[LU3IN005] Statistiques et informatique

Projet 1: Bataille Navale

GIANG Phuong Thu, Cécile

Mars 2020

Sorbonne Université, Faculté de Sciences et Ingénierie

1 Introduction

Un ***jeu de hasard*** est un jeu dont le déroulement est partiellement ou totalement soumis à la chance (tirage ou distribution de cartes, jet de dé, etc...). Or, contrairement aux idées reçues, tous les jeux de hasard ne sont pas soumis au hasard pur !

Alors qu'il est tout à fait possible (parfois avec l'aide d'une chance incroyable) de gagner en décidant aléatoirement de la prochaine manoeuvre à jouer, nous pouvons également décider de nous *adapter au jeu* en question et développer des **stratégies**, basées sur des notions de probabilités, et qui non seulement augmentent grandement nos chances de gagner, mais nous garantissent parfois une victoire plus rapide !

Ainsi au XVIe siècle, le chercheur italien Girolamo Cardano, dépendant au jeu, a été le premier à noter que quand deux dès affichaient un double six, il ne s'agissait pas d'un simple hasard. En s'appuyant sur ses observations, il écrit le *Livre du jeu de hasard, un guide pratique* où il décrit l'espace des événements élémentaires — *c'est-à-dire toutes les issues mutuellement exclusives de l'expérience* — pour jouer aux dés *(sur les 36 combinaisons possibles une sera 6:6)* et a élaboré un système pour calculer la probabilité de gain. Malgré certaines lacunes, son travail a lancé le développement des **théories des combinaisons et des probabilités**.

Dans ce projet, nous comparerons trois stratégies pour le choix des manoeuvres avec l'exemple du jeu de la **Bataille Navale**: une décision aléatoire, une autre heuristique *(ie explorant les coups précédents et prenant une décision selon un certain nombre de règles)* et une décision probabiliste simplifiée. Nous étudierons leur efficacité en terme d'assurance de victoire, et leur influence sur nombre de coups à jouer avant de gagner. Il s'agira dans un premier temps d'étudier la combinatoire du jeu, puis de proposer une modélisation du jeu afin d'optimiser les chances d'un joueur de gagner et enfin d'étudier une variante du jeu plus réaliste.

2 Bataille navale

Le jeu de la bataille navale se joue sur une grille de 10 cases par 10 cases. L'adversaire place sur cette grille un certain nombre de bateaux qui sont caractérisés par leur longueur :

- un porte-avions (5 cases)
- un croiseur (4 cases)
- un contre-torpilleurs (3 cases)
- un sous-marin (3 cases)
- un torpilleur (2 cases)

Il a le droit de les placer verticalement ou horizontalement. Le positionnement des bateaux reste secret, l'objectif du joueur est de couler tous les bateaux de l'adversaire en un nombre minimum de coup. A chaque tour de jeu, il choisit une case où il tire : si il n'y a aucun bateau sur la case, la réponse est *vide* ; si la case est occupée par une partie d'un bateau, la réponse est *touché*; si toutes les cases d'un bateau ont été touchées, alors la réponse est *coulé* et les cases correspondantes sont révélées. Lorsque tous les bateaux ont été coulés, le jeu s'arrête et le score du joueur est le nombre de coups qui ont été joués. Plus le score est petit, plus le joueur est performant.

3 Modélisation et fonctions simples

Nous avons choisi de modéliser la grille de jeu par une matrice d'entiers de taille 10 par 10, à l'aide d'un array la librairie `NumPy`. Nous définissons également un dictionnaire que l'on nomme `dic_bateaux`, où chaque bateau est caractérisé par un identifiant chiffré (sa clé) et a pour valeur le nombre de cases qu'il occupe dans la grille:

- un porte-avion (1): 5 cases
- un croiseur (2): 4 cases
- un contre-torpilleurs (3): 3 cases
- un sous-marin (4): 3 cases
- un torpilleur (5): 2 cases

