Statistiques et Informatique
Projet 1: bataille navale

Ekaterina BOGUSH 21214957
Amelie CHU 21206329

Sommaire

1 Introduction

Dans ce projet notre but est d'étudier le jeu **Bataille navale** en appliquant des méthodes combinatoires et probabilistes. Le projet est composé de plusieurs parties :

- Étude du nombre de configurations de la grille
- Implémentation de différentes stratégies de jeu
- Recherche d'un sous-marin perdu (non liée aux parties précédentes)

On ajoute le lien sur GitHub du projet en cas des problèmes d'affichage des commentaires à cause des accents.

2 Organisation du code

On a organisé le projet en plusieurs fichiers avec des classes :

- Fichier **grille.py** contient la classe **Grille**. Cette classe contient toutes les fonctionnalités pour la gestion de la grille. Ainsi, on a mis dans cette classe les fonctions utiles pour les questions combinatoires.
- Fichier **bataille.py** contient la classe **Bataille**. Cette classe implémente les fonctions auxiliaires pour la simulation du jeu.
- Fichier **joueur.py** contient la classe **Joueur**. Cette classe implémente la fonctionnalité de jouer au jeu Bataille navale.
- Fichier **constants.py** qui contient uniquement des constante utiles pour le jeu.
- Fichier **ObjPerdu.py** qui contient la Classe **ObjPerdu**. Cette classe implémente la recherche du sous marin perdu dans une grille avec probabilité.
- le dossier **/data** contient les codes de tests et les résultats d'expériences

Bibliothèques Python utilisées

- **Numpy** pour gérer la grille de manière plus efficace par rapport aux simples listes de Python.
- **Matplotib** pour dessiner la grille et des graphiques qu'on présente en cours de ce rapport.
- **Pandas** pour sauvegarder les données de la distribution de la variable aléatoire correspondant au nombre de coups nécessaire pour gagner le jeu.

On recommande d'utiliser un environnement virtuel en raison de l'utilisation de plusieurs bibliothèques Python.

3 Combinatoire de jeu

On s'intéresse à trouver le **nombre de configurations** possibles de la grille de taille $n \times n$, noté N_c qu'on pourrait faire dans le jeu de Bataille Navale avec la liste de 5 bateaux. Une configuration de la grille correspond aux **positions** de tous les bateaux.

Tout d'abord on calcule la borne supérieure de N_c. Pour simplifier les calculs on fait ***une hypothèse fausse*** :

> Les bateaux peuvent se superposer.

Dans ce cas pour chaque bateau, les **positions possibles** (verticale et horizontale) sont calculées comme si la grille était vide. Pour chaque bateau sur la grille de taille $n \times n$ le nombre de configurations possible est donnée par la formule suivante :

$$n \cdot 2 \cdot (n - (taille\_bateau) + 1)$$

En utilisant cette formule on calcule le nombre de positions possibles pour chaque type de bateau (ici $n = 10$):

Type du bateau	**Taille du bateau**	**Nombre de positions possibles**
Porte-avion	5	$10 \times 2 \times 6 = 120$
Croiseur	4	$10 \times 2 \times 7 = 140$
Contre-torpilleurs	3	$10 \times 2 \times 8 = 160$
Sous-marin	3	$10 \times 2 \times 8 = 160$
Torpilleur	2	$10 \times 2 \times 9 = 180$

La borne supérieure des configurations possibles des bateaux (un de chaque type) sur la grille de taille 10×10 est donc égale à

$$120 \times 140 \times 160 \times 160 \times 180 \approx 8 \times 10^{10}$$

On ajoute la fonction **Grille.calc_nb_placements_bateau** qui utilise la formule donnée précédemment pour calculer le nombre de placements possibles du bateau en fonction de la taille de la grille et la taille du bateau. En testant la fonction sur les bateaux de chaque type, on obtient bien nos résultats théoriques précédents.

