_echantillons(self, niveau)
|      Pour le niveau 4, demande la taille des échantillons
|
|  get_niveau(self)
|      Demande le niveau de l'ordinateur
|
|  get_seuil(self, niveau)
|      Pour le niveau 6, demande le seuil
|
|  jeu_contre_ordi(self, cheat=False, nom='Toto')
|      Partie en duel contre l'ordinateur
|
|  jeu_ordi(self, affiche=True, xmax=10, ymax=10, taille_bateaux=[5, 4, 3, 3, 2], n
|      Résolution d'une grille par l'ordinateur
|
|  jeu_solo(self, cheat=False)
|      Jeu solo sur une grille aléatoire
|
|  launch_menu(self)
|      Menu de lancement
|
|  launch_test_algo(self)
|      Lancement de la procédure de test
|      de l'algorithme de résolution
|
|  test_algo(self, n=1000, xmax=10, ymax=10, taille_bateaux=[5, 4, 3, 3, 2], niveau
|      Test de l'algorithme de résolution sur n parties
```

```

|         et affichage des statistiques
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)

```

4 Module bn_stats.py

4.1 Classe Stats

Help on class Stats in bn_stats:

```

bn_stats.Stats = class Stats(builtins.object)
|   Implémente les outils d'étude statistique
|
|   Methods defined here:
|
|   __init__(self, data=None, filename='', tmoy=0, param_grille={'taille_bateaux': [
|
|   get_all_stats(self)
|       Récupère tous les indicateurs statistiques
|
|   get_effectif(self)
|       Calcul de l'effectif total
|
|   get_maxi(self)
|       Calcul du maximum
|
|   get_mini(self)
|       Calcul du minimum
|
|   get_mode(self)
|       Calcul du mode
|
|   get_moyenne(self)
|       Calcul de la moyenne

```

```

|
| get_quartiles(self)
|     Calcul des quartiles. Renvoie une liste [Q1, Med, Q3].
|     Q1 et Q3 sont les termes de rangs  $\text{ceil}(n/4)$  et  $\text{ceil}(3n/4)$ 
|     La médiane est définie comme le terme de rang  $\text{ceil}(n/2)$ 
|
| get_sigma(self)
|     Calcul de l'écart-type
|
| histogramme(self, show=True, save=True)
|     Crée la représentation graphique des données avec :
|     - Un histogramme des fréquences
|     - Un diagramme en boîte à moustache
|     - Tous les indicateurs statistiques
|
| load_data(self)
|     Charge les données à partir d'un fichier texte
|
| resume_stat(self)
|     Affiche un résumé statistique
|
| save_data(self)
|     Sauvegarde les données dans un fichier texte
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)

```


L'ALGORITHME EN ACTION

Dans cet annexe, le déroulement d'une partie avec le niveau 5 :

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

<HAL> C'est parti !!!

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4										
5						○				
6										
7										
8										
9										

<HAL> Je tire sur la case F5 qui est la plus probable (34 bateaux possibles)
 <HAL> F5 : Manqué

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4					○					
5						○				
6										
7										
8										
9										

<HAL> Je tire sur la case E4 qui est la plus probable (34 bateaux possibles)
 <HAL> E4 : Manqué

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4					○					
5						○				
6							×			
7										
8										
9										

```

<HAL> Je tire sur la case G6 qui est la plus probable (32 bateaux possibles)
<HAL> G6 : Touché
<HAL> J'ajoute la case H6 à la file d'attente
<HAL> J'ajoute la case F6 à la file d'attente
<HAL> J'ajoute la case G5 à la file d'attente
<HAL> J'ajoute la case G7 à la file d'attente
<HAL> J'ordonne ma file d'attente en fonction des possibilités :
<HAL> F6 : 12 bateaux possibles
<HAL> G5 : 12 bateaux possibles
<HAL> H6 : 11 bateaux possibles
<HAL> G7 : 11 bateaux possibles
<HAL> File d'attente : F6 G5 H6 G7

```

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4					○					
5						○				
6						x	x			
7										
8										
9										

```

<HAL> Je tire sur la case F6 de la file d'attente
<HAL> F6 : Touché
<HAL> Le bateau touché est horizontal
<HAL> J'enlève la case G5 de la file d'attente
<HAL> J'enlève la case G7 de la file d'attente
<HAL> J'ajoute la case E6 à la file d'attente
<HAL> File d'attente : H6 E6

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4					○					
5						○				
6					x	x	x			
7										
8										
9										

```

<HAL> Je tire sur la case H6 de la file d'attente
<HAL> H6 : Touché
<HAL> J'ajoute la case I6 à la file d'attente
<HAL> File d'attente : E6 I6

