Thème : La ville

TIPE: Modélisation des mouvements de foule

PROBLEMATIQUE :

Quels sont les facteurs influençant les mouvements de foule dans des espaces restreints lors d'une évacuation pour assurer son efficacité et la sécurité des individus ? Comment optimiser ces facteurs ?

10157 POMIES Matéo

Objet d’étude

—> Evacuation d’une foule

—> Evacuation d’un bâtiment

—> 30 / 80 personnes

« *Comment évacuer un bâtiment encore plus rapidement ?* » Nat Géo France 2015

Contexte

Quelques dates :

Leroy Henderson

Années 50: Problèmes de flux piétonniers dans les centre-villes

1971: Leroy Henderson propose le premier modèle de déplacement de foules

Son modèle:

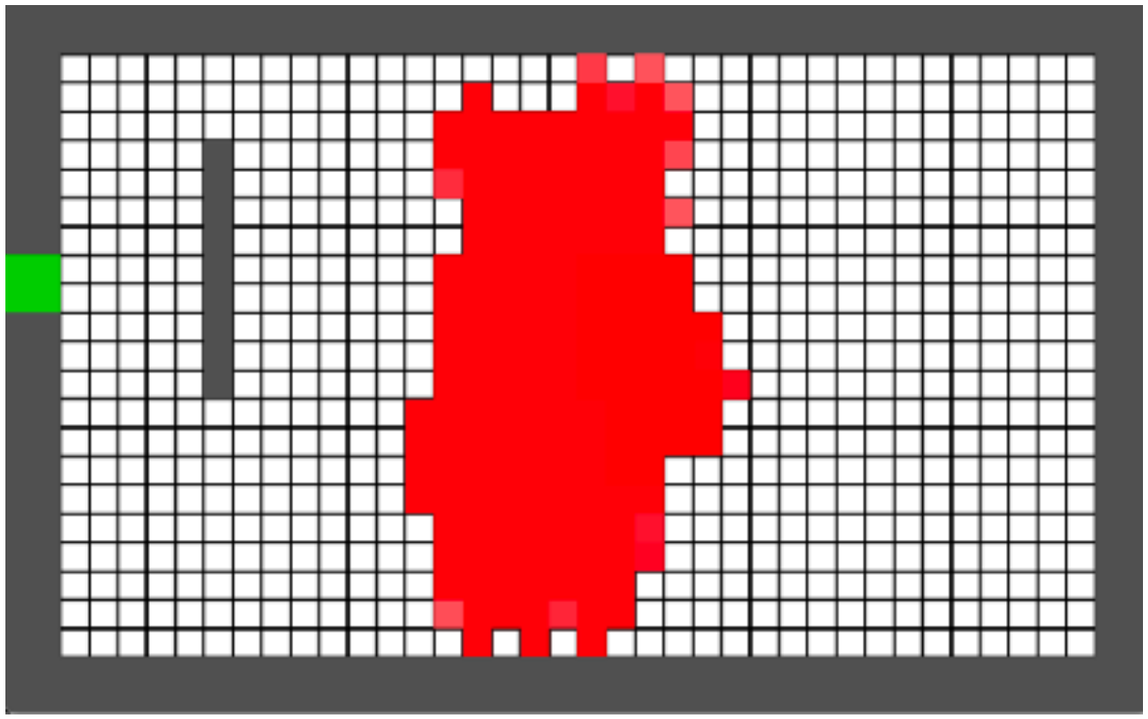
Considérer la foule comme un liquide qui coule dans un tuyau

PLAN

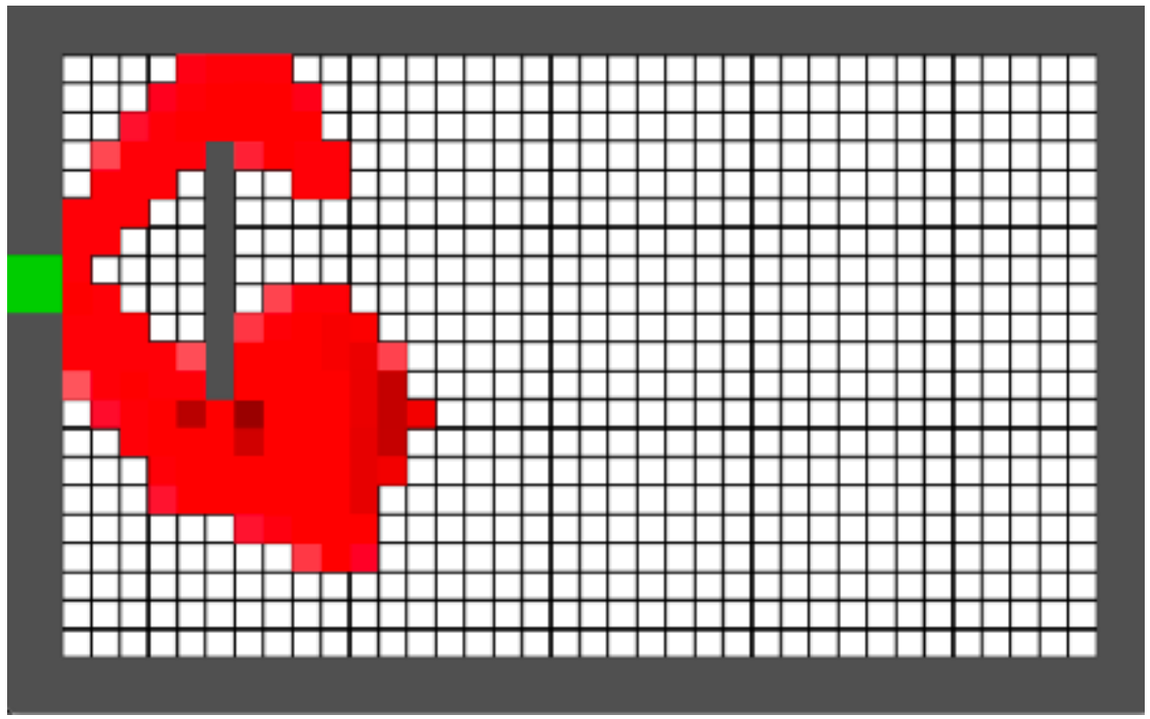
- I) Présentation programme informatique
- II) 1ère étude: phénomène de l'obstacle
- III) 2ème étude : positionnement de la sortie
- IV) Explication avec le modèle de Greenshields