Ainsi, une grille de jeu est constituée de cases à 0 (c'est-à-dire vides), et des cases qui contiendront un entier correspondant au bateau qui l'occupent.
Nous implémentons ensuite quelques fonctions de base pour manipuler notre grille:

- `peut_placer(grille,bateau,position,direction)` qui rend vrai s'il est possible de placer le bateau sur la grille à la position donnée
- `place(grille,bateau,position,direction)` qui rend la grille modifiée s'il est possible de placer le bateau
- `place_alea(grille, bateau)` qui place aléatoirement le bateau dans la grille
- `affiche(grille)` qui affiche la grille de jeu
- `eq(grilleA,grilleB)` qui teste l'égalité entre deux grilles
- `genere_grille()` qui rend une grille avec les bateaux disposés de manière aléatoire

Voici deux exemples de grilles où les bateaux ont été aléatoirement placés:

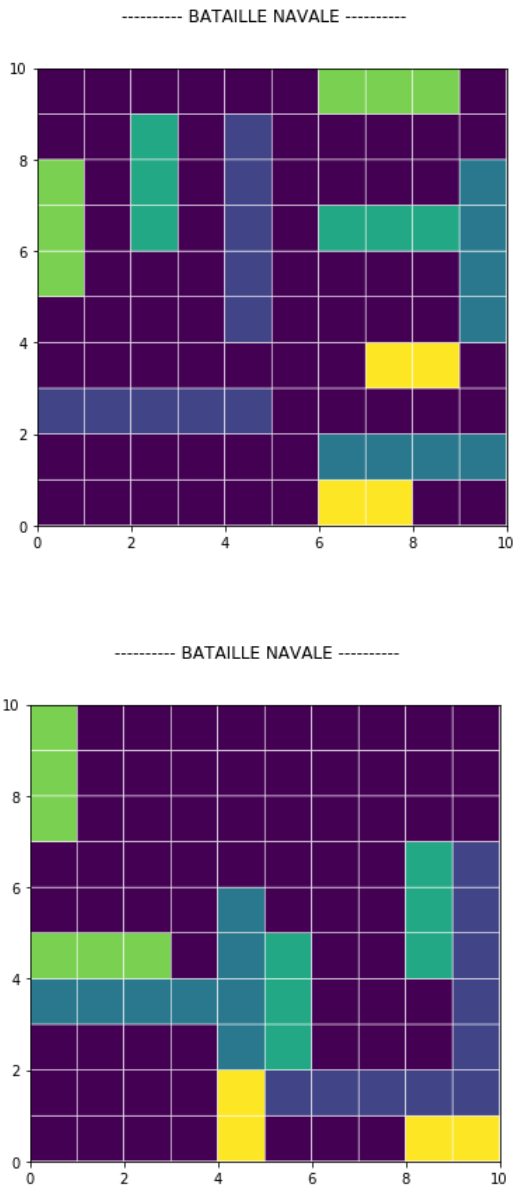

Figure 1: Exemples de grilles aléatoires

4 Combinatoire du jeu

Il s'agit maintenant de se pencher sur la combinatoire du jeu. Pour cela, nous nous intéresserons au nombre de grilles possibles (*ie le nombre de configurations possibles*) dans différentes conditions.

1. Borne supérieure du nombre de configurations possibles pour la liste complète de bateaux sur une grille de taille 10 (calcul à la main)

Commençons tout d'abord par préciser qu'il serait bien trop difficile de calculer exactement **le nombre de configurations possibles de notre grille** pour la liste complète de bateaux. Il est par contre tout à fait possible d'en estimer grossièrement une *borne supérieure* !

On omettra ainsi de supprimer les cas où des bateaux sont superposés, ie qu'une ou plusieurs cases de notre grille peuvent être occupées par plusieurs bateaux (ce qui, dans la réalité, serait impossible).