Maintenant on s'intéresse à approximer le nombre de placements possibles des bateaux. On utilise l'algorithme du type *recherche par force brute* (la fonction **Grille.calc_nb_placements_liste_bateaux).** L'algorithme calcule toutes les configurations en posant les bateaux dans toutes les positions possibles les uns après les autres sans superpositions.

Ci-dessous on présente quelques résultats de la fonction pour une grille de taille 10×10.

Nombre de bateaux	**Liste des bateaux**	**Nombre de configurations possibles**
1	Porte avion	120
1	Croiseur	160
2	Porte avion, croiseur	14 400
2	Croiseur, sous-marin	20 336
2	Torpilleur, sous-marin	27 336
3	Porte avion, croiseur, torpilleur	2 216 256
3	Torpilleur, sous-marin, contre-torpilleurs	3 848 040

Cependant, les calculs prennent trop de temps si le nombre de bateaux ≥ 4, d'où l'absence de tests pour ces valeurs.

Analyse de complexité

Soit une grille de la taille $n \times n$ et B le nombre de bateaux à placer. La complexité de l'algorithme **brute force** est en $O(n^{2(B+1)})$.

On considère que les tirages des configurations de la grille de taille $n \times n$ sont **équiprobables**. Soit la variable aléatoire G qui prend ses valeurs dans toutes les configurations de grilles possible. N_c est le nombre de configurations totale de la grille de taille $n \times n$ pour la liste de 5 bateaux. La probabilité de tirer la configuration spécifique g de la grille est

$$P(G = g) = \frac{1}{N_c}$$

On remarque que pour N_c grand, la probabilité d'obtenir une grille donnée est très faible. Par exemple, pour 5 bateaux sur la grille de la taille 10×10, $N_c \approx 8 \times 10^{10}$, c'est pour cette raison qu'on effectue des tests sur des grilles de taille plus petites.

Notre fonction **Grille.generer_meme_grille** reçoit une grille en entrée. Elle fait le tirage d'une grille de même taille avec les 5 bateaux placés, de manière aléatoire et s'arrête dès qu'elle trouve la même configuration que celle d'entrée.

Les tests sont réalisés sur les grilles de taille 5 (taille minimum) et de taille 6. Des grilles de plus grandes tailles prennent beaucoup trop de temps à calculer.

Taille de la grille	**Nombre moyen de grilles générées**	N_c
$5 * 5$	36 951	80 848
$6 * 6$	2 803 325	6 687 136

Le **nombre moyen de grilles** générées avant d'obtenir une grille donnée est calculé pour 50 appels de fonction. Ce nombre nous donne une indication sur N_c le nombre total de grilles. La moyenne du nombre des tirages aléatoires réalisés nous montre en moyenne combien de grilles différentes nous obtenons. On peut considérer ce nombre comme une **borne inférieure** de N_c comme constaté sur le tableau ci-dessus.

Toutefois, le tirage aléatoire pour des grilles de taille 10×10 est très lent et très lourd en termes de calculs, et la borne inférieure peut être très **éloignée** du nombre N_c recherché.

Pour approximer le plus justement N_c , une solution est de calculer le **nombre de configurations** avec des bateaux qui se superposent, puis de soustraire la **borne supérieure** par ce nombre. On obtient ainsi le nombre de grilles avec les 5 bateaux de tel sorte que les bateaux ne se superposent pas: il s'agit des configurations recherchées.

4 Modélisation probabiliste du jeu

On va maintenant s'intéresser au jeu et au **nombre de coups joués** dans une partie avant de gagner. Un tirage correspond au choix d'une case à jouer. On implémente plusieurs **stratégies**.

4.1 Version aléatoire

Hypothèses :

- Le tirage des cases est équiprobable, i.e. on ne prend pas en compte si un bateau est touché ou non lors du coup précédent, il n'y a pas de stratégie de jeu.
- On ne peut pas jouer sur une case déjà jouée

Soit X la **variable aléatoire** qui représente le nombre de coups joués avant de couler tous les bateaux.

