Rapport du Projet 1 : Bataille Navale

ZHOU Runlin 28717281

MA Peiran 28717249

Introduction :

*Ce projet a comme objectif d'étudier le jeu de la "Bataille Navale" d'un point de vue probabiliste. Le but de ce jeu est de couler tous les bateaux(touché tous les point de bateau) sur la grille 10*10 en un minimum de coups.*

On divise ce projet à 4 partie pour trouver des moyens d'amener les joueurs à la victoire plus rapidement

- **Partie 1:** *Initaliser un jeu (la création d'un jeu / les positions des bateaux)*
- **Partie 2:** *Nombre de la grille possible pour trouver les combinaisons différentes de jeu*
- **Partie 3:** *Exploration du nombre d'étapes requises pour des stratégies différentes*
- **Partie 4:** *Algorithme pour rechercher l'objet perdu*

Présentation du répertoire :

Les codes sont stockés dans des sous-dossiers (sauf que "figure"), au nombre de quatre:

- esperance : les codes pour calculer l'espérance de l'histogramme de fréquence
- model : la création des classes (grille, bateau, bataille). Cela nous permet de générer directement des objets de ces classes à manipuler dans le code suivant
- stratege : 4 versions de stratégie, principalement appliquées à Partie 3 dans le projet
- unknown

Le sous-dossier "figure" sont destinés aux graphiques requis dans le rapport, qui est le résultat de notre analyse des données expérimentales.

- les histogramme de frequence et les traces de fonction

Tous les fichiers commençant par "test_" sont utilisés pour vérifier la fiabilité du programme pour chaque partie du projet. Notez que l'exécution du programme peut prendre un certain temps, probablement 1 à 2 heures.

Partie 1:

Dans cette section, nous utiliserons souvent les variables suivantes

- **grille:** *Zone de jeu, tous les navires doivent se trouver dans cette zone. On l'initialise comme une matrice 10*10 vide.*
- **bateau:** *Il existe cinq types au total. Chaque bateau sera codé par un identifiant entier: 1 pour le porte-avions, 2 pour le croiseur, 3 pour le contre-torpilleurs, 4 pour le sous-marin, 5 pour le torpilleur. Et chaque type de bateau a une taille correspondant.*
- **position:** *La tête d'un certain bateau. C'est une coordonnée constituée de deux entiers.*
- **direction:** *Un entier qui représente la direction d'un bateau(1 pour horizontale et 2 pour verticale)*

Exemple d'un test de programme :

Les nuances de couleur indiquent le type de navire, le noir pour les points vides.

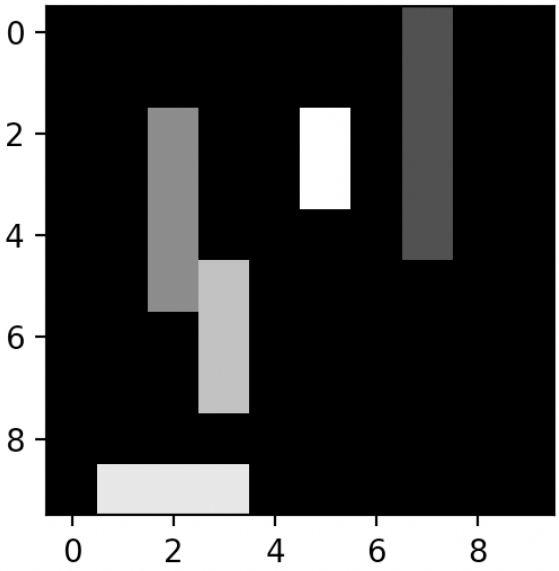


figure 1.1

Partie 2:

Borne supérieure simple du nombre de configurations possibles:

Nous savons que si nous pouvons insérer un bateau plus grand dans la grille, il y aura plusieur placements pour un autre bateau plus petit dans cet espace. Par exemple, présenté dans la figure 2.1.