Ainsi, pour chaque joueur:

- il y a (10 - 5) + 1 = 6 possibilités de placer **un bateau de 5 cases** (adjacentes) sur une ligne ou une colonne. Pour notre grille de 10 lignes et 10 colonnes, cela revient donc à 6 x 20 = 120 façons de placer un porte-avion dans la grille
- de la même manière, il y a (10 - 4) + 1 = 7 possibilités de placer **un bateau de 4 cases** sur une ligne ou une colonne, soit 7 x 20 = 140 façons de placer un croiseur dans la grille
- (10 - 3) + 1 = 8 possibilités de placer **un bateau de 3 cases** sur une ligne ou une colonne, soit 8 x 20 = 160 façons de placer un contre-torpilleur dans la grille
- (10 - 3) + 1 = 8 possibilités de placer **un bateau de 3 cases** sur une ligne ou une colonne, soit 8 x 20 = 160 façons de placer un sous-marin dans la grille
- (10 - 2) + 1 = 9 possibilités de placer **un bateau de 2 cases** sur une ligne ou une colonne, soit 9 x 20 = 180 façons de placer un torpilleur dans la grille

Il y a donc au plus 120 x 140 x 160 x 160 x 180 = 77 414 400 000 configurations possibles de la grille pour 1 joueur, soit $(120 \times 140 \times 160 \times 160 \times 180)^2$ configurations possibles pour 2 joueurs. C'est notre borne supérieure.

Nous essayons maintenant d'implémenter une fonction informatique pour vérifier le calcul que nous avons effectué à la main. Il s'agit simplement, pour chaque joueur et pour chacun de ses bateaux, de parcourir notre grille de case en case et vérifier si l'on peut le placer à cette position là, dans les directions horizontale ou verticale. On compte ainsi le nombre de positions possibles dans la grille pour chaque bateau; pour trouver le nombre de configurations possibles de la grille il suffit de multiplier les résultats pour chacun des bateaux des deux joueurs.

Nous obtenons bien les mêmes résultats que pour le calcul à la main.

2. Nombre de façon de placer une liste de bateaux sur une grille

Nous définissons maintenant une fonction qui prendra en argument une liste de bateaux, et qui comptera le nombre de configurations possibles de la grille initialiement vide en procédant *par récurrence*: pour chaque bateau de la liste et chacun de ses emplacements possibles dans les deux directions horizontale et verticale, on rappelle notre fonction sur le reste des bateaux de la liste qui n'ont pas encore été traités.

Nous obtenons ainsi les résultats suivants pour une liste de 1, 2 puis 3 bateaux dont les identifiants sont respectivement 1 (porte-avion de 5 cases), 2 (croiseur de 4 cases) et 3 (contre-torpilleur de 3 cases):

- il y a 120 façons de placer un porte-avion (1) dans une grille initialement vide
- in y a 16800 façons de placer un porte-avion (1) et un croiseur (2) dans une grille initialement vide
- il y a 2304000 façons de placer un porte-avion (1), un croiseur (2) et un contre-torpilleur (3) dans une grille initialement vide

Il est possible de calculer de cette manière le nombre de grilles pour la liste complète de bateau, bien que le calcul prendra beaucoup de temps à s'effectuer.

3. Probabilité de tirer une grille donnée

En considérant toutes les grilles équiprobables au tirage, soit p la probabilité de tirer une grille donnée et n le nombre total de toutes les grilles possibles. Alors $p = 1/n$.

Nous implémentons une fonction qui prend en paramètre une grille, génère des grilles aléatoirement jusqu'à ce que la grille générée soit égale à la grille passée en paramètre et qui renvoie le nombre de grilles générées.

A l'exécution cette fonction prend bien évidemment beaucoup trop de temps à se finir: en effet nous avons vu qu'il y avait $N = (120 \text{ x } 140 \text{ x } 160 \text{ x } 160 \text{ x } 180)^2$ grilles possibles pour 2 joueurs. Notre fonction la probabilité de tirer la bonne grille est donc de $1/N$, et nous n'avons aucune garantie de même au bout de N itérations notre grille sera la bonne: nous ne gardons pas une liste des grilles à ne pas tirer une nouvelle fois !