La **probabilité** de couler les bateaux en i coups avec $i \in [\![17, 100]\!]$ est

$$P(X = i) = \frac{C_{i-1}^{b-1}}{C_n^b}$$

- b: le nombre de cases bateaux ($b = 17$)
- n: le nombre de total de cases ($10 \times 10 = 100$)
- i: le nombre de coups avant de gagner ($i \geq b$)

Raisonnement derrière la formule:

L'ordre de tirage ainsi que la position $(ligne, colonne)$ des cases n'ont pas d'importance.

On calcule d'abord le nombre de façon de tirer b **cases-bateaux** parmi n cases de la grille, soit C_n^b. Les $n - b$ tirages restants correspondent à des tirage de cases vides. La position des cases ne compte pas, donc on remplie les tirages restants de manière unique.

Ensuite, on compte les cas **favorables** pour $P(X = i)$. Cette probabilité signifie qu'on fait exactement i tirages avant de gagner, tel que le dernier (i^{eme}) tirage corresponde à une case-bateau. On fixe le i^{eme} tirage par une case-bateau, il reste donc $i - 1$ tirages et $b - 1$ cases bateaux. Le nombre de combinaisons de b-1 cases-bateaux parmi i-1 cases est C_{i-1}^{b-1}.

Calcul de l'espérance:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i=b}^{n} i \cdot P(X = i) \\
&= \frac{1}{C_n^b} \cdot \sum_{i=b}^{n} i \cdot C_{i-1}^{b-1} \\
&= \frac{1}{C_n^b} \cdot \sum_{i=b}^{n} b \cdot C_i^b \\
&= b \cdot \frac{1}{C_n^b} \cdot \sum_{i=b}^{n} C_i^b
\end{aligned}$$

Pour simplifier la formule on utilise la **Formule d'itération de Pascal** $\sum_{k=p}^{n} C_k^p = C_{n+1}^{p+1}$ et l'on obtient:

$$\mathbb{E}[X] = \frac{b \cdot C_{n+1}^{b+1}}{C_n^b}$$

En remplaçant les valeurs $b = 17, n = 100$, on obtient $\mathbb{E}[X] \approx 95,39$ soit 95 coups environ. Pour calculer la distribution de la variable aléatoire on utilise la fonction **Joueur.jouer**.

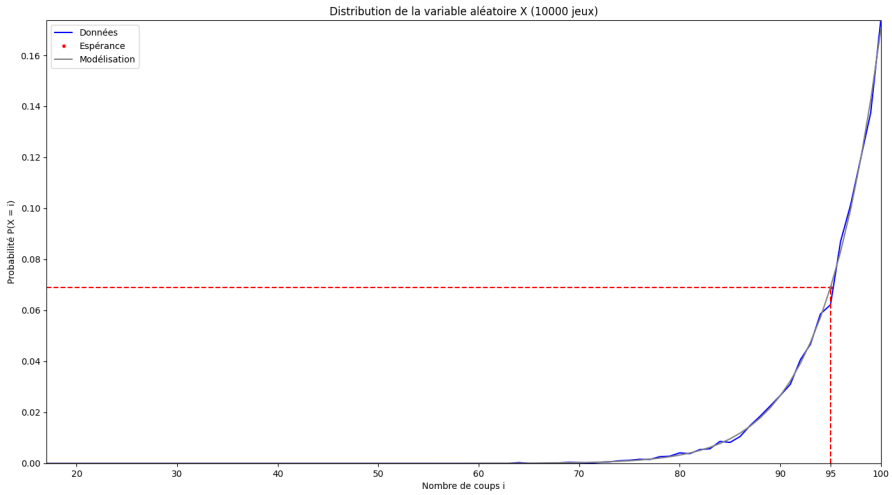

Figure 1: Distribution de la variable aléatoire X (ver. aléatoire)

On joue 10 000 jeux en mémorisant à chaque fois le nombre de coups qu'on a joué avant de gagner. De cette manière, pour chaque $i \in [\![17, 100]\!]$ on aura le nombre de jeux tel qu'il fallait jouer i coups pour couler tous les bateaux (données dans le fichier **data/data.csv**).