```

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4					○					
5						○				
6					x	x	x	x		
7										
8										
9										

```

<HAL> Je tire sur la case E6 de la file d'attente
<HAL> E6 : Touché
<HAL> J'ajoute la case D6 à la file d'attente
<HAL> File d'attente : I6 D6

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4					○					
5						○				
6					x	x	x	x	x	
7										
8										
9										

```

<HAL> Je tire sur la case I6 de la file d'attente
<HAL> I6 : Touché
<HAL> J'ajoute la case J6 à la file d'attente
<HAL> File d'attente : D6 J6
<HAL> Bateau de taille 5 coulé car c'est le plus grand restant
<HAL> Je vide ma file d'attente

```


	A	B	C	D	E	F	G	H	I	J
0										
1										
2				○						
3										
4					○					
5					○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8										
9										

```

<HAL> Bateau de taille 5 coulé ! Je l'enlève de la liste des bateaux à chercher
<HAL> Bateaux restant à couler : 4 3 3 2
<HAL> J'élimine la case adjacente G5
<HAL> J'élimine la case adjacente G7
<HAL> J'élimine la case adjacente F7
<HAL> J'élimine la case adjacente H5
<HAL> J'élimine la case adjacente H7
<HAL> J'élimine la case adjacente D6
<HAL> J'élimine la case adjacente E5
<HAL> J'élimine la case adjacente E7
<HAL> J'élimine la case adjacente J6
<HAL> J'élimine la case adjacente I5
<HAL> J'élimine la case adjacente I7
<HAL> Je tire sur la case D2 qui est la plus probable (23 bateaux possibles)
<HAL> D2 : Manqué

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1										
2				○						
3			○							
4					○					
5					○	○	○	○	○	
6				○	×	×	×	×	×	○
7					○	○	○	○	○	
8										
9										

<HAL> Je tire sur la case C3 qui est la plus probable (23 bateaux possibles)
 <HAL> C3 : Manqué

	A	B	C	D	E	F	G	H	I	J
0										
1										
2				○			○			
3			○							
4					○					
5					○	○	○	○	○	
6				○	×	×	×	×	×	○
7					○	○	○	○	○	
8										
9										

<HAL> Je tire sur la case G2 qui est la plus probable (21 bateaux possibles)
 <HAL> G2 : Manqué

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3			○							
4					○					
5					○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8										
9										

<HAL> Je tire sur la case F1 qui est la plus probable (20 bateaux possibles)
 <HAL> F1 : Manqué

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3			○					○		
4					○					
5					○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8										
9										

<HAL> Je tire sur la case H3 qui est la plus probable (19 bateaux possibles)
 <HAL> H3 : Manqué

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3			○					○		
4					○					
5					○	○	○	○	○	
6				○	×	×	×	×	×	○
7					○	○	○	○	○	
8			○							
9										

<HAL> Je tire sur la case C8 qui est la plus probable (19 bateaux possibles)
 <HAL> C8 : Manqué

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3			○					○		
4		x			○					
5					○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9										

```

<HAL> Je tire sur la case B4 qui est la plus probable (19 bateaux possibles)
<HAL> B4 : Touché
<HAL> J'ajoute la case C4 à la file d'attente
<HAL> J'ajoute la case A4 à la file d'attente
<HAL> J'ajoute la case B3 à la file d'attente
<HAL> J'ajoute la case B5 à la file d'attente
<HAL> J'ordonne ma file d'attente en fonction des possibilités :
<HAL> B5 : 8 bateaux possibles
<HAL> B3 : 8 bateaux possibles
<HAL> C4 : 6 bateaux possibles
<HAL> A4 : 4 bateaux possibles
<HAL> File d'attente : B5 B3 C4 A4

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3			○					○		
4		x			○					
5		○			○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9										

<HAL> Je tire sur la case B5 de la file d'attente
 <HAL> B5 : Manqué

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3		○	○					○		
4		x			○					
5		○			○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9										

<HAL> Je tire sur la case B3 de la file d'attente
 <HAL> B3 : Manqué

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3		○	○					○		
4		x	x		○					
5		○			○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9										

```

<HAL> Je tire sur la case C4 de la file d'attente
<HAL> C4 : Touché
<HAL> Le bateau touché est horizontal
<HAL> J'ajoute la case D4 à la file d'attente
<HAL> File d'attente : A4 D4

```

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3		○	○					○		
4	○	x	x		○					
5		○			○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9										

<HAL> Je tire sur la case A4 de la file d'attente
 <HAL> A4 : Manqué

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0										
1						○				
2				○			○			
3		○	○					○		
4	○	x	x	x	○					
5		○			○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9										

<HAL> Je tire sur la case D4 de la file d'attente
 <HAL> D4 : Touché

	A	B	C	D	E	F	G	H	I	J
0					○					
1						○				
2				○			○			
3		○	○	○				○		
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9										