4. Approximation du nombre total de grilles pour une liste de bateaux

Nous programmons un algorithme qui permet d'**approximer** le nombre total de grilles pour une liste de bateaux. Comme l'on veut une approximation et non un nombre précis, nous nous appuierons sur le même modèle de calcul que dans la la première partie de cette section, ie en ne supprimant pas les cas où l'on a superposition de bateaux sur une case.

Il s'agit, pour chaque bateau de chaque joueur, de calculer le nombre d'emplacements possibles dans la grille, grâce à la formule suivante:

$$(DIMX + DIMY) \text{ x } (DIMX - \texttt{dic\_bateaux[bateau]} + 1)$$

Nous obtenons le résultat suivant pour la liste de bateaux [1,2,3,4,5] (avec 1, 2, 3, 4 et 5 les identifiants des bateaux dans notre dictionnaire `dic_bateaux`): il y a 5 992 989 327 360 000 000 000 configurations possibles, soit $(120 \text{ x } 140 \text{ x } 160 \text{ x } 160 \text{ x } 180)^2$; c'est bien le résultat que nous avons obtenu avec notre calcul à la main dans la partie 1 de cette section, et aussi celui que nous obtenons pour la partie 2; mais le temps d'exécution est bien moindre par rapport à l'algorithme implémenté pour la partie 2: alors que celle-ci devait tester tous les emplacements possibles pour chaque bateau, en créant à chaque fois une copie des grilles intermédiaires, nous n'effectuons ici qu'un simple calcul avec pour seules indications la liste de bateaux et les dimensions de la grille.

5 Modélisation probabiliste du jeu

Pour cette partie nous choisissons d'implémenter plusieurs classes génériques pour mieux modéliser notre jeu:

- la classe `Bataille`: bataille générique, qui contient une grille initialisée aléatoirement, deux joueurs (dont un joueur dit `courant`), leur score (nombre de cases adverses touchées), un booléen `touche` qui vaut `True` si le dernier coup du joueur considéré a été un succès.
  * `joue(self)`: simule une partie de bataille navale entre les 2 joueurs jusqu'à ce que l'un d'entre eux gagne (coups = 17)
  * `next(self,coup)`: met à jour le score du joueur courant s'il a trouvé une position adverse, et change de tour
  * `win(self)`: retourne le joueur gagnant, `None` si la partie n'est pas terminée
  * `reset(self)`: réinitialise la partie
- la classe générique `Joueur`: Classe de joueur générique, contenant au moins un attribut `coups` qui est le nombre de coups joués par le joueur, et `actions` qui est une liste de toutes les positions pas encore testées.
  * `get_action(self, grid=None, touche=False)`: choisit la prochaine action (position à choisir) en fonction de l'état de la grille `grid`, et du booléen `touche` qui indique si le coup précédent du joueur était un succès
  * `get_nb_coups(self)`
  * `reset(self)`

5.1 Version aléatoire

La manière la plus simple de procéder dans une partie de bataille navale est de choisir ses coups (ie les cases à tester) de manière *aléatoire*.

Si l'on procède de cette manière, quelle est le nombre moyen de coups joués avant de couler tous les bateaux ?

Soit X la **variable aléatoire** du *nombre de coups qu'un joueur J doit jouer pour gagner*. Pour connaître le nombre moyen de coups à jouer pour gagner, on fait un calcul d'espérance:

Formule de l'espérance

$$E(X) = \sum_{i=1}^{n} x_i \text{ x } P(X = x_i)$$

Ici: $E(X) = \sum_{i=1}^{100} i \text{ x } P(X = i)$

Avec, pour tout i,

$$P(X = i) = \frac{\binom{i-1}{16}}{\binom{100}{17}}$$

car la dernière case tirée en cas de victoire est forcément noire.

En implémentant une fonction permettant d'effectuer ce calcul, nous trouvons qu'il nous faudrait environ 95.38888888888889 coups pour un joueur procédant de manière aléatoire pour gagner.