Présentation programme

Interface graphique



Lancement simulation



30 tours après le lancement simulation

Obstacle

Sortie

Espace vide

Foule

Très dense

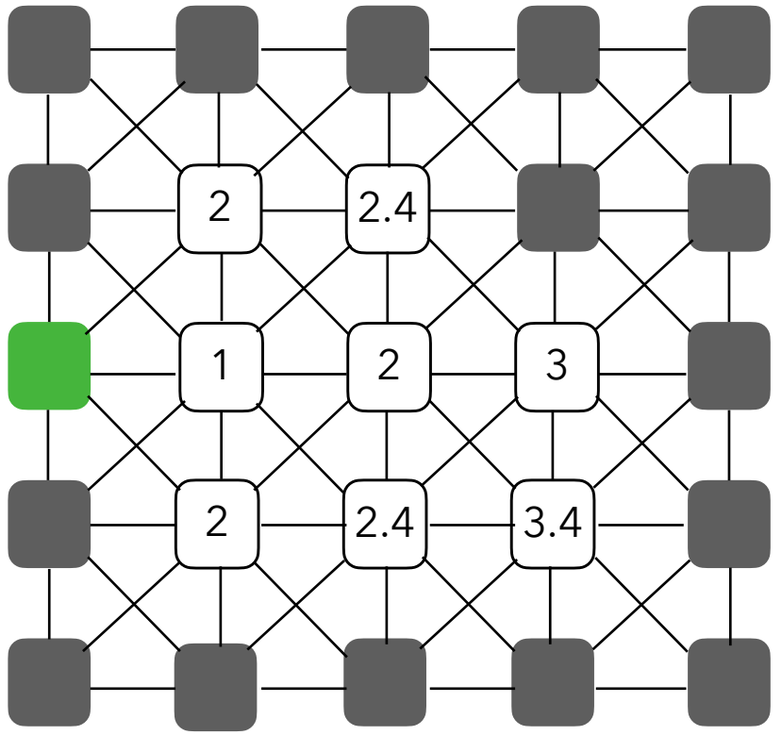
Assez dense

Dense

Peu dense

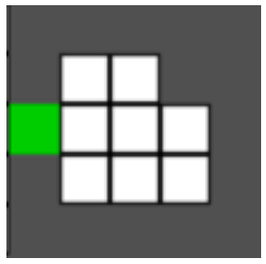
Présentation programme

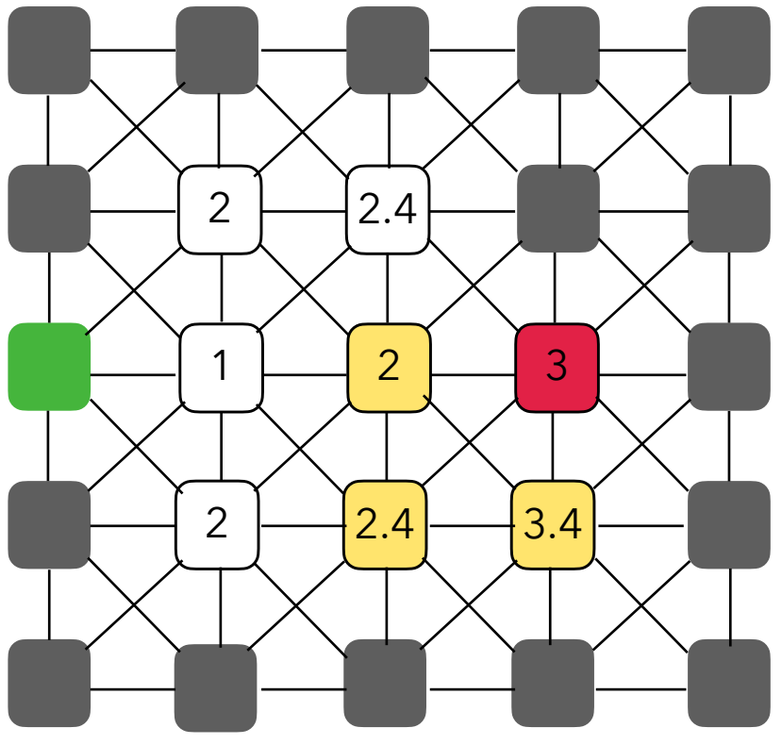
Etude d’un cas simple



Présentation programme

Les voisins



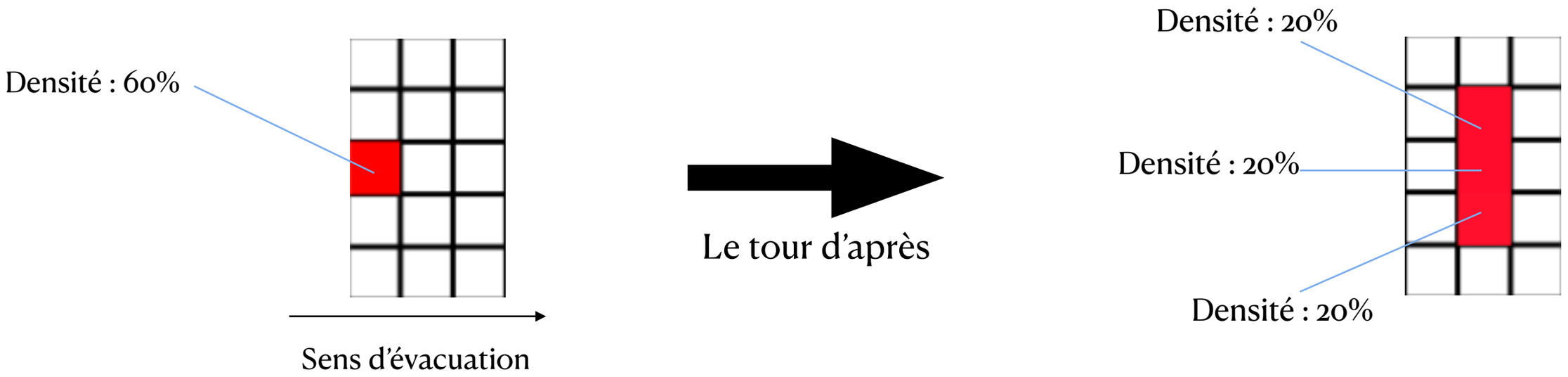


La case foule étudiée

Ses voisins

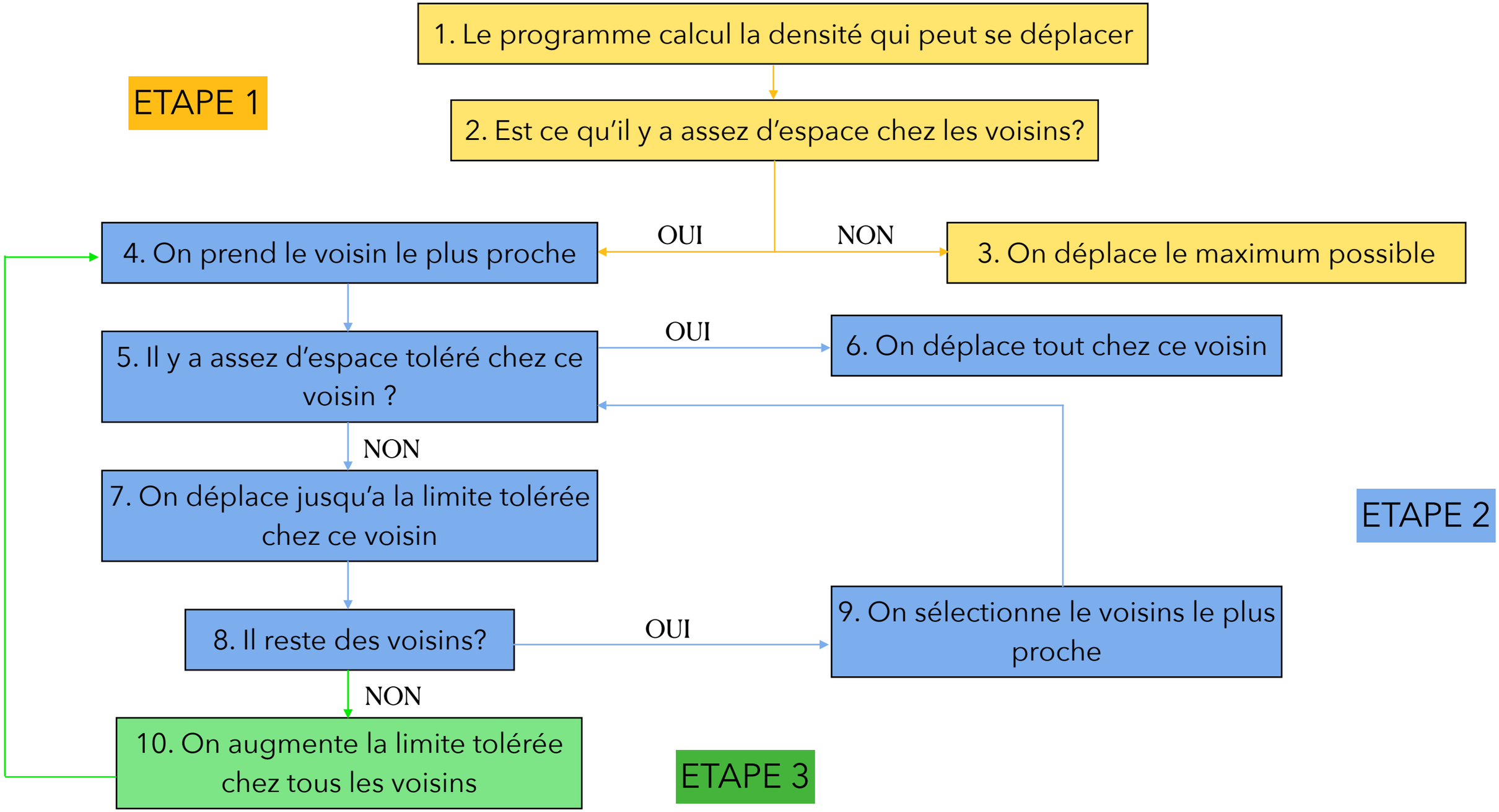
Algorithme de déplacement (1)