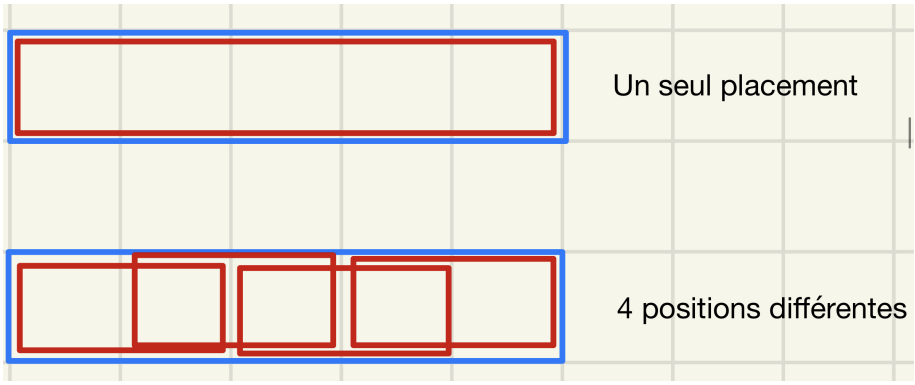


figure 2.1

Par conséquent, selon la numérotation dans Partie 1, nous peut déduire que le nombre de configurations possibles pour la liste de bateau “$ls1$” {5, 4, 3, 3, 2} est plus grand que le nombre de la liste “$ls2$” {5, 5, 5, 5, 5}, et plus petit que le nombre de la liste “$ls3$” {2, 2, 2, 2, 2}

Pour le placement de $ls2$ et $ls3$, nous pouvons simplement penser à choisir au hasard 5 des emplacements possibles où un navire de ce type peut être placé sur la grille 10*10.

donc, on a:

- $nb_{ls2} = C^5_{nbPlacer\_taille5}$
- $nb_{ls3} = C^5_{nbPlacer\_taille2}$
- et pour $ls1$, on a $nb_{ls2} < nb_{ls1} < nb_{ls3}$

Nombre de façons de placer un bateau donné sur une grille vide:

Pour résoudre ce problème, nous allons créer une nouvelle fonction *count_possible()*.

Supposons que la taille de grille est n*m et la taille de bateau est k. Selon la méthode de placement présenté dans figure 2.1, on a

- $n - k + 1$ positions différentes pour chaque colonne
- et $m - k + 1$ positions pour chaque rang
- donc, en total:

$$nbPlacer\_taille(k) = m(n - k + 1) + n(m - k + 1) = 2mn - (m + n)(k - 1)$$

Après calculer, le résultat est: $C^5_{120} < nb_{ls1} < C^5_{180}$, donc l'ordre de grandeur de $ls1$ est 10^{20}, ce qui preuve que c’est impossible pour dénombrer tous les possibilités de placement.

Autre solution possible pour dénombrer tous les cas:

> 💡 En informatique, un **algorithme de Las Vegas** est un type d'algorithme probabiliste qui donne toujours un résultat correct ; son caractère aléatoire lui donne de meilleures performances temporelles en moyenne2.

Selon ma compréhension de l'algorithme de Las Vegas, ***"Plus l'expérience est répétée, plus la probabilité que nous trouver la solution optimale augmente".***

Par conséquent, nous proposons l'algorithme suivant:

```
generer_grille_equiproque(grille):
    g0 = Grille()      #la création d'une nouvelle grille
    count = 0          #le compteur qui peut stocker le nombre de grille différent
    while g0 n'est pas égale à grille:
        g0 = Grille()  #re-faire une autre fois
        count+=1
    return count

"""
    Dans ce fonction, chaque boucle qui vérifier la condition du "while" peut
produit une grille qui est différent de la grille originale, donc le compteur peut
se augement à 1 dans chaque boucle.
"""
```

La valeur de compteur représentent uniquement les placements de bateaux différents par cette exécution du programme. Chaque résultat représente au moins n placement différentes, et nous pouvons également supposer que chaque résultat est correct.

Selon l'algorithme de Monte Carlo, si le nombre d'exécutions est suffisamment grand, nous pouvons supposer que le plus grand nombre sur la liste de résultat est le plus proche de la valeur réelle$\left(nb_{ls1}\right)$, c'est-à-dire la valeur approximative.

Partie 3:

Dans cette section, nous commençons par créer une classe ***"Bataille",*** avec trois fonctions :

- *joue(self, position)* : tenter de toucher un bateau
- *victoire(self)* : vérifier si tous les point de chaque bateau sont touchés
- *reset(self)* : Redémarrer un jeu

Nous construisons deux matrices vide ***grille*** et ***record.***

- ***Grille*** stocke tous les information de bateau
- ***Record*** enregistre les coordonnées de tous les hits, le point touché deviendra changer à 1.