Implémentons maintenant une sous-classe de `Joueur`, que l'on nommera `Joueur_aleatoire`, et qui sera capable de jouer à la bataille navale en appliquant ce procédé de prise de décisions aléatoires. Pour cela chaque `Joueur_aleatoire` gardera en mémoire une liste de toutes les positions qu'il n'a pas encore jouées, et choisira de manière aléatoire (librairie `random` de Python) une nouvelle position (coup)

Pour vérifier si le nombre moyen de coups nécessaires à un `Joueur_aleatoire` pour gagner correspond bien à celui que nous avons trouvé théoriquement avec le calcul de l'espérance, nous implémentons une fonction `distribution(joueur_1, joueur_2, nb_tests)` qui simule `nb_tests` parties entre `joueur_1` et `joueur_2` et retourne le nombre de coups qu'il a fallu au gagnant pour remporter la partie.

Nous traçons le graphe montrant le nombre de coups moyens à jouer pour qu'un `Joueur_aleatoire` remporte la partie en fonction du nombre de tests (nombre de tests allant de 10 à 1000, par intervalle de 10):

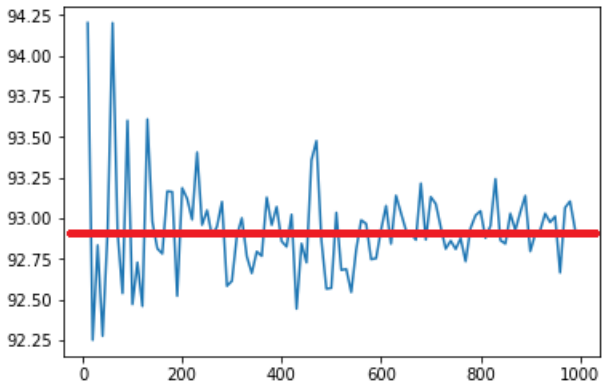

Figure 2: Distribution du nombre de coups avant une victoire pour un `Joueur_aleatoire` en fonction du nombre de parties

Il semblerait d'après le graphique que le nombre de coups nécessaires à un `Joueur_aleatoire` pour remporter la victoire soit légèrement inférieur à 93, ce qui est moins que ce que l'on avait prévu avec le calcul de l'espérance (qui estimait le nombre de coups moyen à 95.38888888888889). Cela s'explique par le fait que notre calcul de l'espérance s'est fait dans l'hypothèse d'une indépendance entre les positions des bateaux: on ne prenait pas en compte les bateaux potentiellement placés dans la grille pour notre calcul, d'où une estimation supérieure à la réalité.

5.2 Version heuristique

La version aléatoire n'exploite pas les coups victorieux précédents, elle continue à explorer aléatoirement tout l'échiquier. Nous nous proposons d'implémenter une version *heuristique* de `Joueur`, que l'on appellera `Joueur_heuristique`. Là aussi le joueur a une liste `actions` des positions qu'il n'a pas encore jouées, mais cette fois-ci il prend aussi une liste `entourage` des voisins de la dernière case touchée qui dont le contenu est le même (une case contient la clé de son bateau dans `dic_bateaux`). Nous définissons ainsi 2 règles dans le choix d'une position:

- tant que le `Joueur_heuristique` n'a rien touché, il choisira ses cases de manière aléatoire, comme un `Joueur_aleatoire`
- une fois qu'il touche un bateau adverse, il va ajouter à `entourage` les cases connexes dont le contenu est le même que le sien. Ainsi tant qu'`entourage` n'est pas vide, `Joueur_heuristique` explorera en priorité les positions retenues dans `entourage`. Une fois `entourage` vide, on repart sur une prise de décision aléatoire.