```

<HAL> Bateau de taille 3 coulé ! Je l'enlève de la liste des bateaux à chercher
<HAL> Bateaux restant à couler : 4 3 2
<HAL> J'élimine la case adjacente C5
<HAL> J'élimine la case adjacente D3
<HAL> J'élimine la case adjacente D5
<HAL> Je tire sur la case E0 qui est la plus probable (12 bateaux possibles)
<HAL> E0 : Manqué

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0					○					
1						○			○	
2				○			○			
3		○	○	○				○		
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9										

<HAL> Je tire sur la case I1 qui est la plus probable (11 bateaux possibles)
 <HAL> I1 : Manqué

	A	B	C	D	E	F	G	H	I	J
0					○					
1						○			○	
2				○			○			
3		○	○	○				○		
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9				x						

```

<HAL> Je tire sur la case D9 qui est la plus probable (11 bateaux possibles)
<HAL> D9 : Touché
<HAL> J'ajoute la case E9 à la file d'attente
<HAL> J'ajoute la case C9 à la file d'attente
<HAL> J'ajoute la case D8 à la file d'attente
<HAL> J'ordonne ma file d'attente en fonction des possibilités :
<HAL> E9 : 6 bateaux possibles
<HAL> C9 : 6 bateaux possibles
<HAL> D8 : 2 bateaux possibles
<HAL> File d'attente : E9 C9 D8

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0					○					
1						○			○	
2				○			○			
3		○	○	○				○		
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9				x	○					

<HAL> Je tire sur la case E9 de la file d'attente
 <HAL> E9 : Manqué

	A	B	C	D	E	F	G	H	I	J
0					○					
1						○			○	
2				○			○			
3		○	○	○				○		
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9			x	x	○					

```

<HAL> Je tire sur la case C9 de la file d'attente
<HAL> C9 : Touché
<HAL> Le bateau touché est horizontal
<HAL> J'enlève la case D8 de la file d'attente
<HAL> J'ajoute la case B9 à la file d'attente
<HAL> File d'attente : B9

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0					○					
1						○			○	
2				○			○			
3		○	○	○				○		
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9		x	x	x	○					

```

<HAL> Je tire sur la case B9 de la file d'attente
<HAL> B9 : Touché
<HAL> J'ajoute la case A9 à la file d'attente
<HAL> File d'attente : A9

```

	A	B	C	D	E	F	G	H	I	J
0					○					
1						○			○	
2				○			○			
3		○	○	○				○		
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8			○							
9	○	x	x	x	○					

<HAL> Je tire sur la case A9 de la file d'attente
 <HAL> A9 : Manqué

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○				○			○	
2				○			○			
3		○	○	○				○		
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8		○	○	○						
9	○	x	x	x	○					

```

<HAL> Bateau de taille 3 coulé ! Je l'enlève de la liste des bateaux à chercher
<HAL> Bateaux restant à couler : 4 2
<HAL> J'élimine la case adjacente D8
<HAL> J'élimine la case adjacente B8
<HAL> Je tire sur la case B1 qui est la plus probable (6 bateaux possibles)
<HAL> B1 : Manqué

```

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○				○			○	
2				○			○			
3		○	○	○				○		x
4	○	x	x	x	○					
5		○	○	○	○	○	○	○	○	
6				○	x	x	x	x	x	○
7					○	○	○	○	○	
8		○	○	○						
9	○	x	x	x	○					

```

<HAL> Je tire sur la case J3 qui est la plus probable (6 bateaux possibles)
<HAL> J3 : Touché
<HAL> J'ajoute la case I3 à la file d'attente
<HAL> J'ajoute la case J2 à la file d'attente
<HAL> J'ajoute la case J4 à la file d'attente
<HAL> J'ordonne ma file d'attente en fonction des possibilités :
<HAL> J2 : 4 bateaux possibles
<HAL> J4 : 3 bateaux possibles
<HAL> I3 : 1 bateaux possibles
<HAL> File d'attente : J2 J4 I3

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○				○			○	
2				○			○			×
3		○	○	○				○		×
4	○	×	×	×	○					
5		○	○	○	○	○	○	○	○	
6				○	×	×	×	×	×	○
7					○	○	○	○	○	
8		○	○	○						
9	○	×	×	×	○					