La **distribution** de la variable aléatoire X en bleue (*Figure 1*) montre qu'il faut explorer presque toute la grille pour gagner. En effet, on a supposé que le tirage de chaque case est équiprobable (ce qui n'est pas le cas en réalité). De plus, la grille ne contient que 17 cases avec les bateaux pour 83 cases libres. Notre variable X suit bien la modélisation calculée précédemment $P(X = i)$. En moyenne il faut réaliser **95 tirages** pour couler tous les bateaux, ce qui correspond bien à nos résultats théoriques calculés précédemment, et cela se justifie par **l'indépendance** des cases entre elles et la **probabilité uniforme**.

4.2 Version heuristique

Dans cette version, la ***stratégie de jeu*** consiste à explorer les cases connexes lorsqu'un bateau a été touché au coup précédent.

Hypothèses :

- Les cases sont considérées équiprobables lors du 1er tour ou lorsqu'au tour précédent aucun bateau n'a été touché. Si on note N_r le nombre des cases restantes, alors la probabilité de choisir une case équivaut à $\frac{1}{N_r}$.
- Si un bateau a été touché au tour précédent, alors la probabilité de choisir une case non jouée connexe à la précédente au tour suivant vaut 1.

Cette stratégie visite les cases connexes de la case bateau, et si un bateau se trouve sur une de ces cases connexes, on répète l'agorithme sur cette case.

Conjecture

En implémentant une stratégie de jeu **Joueur.jouer_heuristique**, on peut faire l'hypothèse qu'il faut moins de coups à jouer avant de gagner, puisque si l'on touche une case bateau, il y a une chance de toucher une autre **case bateau** au prochain coup. De ce fait, comparé à la première tactique, on peut supposer obtenir plus de jeux gagnés avec moins de coups.

Résultats de l'expérience :

La *Figure 2* permet de comparer les versions **aléatoire** et **heuristique**. La courbe rouge représente le nombre de jeux gagnés en i coups avec la stratégie sur 10 000 jeux et la courbe bleue représente nos résultats précédent de jeux sans stratégie. On observe une **meilleure répartition** de nos jeux sur l'intervalle $[50, 100]$ comparé à nos résultats précédents. Le nombre **nombre moyen** de coups joués avant de gagner pour la version heuristique est d'environ **80** coups (données dans **data/data2.csv**), comparé à 95 pour la version aléatoire. On peut donc dire que le jeu avec stratégie est plus **optimal** que la méthode aléatoire si l'on veut gagner avec le moins de coups possibles.

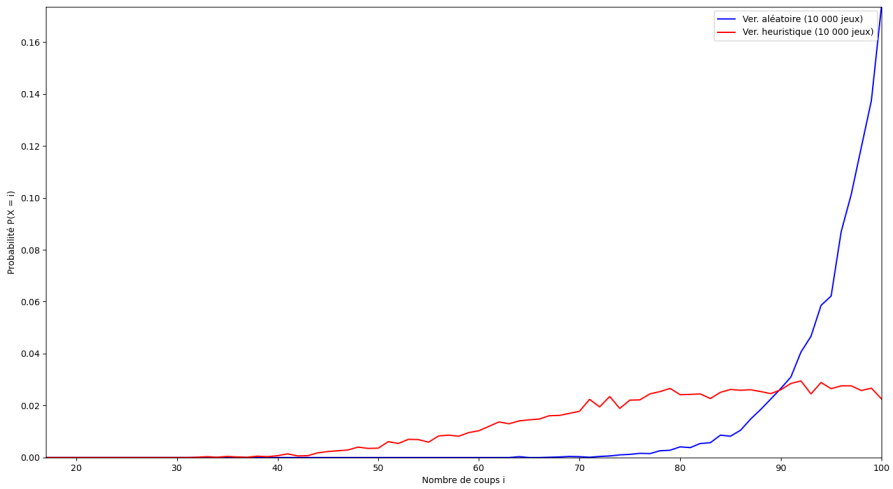

Figure 2: Ver. aléatoire, ver. heuristique

4.3 Version probabiliste simplifiée

Maintenant, on s'intéresse à **améliorer** la stratégie de jeu en prenant en compte la taille des bateaux, les bateaux restants et les probabilités d'avoir un bateau sur une case.

