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

Chapitre 1

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.

Chapitre 2

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'un 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 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 méthodes `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→x
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 parcours 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**				

Une fois que le compte des bateaux possibles a été effectué sur chacune de 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.

2.6 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'etat 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.

Chapitre 3

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 seront largement héritées dans la suite (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 et le nombre de coups joués.

La méthode principale de cette classe est `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).

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 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 (notons l'instruction

```
isinstance(self.adversaire, Ordi)
```

qui permet de savoir que ce dernier est l'ordinateur), de placer les bateaux du joueur et de récupérer les paramètres de l'adversaire (sa grille et le coup qu'il vient de jouer).

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

Chapitre 4

ALGORITHME DE RÉSOLUTION

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 coups pour la résolution, qu'en temps :

- La première méthode consiste tout simplement à 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 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 gains de 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.

Lors de cette phase on va également, avant chaque coup, éliminer les cases dans lesquelles le plus petit bateau restant à trouver ne rentre pas.

1.2 Phase de tir ciblé

1.2.1 Premier tir

Lors du premier tir touché après la phase de tir en aveugle, on va garder une trace de la case touchée dans l'attribut `Ordi.case_touchee` et on va fabriquer une file d'attente dans la liste `Ordi.queue` qui va contenir la liste des prochaines cases à viser. À cette étape, cette file d'attente contient les cases vides adjacentes classées en ordre décroissant de nombre de bateaux possibles comme vu au chapitre 2, section 2.5.2.

On va également créer une liste `Ordi.liste_touches` qui va garder la trace des cases touchées sur ce bateau.

1.2.2 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.

- Si on touche alors, grâce à la méthode `Ordi.update_queue_touche(self)`, on détermine la direction du bateau (horizontal ou vertical) et on enlève les cases de la file d'attente qui ne sont pas dans la bonne direction. On ajoute enfin les 2 cases aux extrémités de la configuration créée à la file d'attente et on met à jour la liste `Ordi.liste_touches`.
- 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 e 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**		O				

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

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

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

1.2.3 Tirs suivants

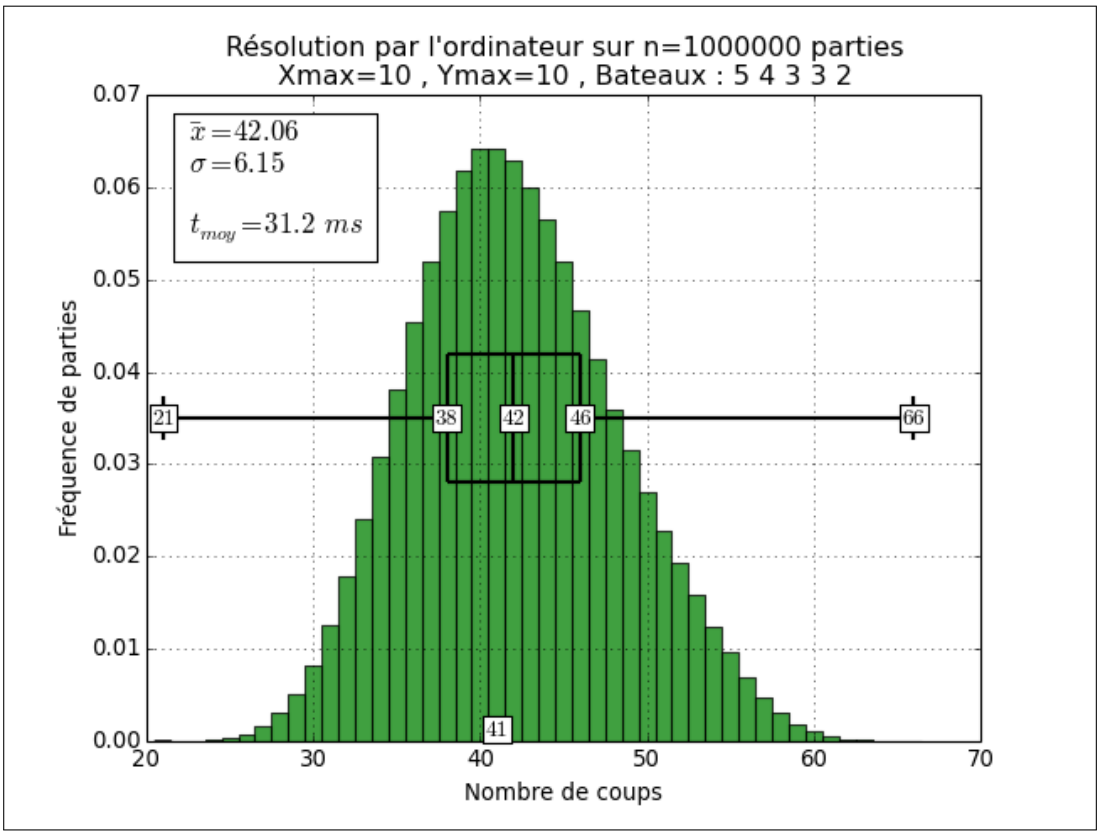
Une fois que la direction du bateau est déterminée, à chaque fois qu'on touche une case, on ajoute à la file d'attente sa case adjacente dans la bonne direction.

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 et, dans ce cas, on vide la file d'attente. La méthode `Ordi.liste_touches` permet de garder la trace des cases touchées sur ce bateau et d'en déterminer le nombre de cases.

Au prochain tour, on sait qu'un bateau vient d'être coulé lorsque la file d'attente est vide et `Ordi.liste_touches` 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.

2 Étude statistique

Des tests de l'algorithme de résolution sur $n = 1\,000\,000$ de parties donnent les résultats suivants, obtenus avec les modules `numpy` et `matplotlib` :



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

La forme de cette distribution semble correspondre à une loi normale asymétrique.

Chapitre 5

AFFICHAGE CONSOLE

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

1 Préliminaires

1.1 Constantes graphiques

Pour afficher les grilles nous utilisons 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).

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 nous utilisons 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.join('\n')`. Ensuite, en prenant les éléments 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.

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()` 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.

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

2.2 Affichage de la grille avec ses propres bateaux surlignés

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 (en fait on ne test que si `grille.etat[case]==1` ou si `grille.etat[case]!=1`).

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 des cases de la grille se fera sur la ligne du bas (hormis la première ligne) et sur la droite (hormis pour la première colonne). Le cas de la dernière ligne horizontale et de la dernière ligne verticale se fera à part.

Pour savoir la configuration d'une case on va donc devoir tester, outre cette dernière, sa case à droite et sa case en dessous (pour savoir si on commence un bateau, si on le termine, si on est dedans ou encore s'il n'y a pas de bateau dans ce secteur) ainsi que sa

case en dessous à droite (pour savoir s'il n'y a pas deux bateaux qui se touchent sur un coin).

Chapitre 6

INTERFACE GRAPHIQUE

Chapitre 7

GUIDE DES MODULES UTILISÉS

Chapitre 8

POINT DE VUE PÉDAGOGIQUE

Chapitre 9

CONCLUSION