Nous comptons donc simplement le nombre de points qui passent à 1 dans ***victoire(self)***, et si nous atteignons 17, alors nous gagnons la bataille.

Nous commençons à présenter successivement les algorithmes et l'analyse des différentes stratégies:

Version aléatoire:

Cette stratégie consiste à frapper les coordonnées au hasard (sans les répéter).

Supposons que n actions permettent de gagner la bataille, alors cela revient à prendre n cases sur 100 et que les coordonnées des positions des bateaux sont dans la liste de n cases, car $17 \leq n \leq 100$.

On peut dériver la probabilité que le jeu peut fini dans ***n*** actions:

$$P(n) = \frac{C_{n-17}^{(100-17)}}{C_{100}^{n}}$$

A l'aide d'un ordinateur, on peut réussir à déduire la valeur de la probabilité d'occurrence de l'événement correspondant à différentes valeurs de n. Et on peut obtenir le graphe suivant (figure 3.1):

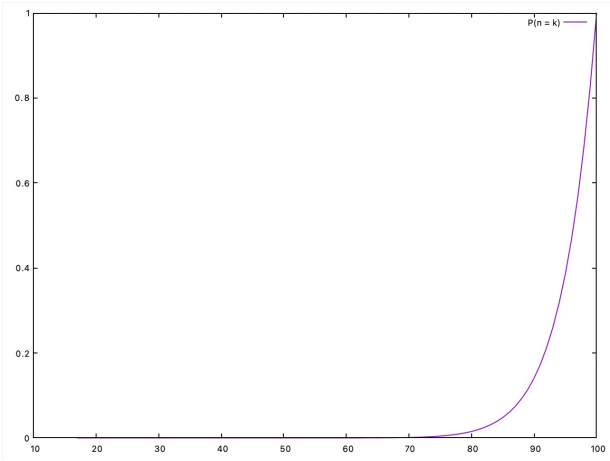


figure 3.1

donc l'espérance de nombre d'action est :

$$E(n) = \sum n * P(n) = 100.0000$$

Enfin, nous avons répété l'expérience 1000 fois et compté la fréquence des n actions gagnées, tout en traçant un histogramme de fréquence (figure 3.2).

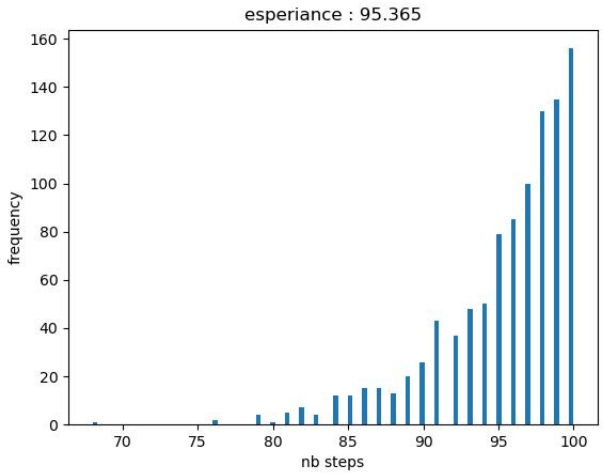


figure 3.2

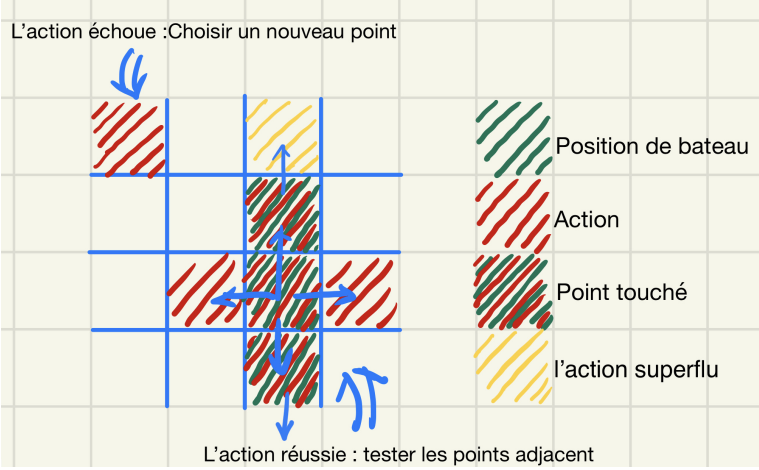
Nous avons comparé les résultats graphiques avec les prévisions que nous avions calculées et la réponse était conforme aux prévisions.

