

BATAILLE NAVALE

Projet de validation ISN 2016
de l'académie de Lyon

FRÉDÉRIC MULLER - maths.muller@gmail.com
LIONEL REBOUL

13 mars 2016

TABLE DES MATIÈRES

1	Présentation du projet	5
1	Le jeu de la bataille navale	5
2	Objectifs du projet	5
3	Liste des modules du projet	6
4	Constantes de direction	6
2	Gestion de la grille	7
1	La classe Bateau	7
2	La classe Grille	7
3	Gestion des joueurs et de la partie	13
4	Algorithme de résolution	15
1	Description de l'algorithme	15
2	Étude statistique	16
5	Affichage console	17
6	Interface graphique	19
7	Guide des modules utilisés	21
8	Point de vue pédagogique	23
9	Conclusion	25

TABLE DES MATIÈRES

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, chacun son tour, de deviner l'emplacement des bateaux de l'adversaire par des tirs successifs, ce dernier annonçant à chaque coup « manqué » ou « touché ». Nous avons 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 règles retenues dans ce projet sont les règles du jeu original, mais elles peuvent être facilement modifiées, 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 l'algorithme de résolution :

- Le plus petit bateau étant de taille 2, il suffit de ne tirer que sur une cases sur 2 (imaginez les cases noires d'un damier) lors de la recherche d'un bateau.
- Une fois qu'un bateau a été coulé (soit parce que c'est le plus grand de la liste, soit parce que les cases adjacentes à ses extrémités ont été manquées), on peut éliminer de la recherche toutes ses cases adjacentes.

2 Objectifs du projet

Nos 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. Cette interface a été réalisée d'un part en mode console avec un affichage grâce à des caractères graphiques (en unicode) et, d'autre part, avec le module tkinter.

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_console.py` : toute l'interface en mode console, et l'étude statistique de l'algorithme de résolution.

4 Constantes de direction

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

- `BN_DROITE = (1, 0)`
- `BN_GAUCHE = (-1, 0)`
- `BN_BAS = (1, 0)`
- `BN_HAUT = (-1, 0)`

Ainsi que :

- `BN_ALLDIR = (1, 1)` (toutes les direction)
- `BN_HORIZONTAL = (1, 0)` (à gauche et à droite)
- `BN_VERTICAL = (0, 1)` (en haut et en bas)

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

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, sa taille et sa direction. Elle permet de récupérer :

- sa case de fin,
- la liste de ses cases occupées,
- la liste de ses cases adjacentes.

2 La classe Grille

2.1 Présentation

Cette classe est l'une des principales 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 fonctions 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), nous avons 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 nous nous sommes rendu compte que cela n'apportait pas d'avantage significatif en terme de qualité de code donc nous ne les utiliseront pas, mais elles sont encore présentes dans notre 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 non jouée
- 1 : case touchée
- -1 : case manquée ou impossible

L'intérêt d'utiliser un dictionnaire plutôt qu'une double liste tient au fait que les appels sont plus simples et plus naturels et, surtout, que l'utilisation d'une table de hachage permet la recherche d'un élément en $O(1)$.

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 des fonctions de mise à jour de l'état de la grille (liste des cases vides, tailles des plus petits et plus grand bateaux restants).

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. L'algorithme est le suivant :

```

0 → m
case[0] → x
case[1] → y
Tant que la case (x+direction[0], y+direction[1]) est vide :
    m+1 → m
    x+1 → x
    y+1 → y
Retourner m

```

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

La méthode `Grille.elimine_petites(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 de bateaux possibles sur chaque case

La méthode `Grille.get_possibles(self)` renvoie d'une part la liste des bateaux possibles sur chaque case (ainsi que leur direction) et, d'autre part, la liste des positions (et directions) 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 rentre. 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)),...], (0,1):...}`
- 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 nous servir à faire deux choses :

- Placer les bateaux aléatoirement grâce au dictionnaire `Grille.possibles`
- Déterminer la case optimale dans l'algorithme de résolution

2.5 Nombre de possibilités sur chaque case

L'une des parties importantes de l'algorithme de résolution consiste en la détermination de la case dans laquelle rentrent le plus de bateaux. Cette question intervient lors de la phase de tirs en aveugle et lorsqu'on a touché une première case et qu'on doit tester ses cases adjacentes (phase de tir ciblé).