Etude d'un tour



Algorithme de déplacement (2)

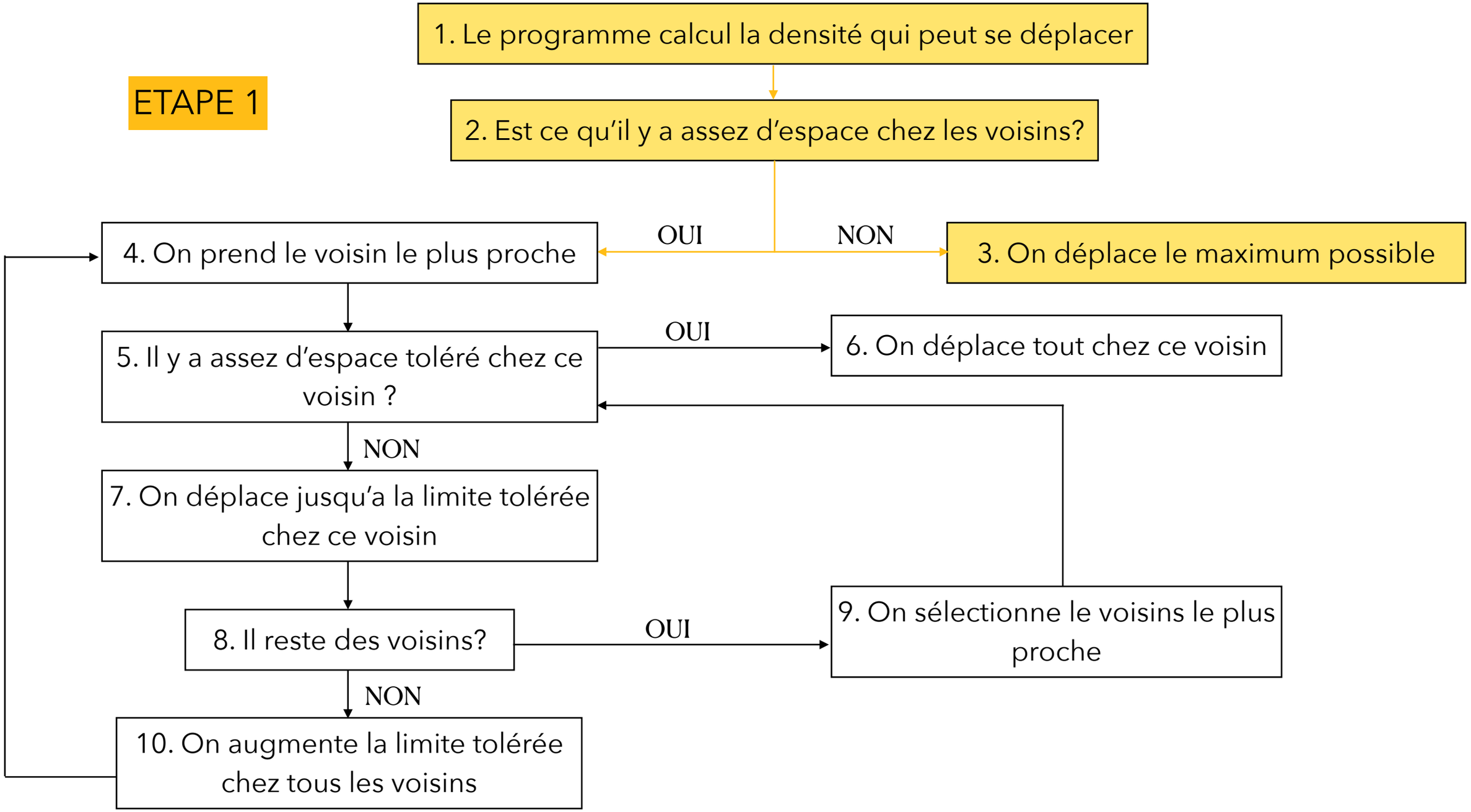
Etude de la distribution de densité



Algorithme de déplacement (2)

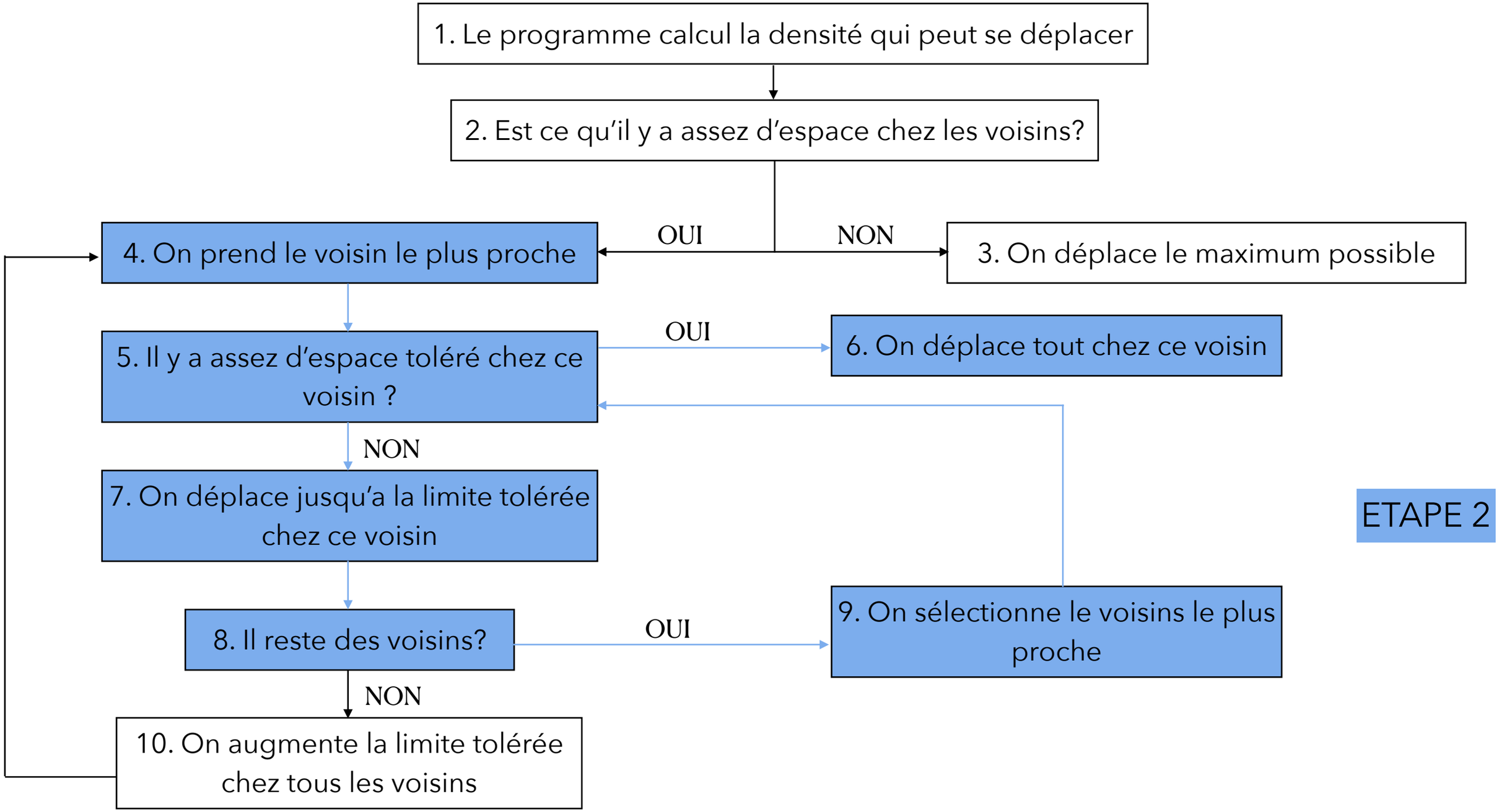
Etude de la distribution de densité

ETAPE 1



Algorithme de déplacement (2)