```

<HAL> Je tire sur la case J2 de la file d'attente
<HAL> J2 : Touché
<HAL> Le bateau touché est vertical
<HAL> J'enlève la case I3 de la file d'attente
<HAL> J'ajoute la case J1 à la file d'attente
<HAL> File d'attente : J4 J1

```

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○				○			○	
2				○			○			×
3		○	○	○				○		×
4	○	×	×	×	○					○
5		○	○	○	○	○	○	○	○	
6				○	×	×	×	×	×	○
7					○	○	○	○	○	
8		○	○	○						
9	○	×	×	×	○					

<HAL> Je tire sur la case J4 de la file d'attente
 <HAL> J4 : Manqué

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○				○			○	○
2				○			○			×
3		○	○	○				○		×
4	○	×	×	×	○					○
5		○	○	○	○	○	○	○	○	
6				○	×	×	×	×	×	○
7					○	○	○	○	○	
8		○	○	○						
9	○	×	×	×	○					

<HAL> Je tire sur la case J1 de la file d'attente
 <HAL> J1 : Manqué

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○	○	○	○	○	○	○	○	○
2		○	○	○	○	○	○	○	○	×
3		○	○	○	○	○	○	○	○	×
4	○	×	×	×	○					○
5		○	○	○	○	○	○	○	○	○
6		○	○	○	×	×	×	×	×	○
7					○	○	○	○	○	○
8		○	○	○			×			
9	○	×	×	×	○					

```

<HAL> Bateau de taille 2 coulé ! Je l'enlève de la liste des bateaux à chercher
<HAL> Bateaux restant à couler : 4
<HAL> J'élimine la case adjacente I3
<HAL> J'élimine la case adjacente I2
<HAL> J'élimine la cases B2 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases B6 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases C1 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases C2 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases C6 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases D1 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases E1 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases E2 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases E3 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases F2 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases F3 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases G1 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases G3 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases H1 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases H2 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases J5 : zone trop petite pour le plus petit bateau de taille 4
<HAL> J'élimine la cases J7 : zone trop petite pour le plus petit bateau de taille 4
<HAL> Je tire sur la case G8 qui est la plus probable (3 bateaux possibles)
<HAL> G8 : Touché
<HAL> J'ajoute la case H8 à la file d'attente
<HAL> J'ajoute la case F8 à la file d'attente
<HAL> Le plus petit bateau, de taille 4, ne rentre pas verticalement en case G8
<HAL> J'ordonne ma file d'attente en fonction des possibilités :
<HAL> H8 : 3 bateaux possibles
<HAL> F8 : 2 bateaux possibles
<HAL> File d'attente : H8 F8

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○	○	○	○	○	○	○	○	○
2		○	○	○	○	○	○	○	○	×
3		○	○	○	○	○	○	○	○	×
4	○	×	×	×	○					○
5		○	○	○	○	○	○	○	○	○
6		○	○	○	×	×	×	×	×	○
7					○	○	○	○	○	○
8		○	○	○			×	×		
9	○	×	×	×	○					

```

<HAL> Je tire sur la case H8 de la file d'attente
<HAL> H8 : Touché
<HAL> Le bateau touché est horizontal
<HAL> J'ajoute la case I8 à la file d'attente
<HAL> File d'attente : F8 I8

```

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○	○	○	○	○	○	○	○	○
2		○	○	○	○	○	○	○	○	×
3		○	○	○	○	○	○	○	○	×
4	○	×	×	×	○					○
5		○	○	○	○	○	○	○	○	○
6		○	○	○	×	×	×	×	×	○
7					○	○	○	○	○	○
8		○	○	○		×	×	×		
9	○	×	×	×	○					

```

<HAL> Je tire sur la case F8 de la file d'attente
<HAL> F8 : Touché
<HAL> J'ajoute la case E8 à la file d'attente
<HAL> File d'attente : I8 E8

```

ANNEXE E : L'ALGORITHME EN ACTION

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○	○	○	○	○	○	○	○	○
2		○	○	○	○	○	○	○	○	×
3		○	○	○	○	○	○	○	○	×
4	○	×	×	×	○					○
5		○	○	○	○	○	○	○	○	○
6		○	○	○	×	×	×	×	×	○
7					○	○	○	○	○	○
8		○	○	○		×	×	×	○	
9	○	×	×	×	○					

<HAL> Je tire sur la case I8 de la file d'attente
 <HAL> I8 : Manqué

	A	B	C	D	E	F	G	H	I	J
0					○					
1		○	○	○	○	○	○	○	○	○
2		○	○	○	○	○	○	○	○	×
3		○	○	○	○	○	○	○	○	×
4	○	×	×	×	○					○
5		○	○	○	○	○	○	○	○	○
6		○	○	○	×	×	×	×	×	○
7					○	○	○	○	○	○
8		○	○	○	×	×	×	×	○	
9	○	×	×	×	○					

```

<HAL> Je tire sur la case E8 de la file d'attente
<HAL> E8 : Touché
<HAL> Bateau de taille 4 coulé car c'est le plus grand restant
<HAL> Je vide ma file d'attente
<HAL> Partie terminée en 36 coups

```