Pour calculer la ***distribution jointe*** de tous les bateaux, il faut prendre en compte toutes les configurations possibles (en plaçant tous les bateaux) de la grille. Une fois la case choisie, il faut recalculer toutes les configurations possibles, mais cette fois en tenant compte du fait que la case choisie contenait ou non un bateau (i.e. que certaines configurations pourraient être éliminées). Cependant, on a vu que pour la grille de taille 10×10 contenant 5 bateaux (un de chaque type) il existe beaucoup trop de possibilités, ce qui rend le recalcul de toutes les configurations possibles à chaque tour très coûteux et donc non envisageable.

Afin de simplifier les calculs, nous sommes obligés de poser l'hypothèse suivante:

Hypothèse :

> Les bateaux ne dépendent pas les uns des autres i.e. leur position est indépendante

Il faut noter que cette ***hypothèse est fausse*** en réalité. Cela vient du fait que les bateaux placés sur la grille ne peuvent pas se superposer. Ainsi, une fois le bateau placé, certaines positions sur la grille sont **éliminées**. Par exemple, en plaçant un bateau de taille 3 horizontalement sur la position $(1, 0)$, il est

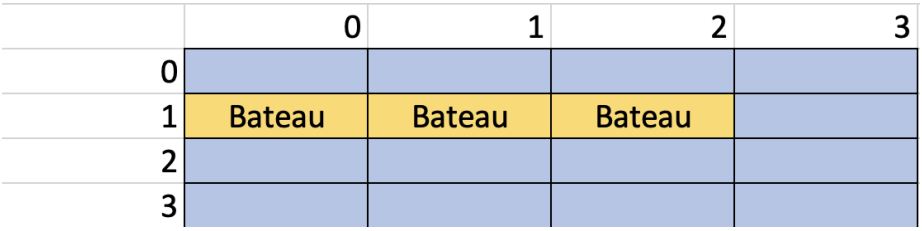

Figure 3: Position du bateau

impossible de placer les autres bateaux verticalement sur les cases $(0,0),(0,1)$ et $(0,2)$, d'où la dépendance entre les placements des bateaux (*Figure 3*).

Résultats de l'expérience :

Conjecture :

Cette stratégie de jeu ressemble à celle de la version heuristique qui explore les cases connexes. La différence se pose lors du choix des cases. La version probabiliste simplifiée prend en compte la probabilité des cases de contenir le bateau, ce qui n'est pas le cas pour la version heuristique (le choix est fait soit de manière aléatoire, soit en explorant les cases connexes de la case précédente). On peut donc faire l'hypothèse que la version probabiliste simplifiée sera plus optimale par rapport à la version heuristique.

L'algorithme de la stratégie de jeu est implémenté dans la fonction **joueur.jouer_proba_simple**.

Dans notre algorithme on utilise une grille supplémentaire pour chaque bateau, nommé ***grille-probabilité***.

Grille-probabilité du bateau b : une grille de même taille que la grille de jeu. Dans chaque case de la grille on stocke le **nombre de configurations** du bateau b qui passent par cette case. On peut donc en déduire que certaines cases ont une probabilité plus élevée d'avoir un bateau b. À chaque tour, on choisit la case ayant le **nombre maximal** (équivaut à choisir la probabilité maximale) parmi toutes les grilles-probabilités restantes. Après chaque choix, une grille est mise à jour en recalculant toutes les configurations possibles. Ensuite, chaque case de la grille est mise à jour de la manière suivante (pseudo-code):

Algorithm 1 Algorithme de mise à jour de la grille-probabilité

```
Pour chaque position du bateau faire
    nb_touches ← nb de cases BAT_TOUCHE occupées par position
    Pour chaque case de position faire
        case ← case + nb_touches
```