2.5.1 Optimisation de la phase de tir en aveugle

La méthode `Grille.case_max(self)` renvoie la case vide contenant le plus de bateaux, ainsi que le nombre de bateaux qu'elle contient. L'algorithme est très simple : d'abord on crée un dictionnaire `Grille.probas` indexé par les cases et contenant le nombre de bateaux possibles grâce à `Grille.possibles`. Ensuite il ne reste plus qu'à renvoyer celle qui en contient le plus.

2.5.2 Optimisation de la phase de tir ciblé

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

Notons `(x, y)` les coordonnées de `case_touchee` 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			

Les cases (4,0) et (6,0) sont augmentées de 1

	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 1

- 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				

2.6 Gestion de la flotte

Le 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.

2.7 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.

Pour initialiser une flotte aléatoire, c'est la méthode `Grille.init_bateaux_alea(self)` dont l'algorithme est le suivant :

```

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 bateau à placer :
        On récupère les positions possibles pour ce bateau dans gtmp
        Si aucune possibilité :
            On casse la boucle et on recommence tout
            (pour éviter une situation de blocage)
        Sinon :
            On choisi une position et une direction au hasard
            (parmi celles possibles)
            On ajoute la bateau à gtmp
            nb_bateaux+1→nb_bateaux
Enfin on copie l'état de gtmp dans notre grille

```

2.8 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

ALGORITHME DE RÉOLUTION

1 Description de l'algorithme

L'algorithme de résolution est implémenté dans la classe `Ordi (Joueur)` du module `bn_joueur.py` (qui hérite donc de la classe `Joueur`). Il fonctionne en deux temps : dans un premier temps une phase de tir en aveugle et, une fois qu'une case a été touchée, une phase de tir ciblé jusqu'à ce que le bateau soit coulé.

1.1 Phase de tir en aveugle

Lors de cette phase, l'algorithme va tirer sur la case qui peut contenir le plus de bateau comme vu au chapitre 2, section 2.5.1.

C'est la méthode la plus efficace que nous ayons trouvée. Néanmoins nous avons fait d'autres essais avec d'autres méthodes mais celles-ci étaient beaucoup moins performantes, que ce soit aussi bien en nombre de coup pour la résolution qu'en temps :

- La première méthode consiste à tout simplement tirer au hasard sur une case vide.
- On peut raffiner la méthode précédente en ne tirant que sur une case sur deux (le plus petit bateau étant de taille 2, chaque bateau tombe obligatoirement sur une case noire du damier).
- Nous avons aussi essayé de déterminer la case la plus probable en créant un échantillon d'un certain nombre n de répartitions aléatoires des bateaux restant sur le grille et en comptant, pour chaque case, le nombre de bateaux la contenant. Les performances en nombre d'essais étaient satisfaisantes, mais le temps de calcul beaucoup trop élevé. Voici un petit tableau récapitulatif de quelques essais avec différents paramètres :

Taille des échantillons	100	1 000	10 000	100 000
Nombre de parties	10 000	10 000	1 000	100
Nombre de coups moyens	43,68	43,30	42,72	42,63
Temps moyen par partie (en secondes)	0,38	3,6	36,2	380

Temps mesurés sur un processeur Intel Core i7 4800-MQ à 2,7 GHz

Au final, le temps de résolution étant linéaire en n pour des performances négligeables, cette approche a été abandonnée.

- Enfin, une dernière approche consisterait à 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 astronomique de configurations, cette approche est irréalisable que ce soit en temps de calcul qu'en utilisation mémoire.

1.2 Phase de tir ciblé

1.3 Algorithme complet

2 Étude statistique

AFFICHAGE CONSOLE

INTERFACE GRAPHIQUE

GUIDE DES MODULES UTILISÉS

POINT DE VUE PÉDAGOGIQUE

CONCLUSION