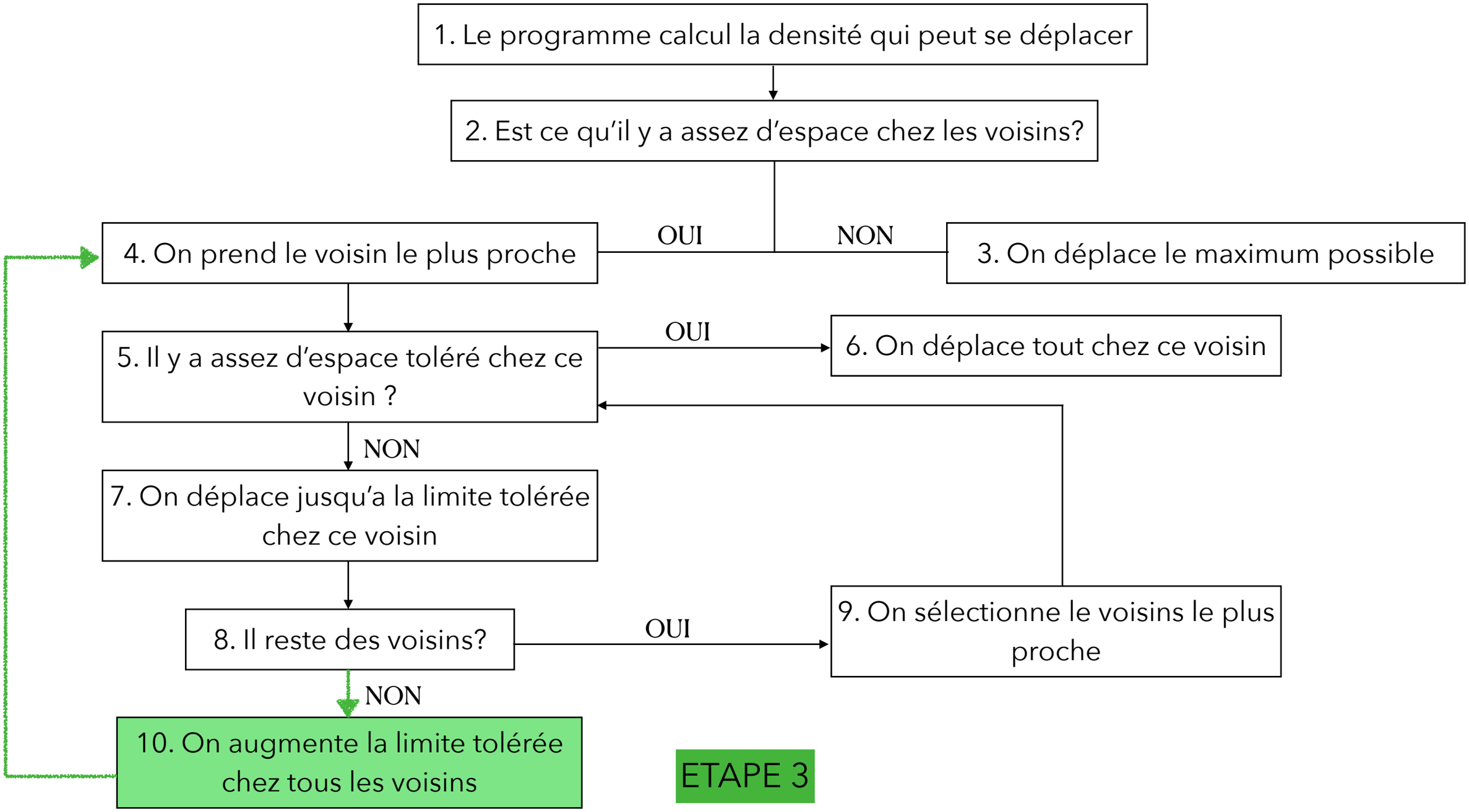
Etude de la distribution de densité



ETAPE 2

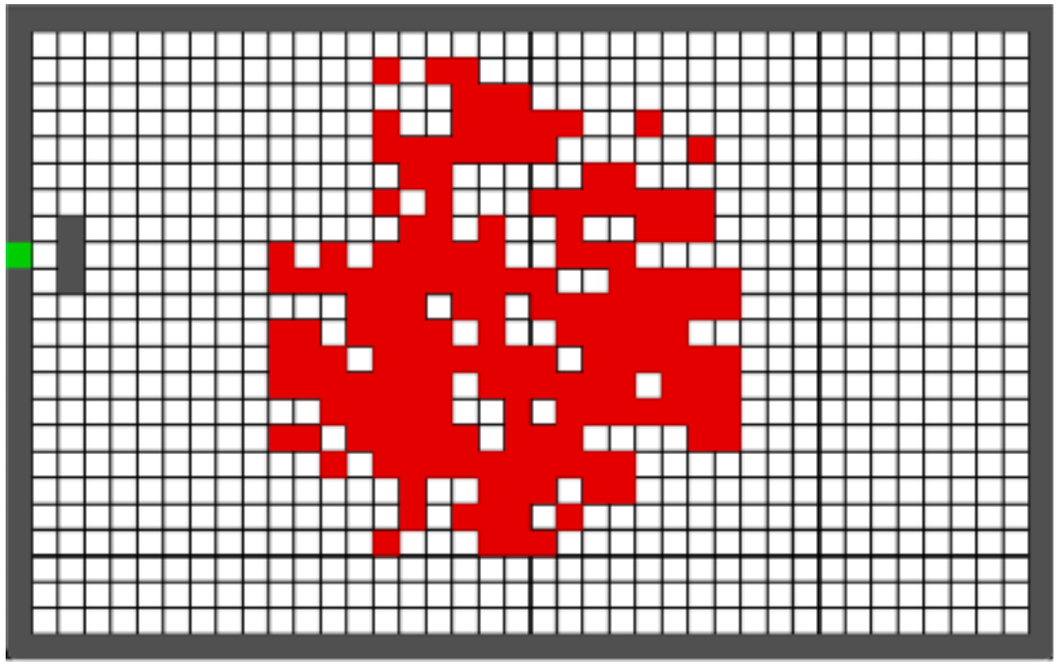
Algorithme de déplacement (2)