Version heuristique:

Dans le cadre de cette stratégie, nous sélectionnons toujours les points cibles de manière aléatoire comme précédemment, mais nous les divisons en deux cas différents：

- Si cette opération échoue, les nouvelles coordonnées sont à nouveau choisies au hasard.
- S'il réussit, les 4 points autour du point d'impact sont sélectionnés à tour de rôle et la direction du navire est jugée. Après cela, nous continuerons dans la direction du navire jusqu'à ce qu'il soit coulé.

Dans le shéma suivant (figure 3.3), nous pouvons comprendre la stratégie de manière plus intuitive :



> 💡 Il y a des actions inutiles pour cette méthode (marquées en jaune)

En utilisant la même méthode, nous avons tracé l'histogramme de fréquence (figure 3.4) pour cette stratégie, et nous pouvons directement comparer que **Version heuristique** a moins d'actions :

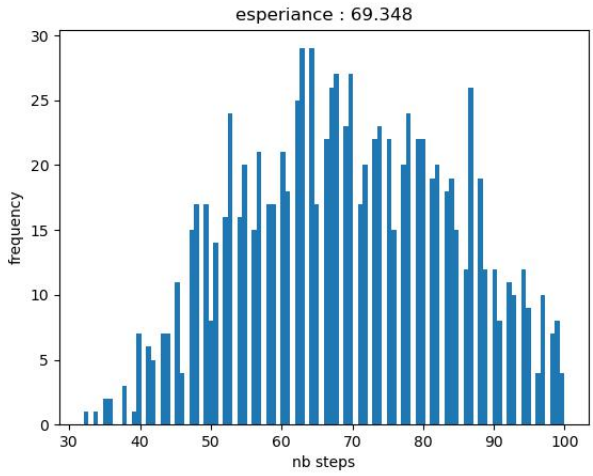


figure 3.4

Version probabiliste simplifiée :

Dans cette stratégie, nous comparons la taille des navires restants avec les points disponibles dans la grille pour tirer certains points impossibles, ou pour sélectionner les points où les navires sont le plus susceptibles d'être présents.

La façon la plus direct d'y parvenir serait d'addresser la liste de toutes les possibilités de placement de tous les navires après chaque tour d'action. Mais cela n'est pas possible car le nombre des placement peut être très grande, ce qui est déjà été démontré dans ***Partie 2.***

Nous sélectionnons donc un navire dans la liste des navires restants après chaque tour d'actions. Nous insérons ensuite ce navire dans la grille ***k fois*** et comptons la fréquence de chaque point où le navire apparaît. Une fréquence plus élevée indique une plus grande probabilité que le navire sélectionné apparaisse à ce point. (*Plus la valeur de k est grande, plus la prédiction est précise*).

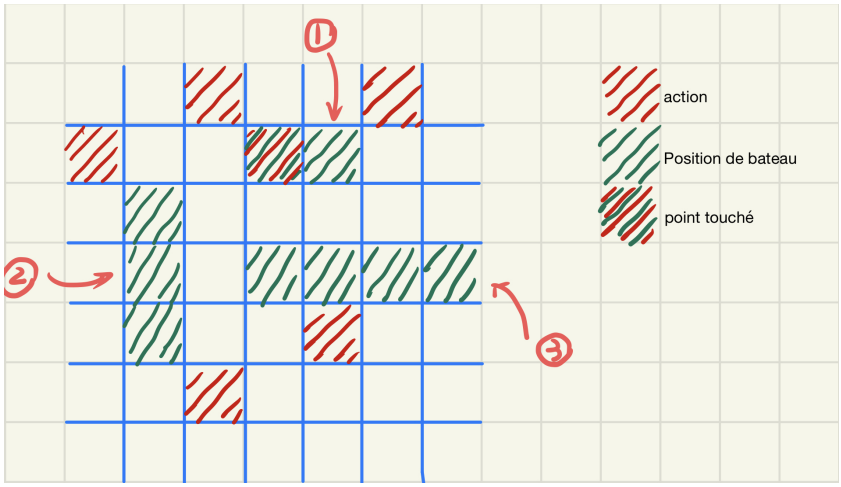


figure 3.5

Comme représenté sur la figure 3.5, le navire 1 est touché et nous allons faire une prédiction à partir d'une sélection aléatoire des navires 2 et 3. Pour chaque tour d'action, nous utiliserons la méthode de la figure 3.3 (tester les points adjacent).