Nous faisons jouer l'un contre l'autre un `Joueur_aleatoire` et un `Joueur_heuristique` traçons le graphe montrant le nombre de coups moyens à jouer que le gagnant remporte la partie en fonction du nombre de tests (nombre de tests allant de 10 à 1000, par intervalle de 10), ainsi que le nombre de parties remportées par chaque joueur):

```
Distribution heuristique:

Nombre de victoires par joueur: {1: 683, -1: 48817}

------- Nombre de coups moyens pour gagner -------
```

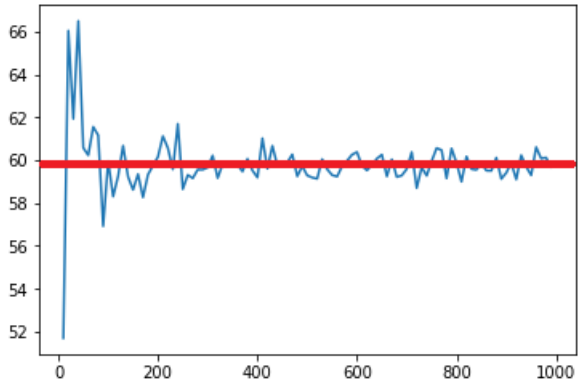

Figure 3: Distribution du nombre de coups avant une victoire pour un `Joueur_aleatoire` (score 1) et un `Joueur_heuristique` (score -1) en fonction du nombre de parties

D'après le graphe ci-dessus, il est clair que le `Joueur_heuristique` le remporte souvent contre le `Joueur_aleatoire`, à 48817 victoires sur 49 500 parties contre 683 victoires pour `Joueur_aleatoire`. `Joueur_heuristique` a ainsi un taux de victoire de plus de 98,6% lorsqu'il joue contre `Joueur_aleatoire`, avec un nombre moyen de coups aux alentours de 60 !

Cela s'explique par l'exploration des cases connexes à une case qui a été touchée par le `Joueur_heuristique`, et qui présentent alors une plus grande probabilité d'abriter elles aussi un bateau (le même que la case touchée). Cette propriété intéressante n'avait pas été exploitée par le `Joueur_aleatoire`.

5.3 Version probabiliste simplifiée

La version précédente ne prend pas en compte le type de bateaux restant ni si le positionnement du bateau est admissible : or, certaines positions ne seront pas possibles en fonction de la localisation de la case touchée (par exemple à coté des bords de la grille ou si un autre bateau a déjà été touché dans la région connexe). Chaque coup nous renseigne sur une position impossible (ou possible d'un bateau). Il est bien sûr hors de questions de calculer à chaque tour la nouvelle distribution jointe sur tous les bateaux en entier.

Nous décidons donc de procéder de la manière suivante:

- au lieu de choisir les cases de manière aléatoire en attendant de toucher une case comme le ferait `Joueur_heuristique`, `Joueur_probabiliste_simplifie` va calculer pour chaque case de la grille actuelle la probabilité qu'elle contienne un bateau (tout bateau confondu), et choisira la case de plus grande probabilité. Pour calculer ces probabilités, il suffit simplement pour chaque bateau qui n'a pas encore été coulé d'incrémenter le compteur de chaque case de la grille s'il est possible pour le bateau d'être placé à cet endroit.
- une fois que l'on réussit à toucher et tant que l'`entourage` n'est pas vide, on procède comme un `Joueur_heuristique` en choisissant en priorité les cases connexes aux cases touchées.

En calculant ainsi nos probabilités, on fait l'hypothèse que la position des bateaux est indépendante, ce qui est faux puisque placer un bateau à une certaine position réduit potentiellement le nombre de positions possibles pour un autre bateau. Cette manière de procéder repose ainsi sur un aspect probabiliste qui ne respecte pas complètement la réalité de notre modélisation. Elle n'en reste néanmoins pas moins efficace que la stratégie heuristique, comme nous pouvons l'observer dans le graphe ci-dessous:

```
Distribution probabiliste simplifié:

Nombre de victoires par joueur: {1: 162, -1: 12088}

------- Nombre de coups moyens pour gagner -------
```