Etude de la distribution de densité

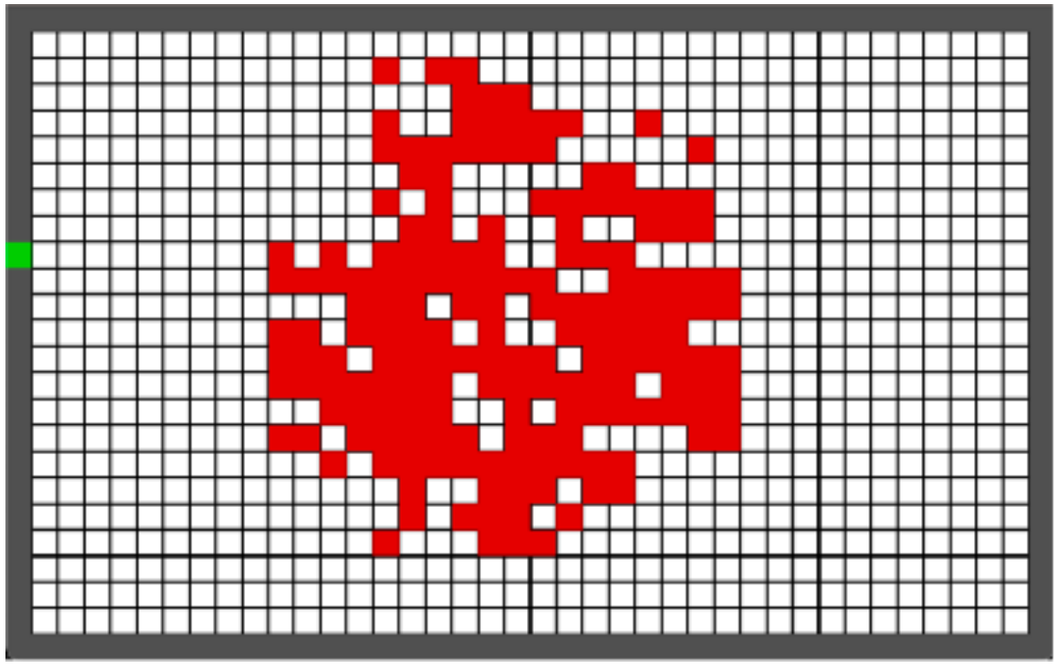


Phénomène du bouchon devant la sortie

Cadre de l'expérience:



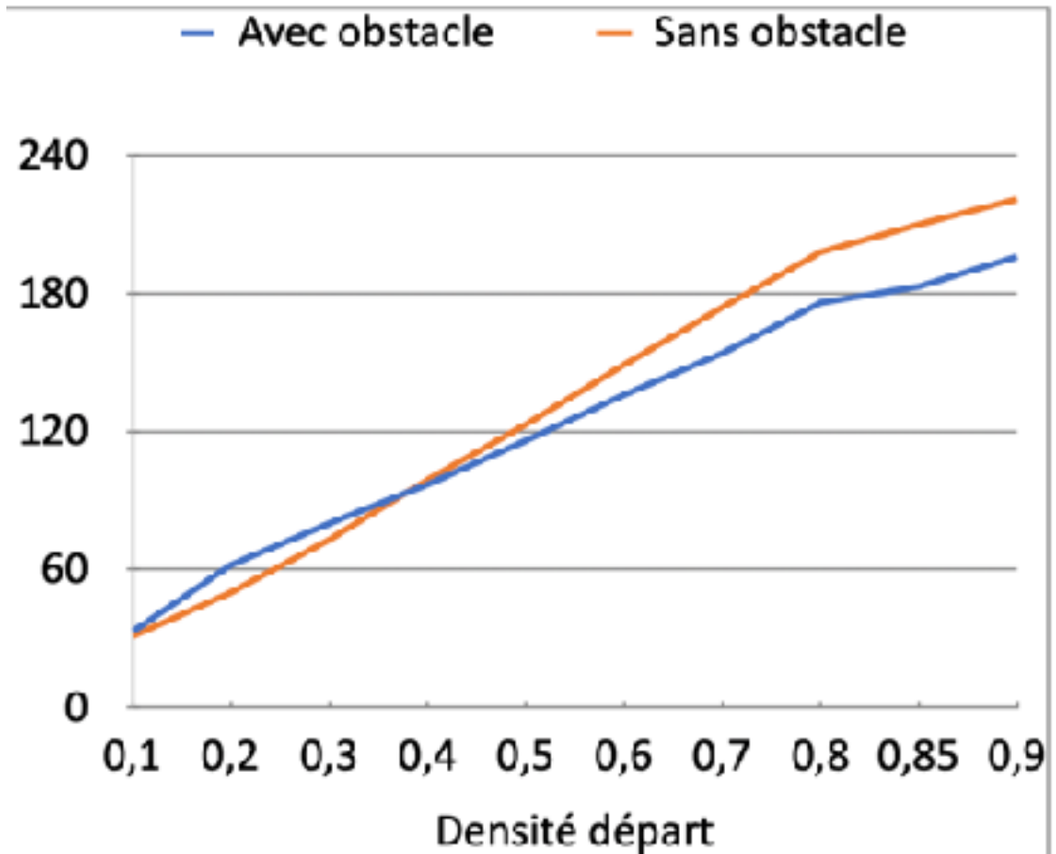
Simulation avec obstacle



Simulation sans obstacle

Resultats:

Nombre de tours nécessaires à l'évacuation

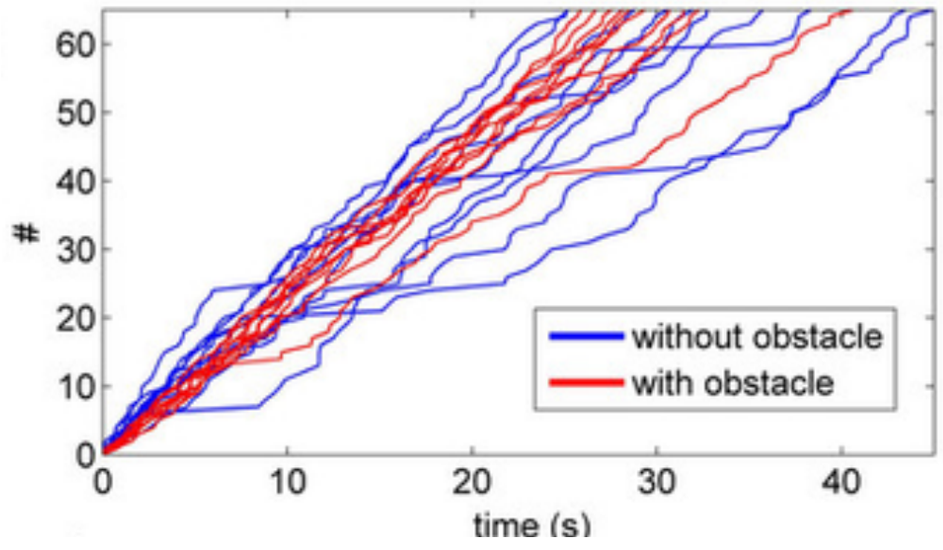


Simulations réelles

Experience 1 :
Avec des moutons

<u>20% plus rapide avec obstacle</u>

Nombre de moutons évacuées par secondes



"Clogging transition of many-particle systems flowing through bottlenecks." Scientific reports 4 (2014)