De même, nous avons également tracé l'histogramme de fréquence pour cette stratégie en fonction des résultats de l'expérience (figure 3.6)

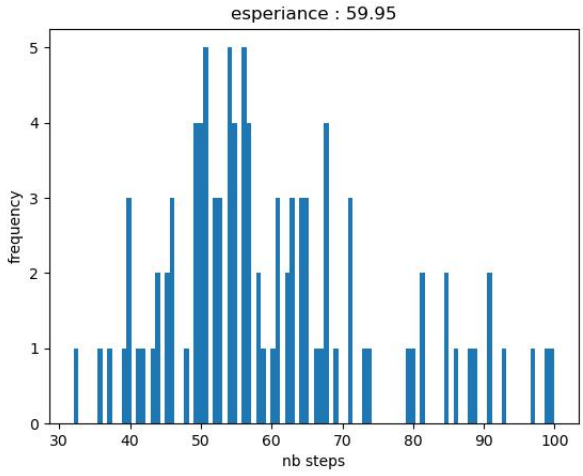


figure 3.6

Version Monte-Carlo :

> 💡 *Le terme **méthode de Monte-Carlo**, ou **méthode Monte-Carlo**, désigne une famille de méthodes algorithmiques visant à calculer une valeur numérique approchée en utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes.*

Selon ma compréhension de l'algorithme de Monte Carlo, ***"plus l'expérience est répétée, plus on se rapproche de la solution optimale"***.

Nous constatons que la méthode utilisée dans la figure 3.3 génère un certain nombre d'actions supplémentaires. Pour éviter cela, nous allons appliquer l'algorithme de Monte-Carlo, ce qui signifie que nous allons considérer la manière de placement de la liste entière de navire.

Nous allons choisir au hasard des navires dans la liste des navires et les insérer dans la grille à tour de rôle. Nous allons nous concentrer sur les points qui ont été touchés, car cela indique qu'un navire doit être présent dans cette zone, tout en évitant tous les points qui n'ont pas été touchés sur la cible.

Comme pour la stratégie précédente, nous allons compter la fréquence de coordonnées d'apparition des navires, sélectionner le point avec la fréquence la plus élevée pour l'action dernière et répéter les étapes ci-dessus.

En fin, nous avons produit un histogramme de fréquence selon cette approche (Figure 3.7):

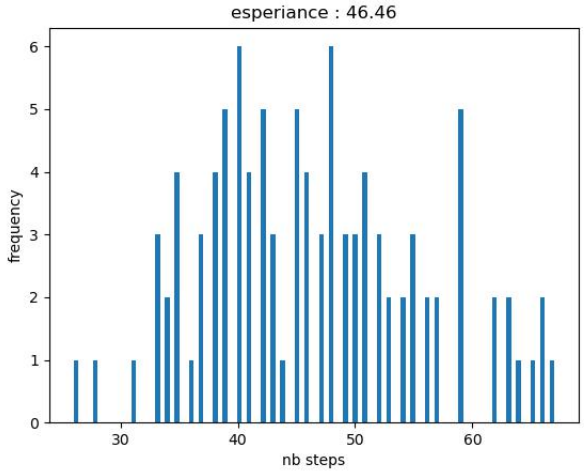


figure 3.7

Conclusion des stratégie:

D'après l'histogramme de fréquence et l'espérance calculée (située en haut du courbe), il s'ensuit que le nombre d'actions est réduit, ce qui augmente les chances du joueur de gagner un jeu.

Nous avons mesuré les chances qu'un navire apparaisse à chaque point en superposant différents placements de navires. Bien entendu, les résultats de l'expérienceont prouvé que notre méthode était

valable.