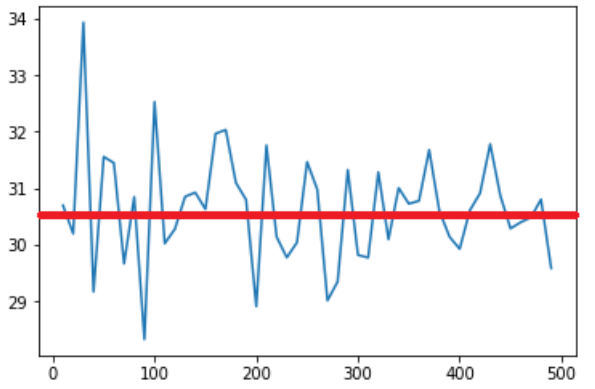

Figure 4: Distribution du nombre de coups avant une victoire pour un `Joueur_heuristique` (score 1) et un `Joueur_probabiliste_simplifie` (score -1) en fonction du nombre de parties

Dans le graphe ci-dessus faisons jouer l'un contre l'autre un `Joueur_heuristique` et un `Joueur_probabiliste_simplifie`. Nous avons tracé le graphe montrant le nombre de coups moyens à jouer que le gagnant remporte la partie en fonction du nombre de tests (nombre de tests allant de 10 à 500, par intervalle de 10), ainsi que le nombre de parties remportées par chaque joueur).

Nous observons clairement que le `Joueur_probabiliste_simplifie` le remporte très souvent contre le `Joueur_heuristique`, à 12 088 victoires sur 12 250 parties contre 162 victoires pour `Joueur_heuristique`. `Joueur_probabiliste_simplifie` a ainsi un taux de victoire de presque 98,7% lorsqu'il joue contre `Joueur_heuristique`, avec un nombre moyen de coups aux alentours de 30, ce qui est la moitié dnu nombre de coups dont avait besoin `Joueur_heuristique` pour vaincre `Joueur_aleatoire` !

Cela s'explique par le fait que, bien que les cases connexes aux cases déjà touchées sont prioritaires dans les deux implémentations, `Joueur_probabiliste_simplifie` gagner en efficacité dans les cas où il n'y a plus de case connexe à explorer: alors que `Joueur_heuristique` se contentait de tirer de nouveau une case au hasard, `Joueur_probabiliste_simplifie` choisit de mettre toutes les chances de son côté en optant pour les cases qui ont potentiellement le plus de chance d'abriter un bateau: il s'agit, de manière générale, des cases les plus au centre de la grille, car ce sont elles qui pourront accueillir à la fois les bateaux

les plus longs mais aussi les plus petits. Cette propriété avait été négligée par le `Joueur_heuristique`, ce qui lui vaut d'être aussi peu efficace contre `Joueur_probabiliste_simplifie` !

6 Conclusion

Nous avons donc pu montrer au travers de ce projet que le choix d'implémentation d'une stratégie pour les jeux dits *de hasard* était particulièrement important pour améliorer les performances de notre joueur ! Bien que l'on puisse très bien se contenter de compter sur le hasard pour le choix de nos cases, comme le fait `Joueur_aleatoire`, une simple analyse des cases adjacentes aux cases déjà touchées réduit de 1/3 le nombre moyen de coups nécessaires à la victoire et ramène les chances de remporter la partie à plus de 98%. Il en est de même pour `Joueur_probabiliste_simplifie`: on aurait pu se contenter des performances déjà satisfaisantes de `Joueur_heuristique`, mais le simple fait de ne pas compter sur l'aléatoire pour choisir ses cases (dans le cas où il n'y a plus de case adjacente à explorer) et de choisir les cases les plus à même d'héberger un bateau a là encore réduit de moitié le nombre moyen de coups nécessaires à la victoire contre un `Joueur_heuristique`, et ramène les chances de remporter la partie à plus de 98% !

Il est donc vital, dans toute modélisation, de décrire au mieux les variables qui la caractérisent et ainsi trouver les meilleures façons de les implémenter: un simple détail pourrait suffire à augmenter grandement son efficacité !