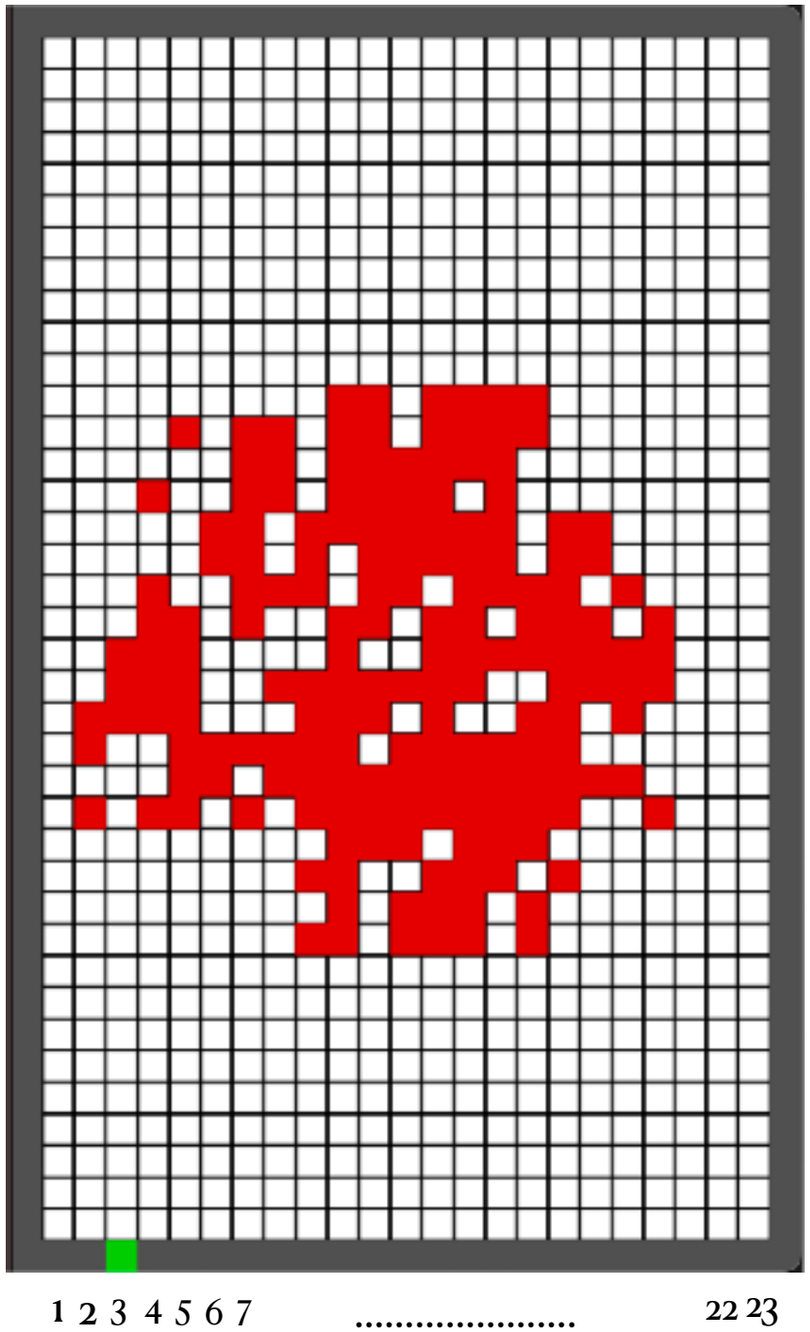
Experience 2 :
Avec des humains

<u>Aucune différence notable</u>

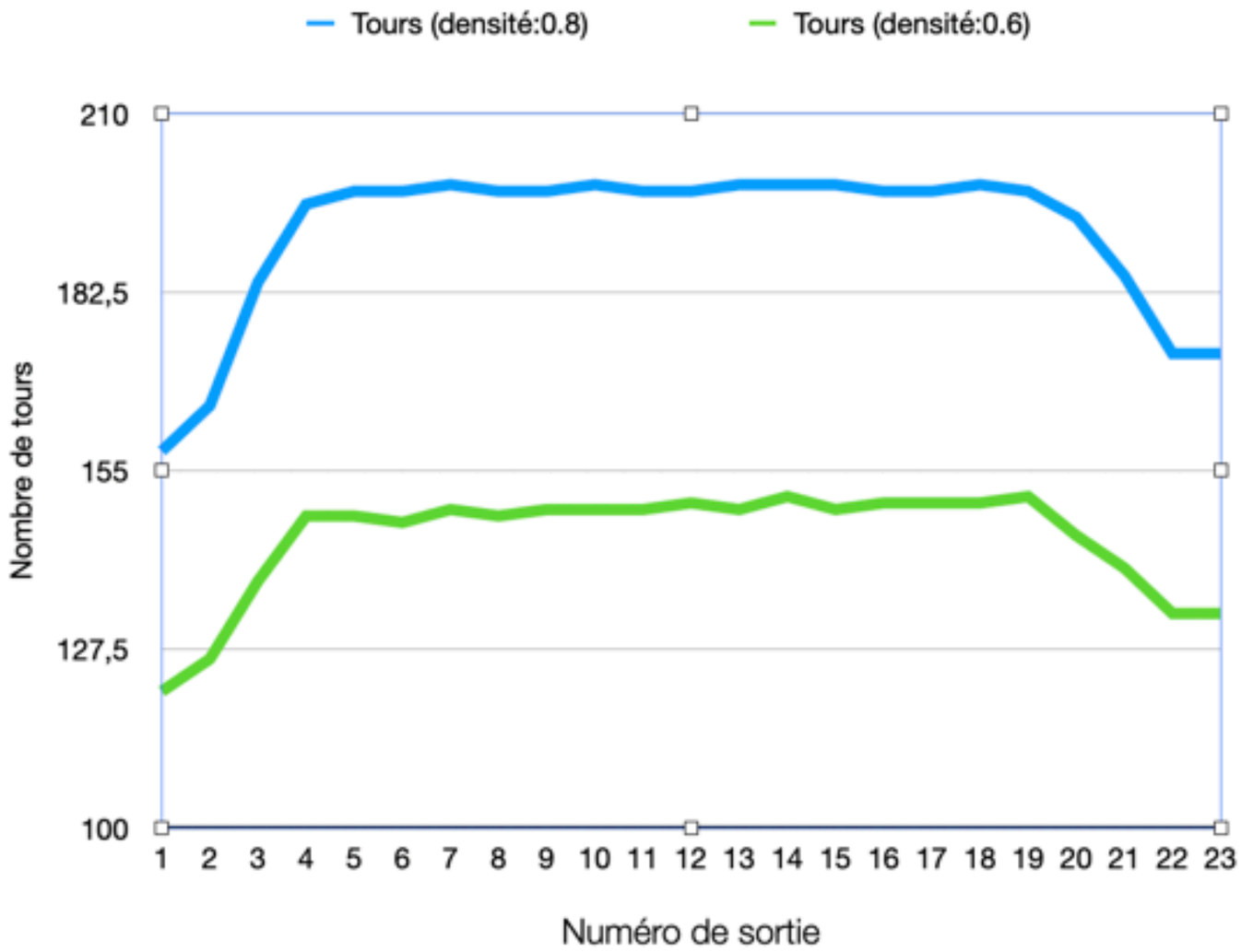
« Redefining the role of obstacles in pedestrian evacuation" IOP Publishing Ltd (2018)