En effectuant une telle mise à jour, on priorise les cases autour de celles qui contiennent les bateaux. Si le bateau est coulé, alors la grille-probabilité associée

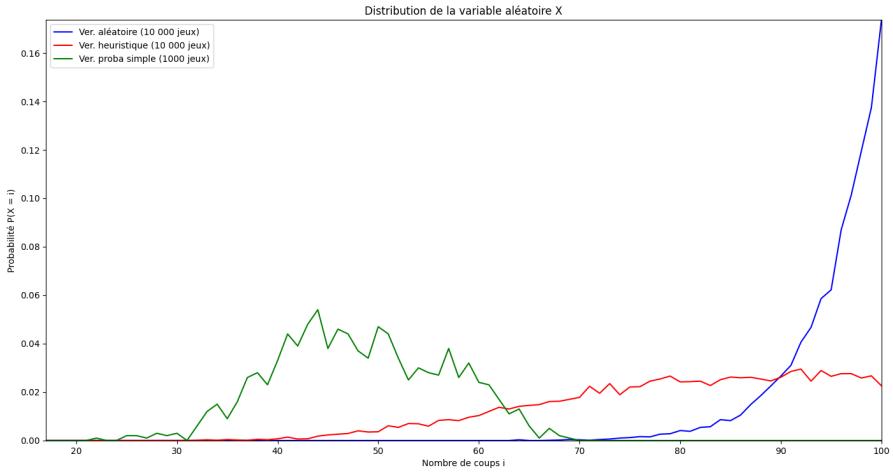

Figure 4: Ver. alétoire, ver.heuristique, ver. probabiliste simplifiée

à ce bateau est également supprimée.

Remarque : Pour la version probabiliste simplifiée on a réduit le nombre de jeux à 1000. L'algorithme est beaucoup plus lourd par rapport aux 2 autres versions. Cela vient du fait qu'à chaque tour de boucle on recalcule toutes les configurations possibles de la grille (complexité de $O(n^2)$ pour la grille de taille $n \times n$) en prenant en compte l'information sur la case choisie (si elle contient ou non le bateau).

La *Figure 4* permet de comparer les 3 stratégies pour jouer à la bataille navale. La courbe bleue représente la version aléatoire, la courbe rouge représente la version heurisitique et la courbe verte représente la version probabiliste simplifiée. On observe que dans la version probabiliste simplifiée, le nombre de coups nécessaires pour gagner le jeu **diminue** de manière significative. Les probabilités les plus élevées pour cette version se situent dans l'intervalle $[30, 70]$. En moyenne, il faut jouer environ **48** coups avant de gagner (données dans **data/data3.csv**), ce qui est bien plus optimal que les deux autres stratégies. On remarque que dans les versions aléatoire et heuristique, les probabilités de gagner au jeu avec un tel nombre de coups sont très faibles, voire quasi inexistantes.

5 Senseur imparfait : à la recherche du sous-marin

Dans cette partie on s'intéresse à **localiser** un sous-marin perdu. Pour cela on utilise un senseur imparfait. Si l'objet se trouve dans la région, le senseur le détectera avec une probabilité p_s. Cependant, si aucun objet est présent dans la région, alors le senseur ne détecte rien.

Hypothèse :

> Dans la région de recherche il n'y a qu'un seul sous-marin occupant une case. Il n'y a pas d'autres objets présents sous-l'eau.

On divise la section de recherche sur les cases en les numérotant de 1 à N. Pour chaque case i on pose :

- la probablité π_i de contenir le bateau. On note que $\sum_{i=1}^{N} \pi_i = 1$
- la variable aléatoire $Y_i \in \{0, 1\}$ qui représente la position de sous-marin. $Y_i = 1$ si la case i contient le sous-marin, $Y_i = 0$ sinon.
- la variable aléatoire $Z_i \in \{0, 1\}$ qui représente le résultat de recherche par le senseur. $Z_i = 1$ si le senseur a détecté le sous-marin dans la case i, $Z_i = 0$ sinon.