Partie 4 :

Pour recherche un object plus rapid sur une grille de N cellules, on introduit 4 variables :

- $y_i \in \{0, 1\}$: la variable aléatoire qui vaut 1 pour la case i où l'objet se trouve et 0 partout ailleurs
- $z_i \in \{0, 1\}$: la variable aléatoire qui vaut 1 en cas de détection détection et 0 sinon dans case i
- $\pi_i \in [0, 1]$: la probabilité a propri qui indique la probabilité d'occurrence des objets que nous recherchons
- $ps \in [0, 1]$: la probabilité le senseur détecte l'objet

On peut déduire l'abre suivant (figure 4.1) :

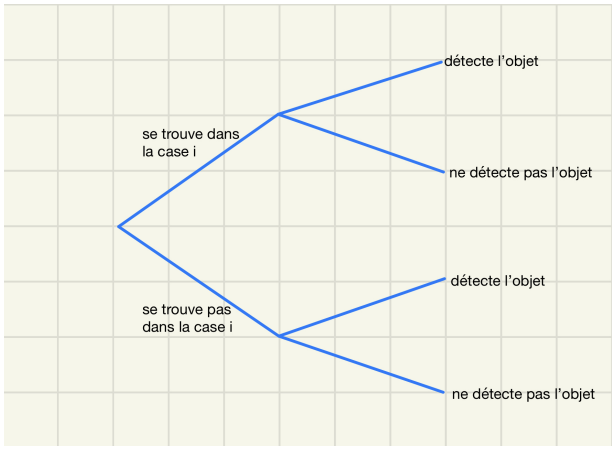


figure 4.1

Et on a :

$$p(z_i = 1|y_i = 1) = ps$$
$$p(z_i = 0|y_i = 1) = 1 - ps$$
$$p(z_i = 1|y_i = 0) = 0$$
$$p(z_i = 0|y_i = 0) = 1$$

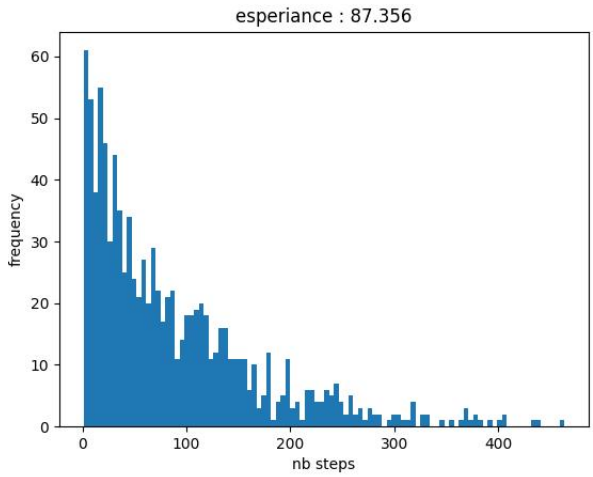
Par conséquent, nous pouvons conclure que les deux événements sont indépendants l'un de l'autre. Donc, pour le sous-marin se trouve en case k et un sondage est effectué à cette case mais ne détecte pas le sous-marin, la probalibité de cet événement est $\pi_k * (1 - ps)$.

Dans ce cas, nous changerons la valeur de π_k pour mesurer à nouveau la probabilité d'apparition d'un sous-marin pour chaque cellule. Nous proposons l'algorithme suivant:

1. On traverse le grille et choisit la cellule qui a le plus grand π_i.
2. Si le senseur détecte le sous-marin, le programme se termine.
3. Sinon, la valeur de π_i de cette case doit être soustraite de $\pi_k * (1 - ps)$. Ensuite, le programme redémarre la boucle à partir de l'étape 1.

En fin, le nombre de fois de détection en moyenne pour comparer cet algorithme avec l'algorithme de traversée (qui détecte π_i le plus élevé, ensuite deuxième plus élevé,… ,et puis k-ième plus élevé).

Nous exécutons le programme (test_Partie4.py) et obtenons l'image suivante:



Conclusion :

Qu'il s'agisse d'une bataille navale ou de la recherche d'un sous-marin, nous pouvons résumer le but de ce projet comme suit : trouver un objet dans une certain grille avec un nombre minimum d'action en base de statistiques et porbabilité. Notre programme amélioré comporte en deux dimensions:

- Nous prédisons les positions possibles d’un objet en listant les possibilités (dénombrement)
- Nous utilisons les résultats de la détection pour modifier la probabilité d'occurrence d‘un objets dans cette région

On peut également dire que le but de notre algorithme est de trouver le point avec la plus grande probabilité d'occurrence de l'objet.