Positionnement de la sortie

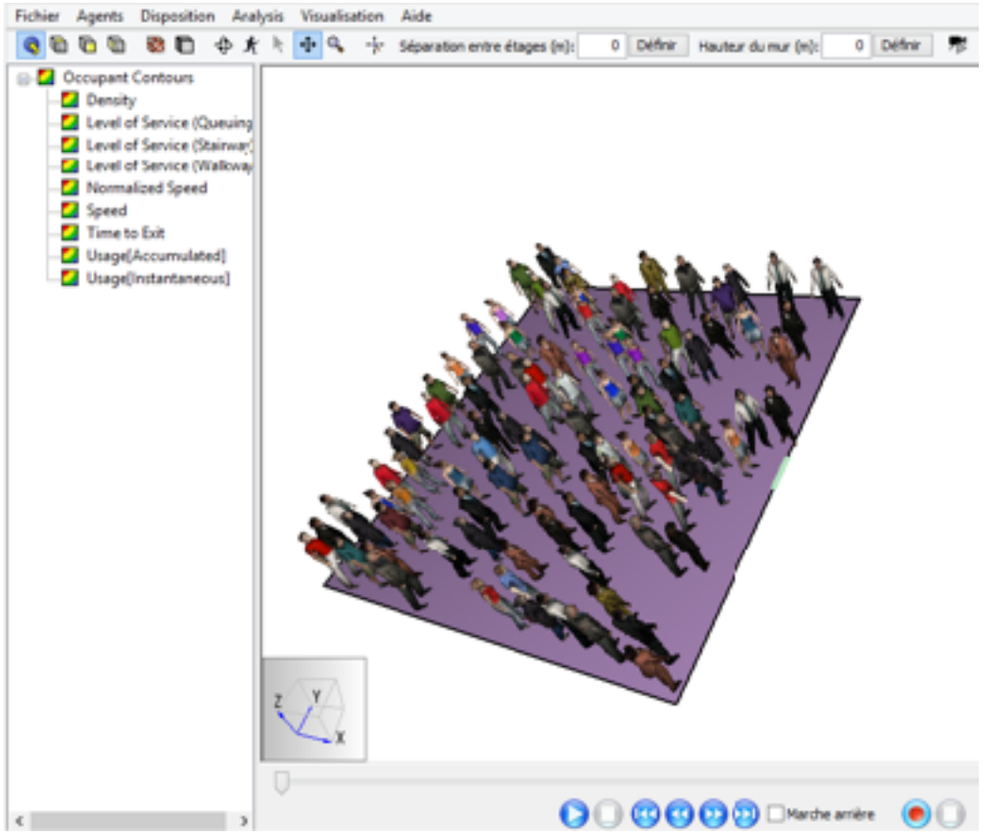
Avec le programme informatique



Resultats



Simulation avec Pathfinder

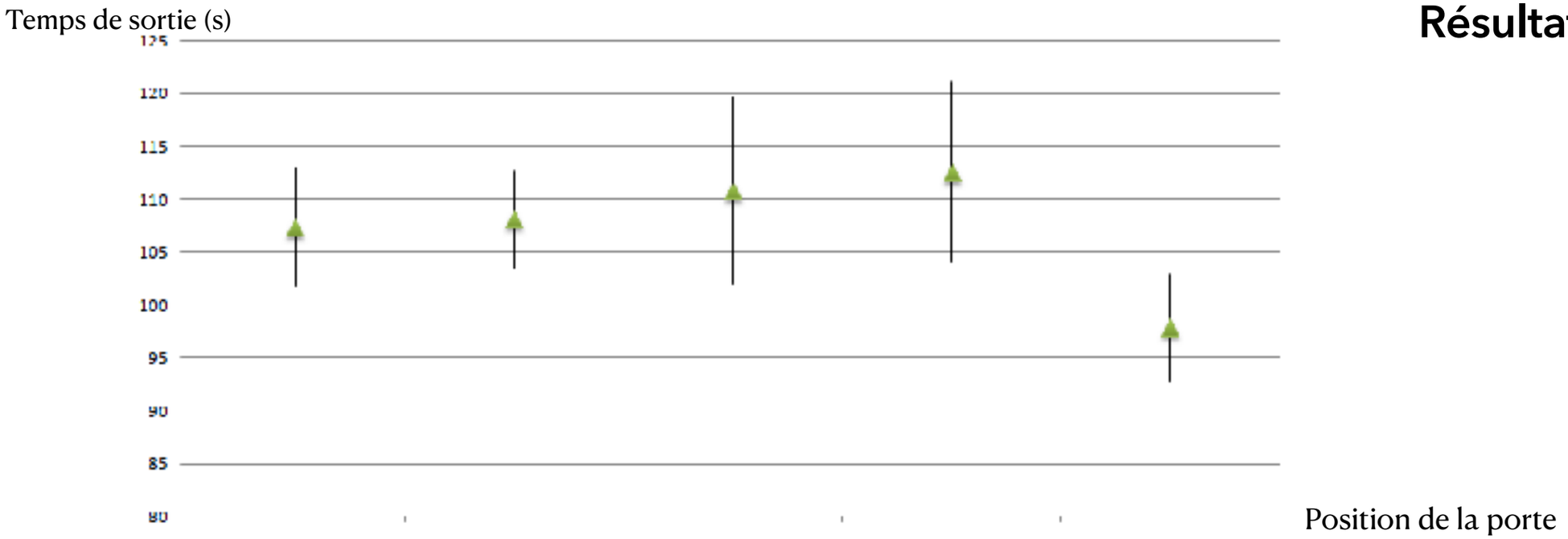


Cadre de l'étude

- Pièce close ($100m^2$)
- Une seule sortie
- 100 personnes

- 10 simulations avec placement aléatoire

Résultats



http://www.simulation-pietons.fr/evacuation-impact-de-lemplacement-de-la-sortie/. 2015

Simulation réelle

(a)

(b)

Cadre de l'étude

- 50 personnes
- Salle de $10m^2$
- 12 essais par sortie

Conclusion de l'étude

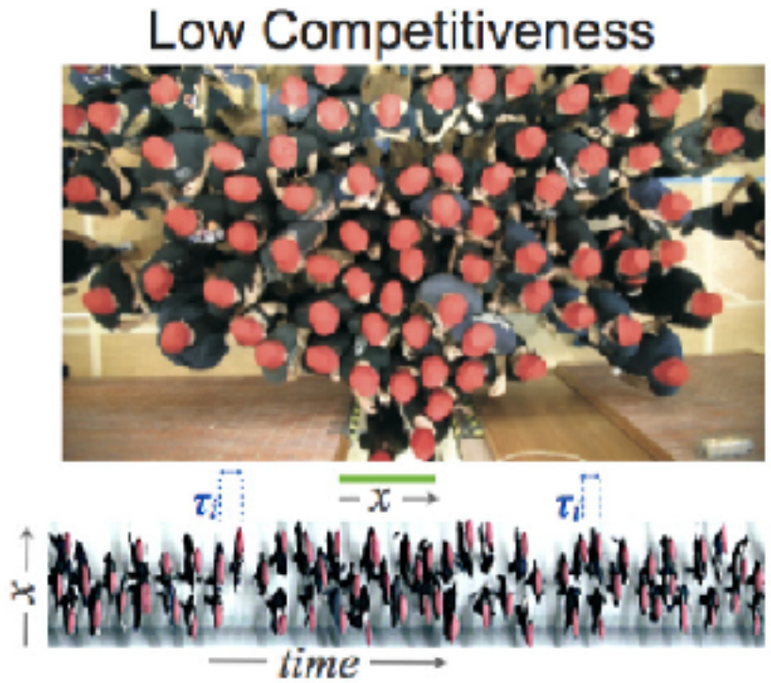
La sortie dans le coin est 8.6% plus efficace

« A comparative study of pedestrian crowd flow at middle and corner exits », Jiangsu Key Laboratory of Urban ITS, 2016