Soit $i \in [\![1, N]\!]$.

Loi de la variable aléatoire Y_i :

$$P(Y_i = 0) = 1 - \pi_i$$
$$P(Y_i = 1) = \pi_i$$

Loi de la variable aléatoire $Z_i|Y_i$:

$$P(Z_i = 1, Y_i = 1) = p_s$$
$$P(Z_i = 0, Y_i = 1) = 1 - p_s$$
$$P(Z_i = 1, Y_i = 0) = 0$$
$$P(Z_i = 0, Y_i = 0) = 1$$

Puisque le senseur est imparfait, il est également nécessaire de connaître la probabilité que la case contienne le sous-marin, même si le senseur ne détecte

rien.

$$P(Y_k = 1|Z_k = 0) = \frac{P(Z_k = 0|Y_k = 1) \ \cdot \ P(Y_k = 1)}{P(Z_k = 0)}$$

$$= \frac{P(Z_k = 0|Y_k = 1) \ \cdot \ P(Y_k = 1)}{P(Z_k = 0 \cap Y_k = 1) + P(Z_k = 0 \cap Y_k = 0)}$$

$$= \frac{P(Z_k = 0|Y_k = 1) \ \cdot \ P(Y_k = 1)}{P(Z_k = 0|Y_k = 1)P(Y_k = 1) + P(Z_k = 0|Y_k = 0)P(Y_k = 0)}$$

$$= \frac{(1 - p_s) \cdot \pi_k}{(1 - p_s) \cdot \pi_k + 1 \cdot (1 - \pi_k)}$$

$$= \frac{(1 - p_s) \cdot \pi_k}{1 - p_s \cdot \pi_k}$$

$$= \frac{(1 - p_s) \cdot (1 - \sum_{i \neq k} \pi_i)}{1 - p_s \cdot (1 - \sum_{i \neq k} \pi_i)}$$

Une fois la recherche de la case effectuée, on doit augmenter ou diminuer les probabilités des autres cases. Pour cela, on met à jour la probabilité de π_k et on **redistribue** le reste aux autres cases de manière uniforme :

$$A = P(Y_k = 1|Z_k = 0); \ \pi_k = A; \ \forall i \neq k, \ \pi_i = \pi_i + \frac{\pi_k - A}{N - 1}$$

Algorithm 2 Algorithme de recherche de l'objet perdu

1: $iteration \leftarrow 1$
2: **Tant que** objet pas trouvé **faire**
3: Chercher sur la case k avec la plus grande probabilité
4: **Si** senseur renvoie 1 **alors**
5: Renvoyer iteration
6: **Sinon**
7: $\pi_k \leftarrow A$
8: pour toute case $i \neq k, \ \pi_i = \pi_i + \frac{\pi_k - A}{N - 1}$
9: $iteration \leftarrow iteration + 1$

L'algorithme est implémenté dans le fichier **ObjPerdu** avec la fonction **rechercher** qui va simuler la recherche du senseur. On réalise 1 000 tests sur une grille de taille 20x20, pour une répartition π_i uniforme, élevée au centre puis aux bords de la grille, avec la probabilité p_s variant entre 0.1 et 0.9

On calcule ensuite le nombre moyen de recherches pour chaque type de tests:

p_s	Uniforme	Centre	Bords
0.1	3 397	3 643	3 590
0.3	960	923	971
0.5	395	423	403
0.7	163	183	189
0.9	56	44	46

On observe que le nombre de recherches est **semblable** peu importe la répartition de π_i sur la grille. Cela peut s'expliquer par le fait qu'en visitant et mettant à jour les cases ayant la probabilité π_i les plus élevées, les π_i vont être modifié de sorte qu'après un certain nombre de recherches, la distributivité de π_i va tendre vers la **répartition uniforme**.

En conclusion, la recherche de l'objet perdu dépend beaucoup plus de la **probabilité** p_s que le senseur détecte l'objet, que de la **répartition** des π_i sur la grille.