Les causes de ces phénomènes

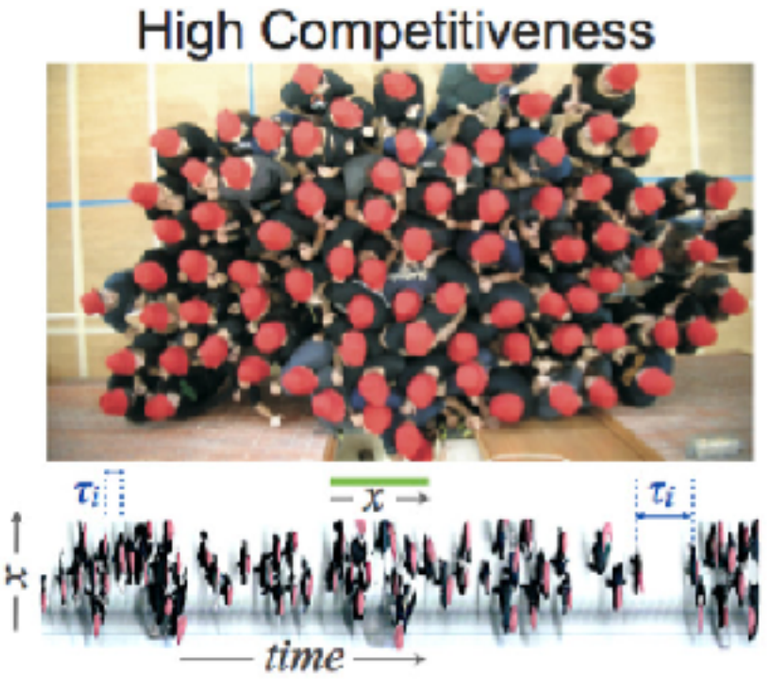
Avant tout : une première expérience

Situation 1 :
Tout le monde sort lentement vers la sortie



Temps de sortie : 34 secondes

Situation 2 :
Tout le monde se précipite vers la sortie



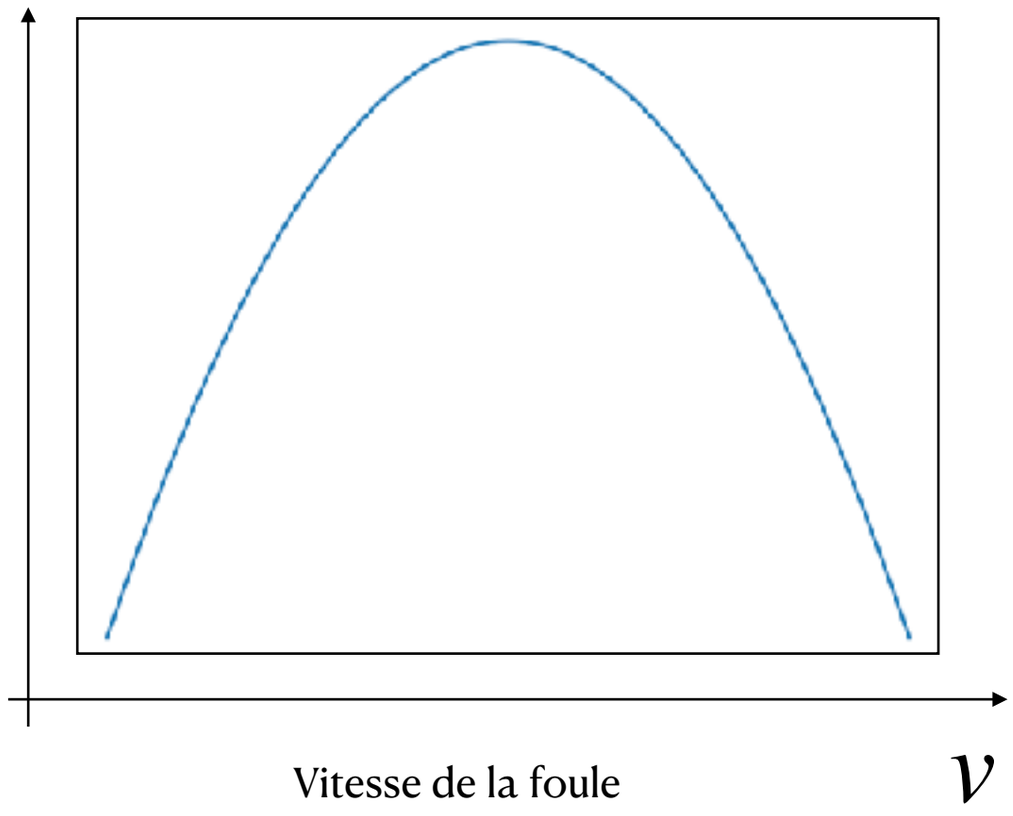
Temps de sortie : 43 secondes

« Redefining the role of obstacles in pedestrian evacuation" IOP Publishing Ltd (2018)

Modèle de Greenshields

Modèle physique de congestion de foule:

$$q = k_j \,.\, v - \frac{k_j}{v_f} \,.\, v^2$$



v_f : Vitesse maximale

k_j : Densité maximale

q : Débit de foule

v : Vitesse des individus

« *Pedestrian Speed-Flow-Density relationships*» On Transportation Research Board (1994)

Conclusion

Objectif 1 : réussir à développer une simulation informatique cohérente

— > Cohérente avec les attendus théoriques

— > Résultats pas toujours conformes aux experiences

Objectif 2 : Facteurs déterminants dans les déplacements de groupes et solutions

— > Facteurs : Panique / Densité

— > Solutions : Diviser / Ralentir localement la foule

Possibilité d'amélioration :

— > Utilisation de l'intelligence artificielle

```python
#-----------------------------------------------------------------
# Modification de la carte
#-----------------------------------------------------------------
# width : largeur de la fenêtre en pixel
# height : hauteur de la fenêtre en pixel
size = width, height = [illegible]

# cellules_hauteur : nombre de cellules sur une colonne
# cellules_largeur : nombre de cellules sur une ligne
cellules_hauteur = 25
cellules_largeur = 46

#-----------------------------------------------------------------
# Modification du comportement du programme
#-----------------------------------------------------------------

# portee_ajout_foule : portée du clic de l'ajout de foule (pour ajouter beaucoup de cases foule rapidement)
portee_ajout_foule = 1

# plan : fichier texte à charger pour charger la carte de simulation
plan = "cartes/Plan_base.txt"

# seconde_par_deplacement : temps en seconde à attendre au minimum entre chaque tour
seconde_par_deplacement = 0.5

# densite_sortie_imposee : la densitée autorisée à se déplacer est fixée par une fonction, si mis sur True
# ignore la densitée calculée, et impose la valeur définie pour sortir de la pièce
densite_sortie_imposee = False

#-----------------------------------------------------------------
# Constantes à ne pas toucher
#-----------------------------------------------------------------
pygame.init()
densite_maximale_supportee = 5
densite_sortie = 2.
foule_pourcentage_depart = 0
coef_deplacement_densite = []
dico_carte = {}  # Ce dictionnaire contient les informations de toutes les cellules et de leur état
dico_distance_euclidienne = {}
dico_pcpe = {}
screen = pygame.display.set_mode(size)
horloge = time.process_time()
sorties = []
dico_chemin = {}  # Dictionnaire contenant les cases qui représentent le chemin que les cases foules vont prendre
temps_avant_deplacement = 0  # Constante lié à l'horloge
nombre_tours_simulation = 0
dico_numerotation_chemin = {}
couleur_obstacle = (82, 82, 82)  # gris
couleur_sortie = (34, 181, 26)  # vert
# Variables pour les statistiques
densite_moyenne_tour = []
# Mode édition. Ce dictionnaire contient tous les états actuels du mode édition
bool_edition = {
    'edition': False,
    'obstacle': False,
    'sortie': False,
    'foule': False,
}
demarrer_simulation = False
affichage_chemin = False

#-----------------------------------------------------------------
# Début du programme
#-----------------------------------------------------------------

def affichage_console():
    """Affiche le menu pour modifier le programme pendant son fonctionnement"""
    menu = {
        "Mode édition": '(p)',
        "Sauvegarde": '(s)',
        "Afficher/Masquer chemin": '(c)',
        "Démarrer/Arrêter simulation": '(entrer)'
    }
    sous_menu = {
        "Mode édition": [['Ajout obstacle', '(o)'], ['Ajout sortie', '(s)'], ['Ajout foule', '(f)']]
    }
    print("Mode emploi (touches)")
    for key in menu.keys():
        print(f"{menu[key]} : {key}")
        if key in sous_menu.keys():
            for item in sous_menu[key]:
                print(f"    {item[1]} : {item[0]}")


def importer_parametres():
    """Importe les paramètres du programme"""
    densite_maximale_supportee, foule_pourcentage_depart, densite_sortie = 0, 0, 0  # Initialisation des variables po
    with open("[illegible]", 'r') as fichier:
        lignes = fichier.readlines()
        # On récupère d'abord les valeurs de deplacement_densite
        tableau_deplacement_densite = []
        for valeur in lignes[0].split(";"):
            # On ne veut pas prendre en compte le saut d'espace dans la dernière variable
            if valeur[len(valeur)-1] == "\n":
                valeur = valeur[:-1]
            tableau_deplacement_densite.append(float(valeur))
        # Ensuite on importe les autres variables
        lignes.pop(0)
        for ligne in lignes:
            ligne = ligne.split(':')
            if ligne[0] == "densite_maximale_supportee":
                densite_maximale_supportee = float(ligne[1].split("\n")[0])
            elif ligne[0] == "foule_pourcentage_depart":
                foule_pourcentage_depart = float(ligne[1].split("\n")[0])
            elif ligne[0] == "densite_sortie":
                densite_sortie = float(ligne[1].split("\n")[0])
            else:
                print("Erreur dans l'importation des paramètres : paramètre inconnu")
        return tableau_deplacement_densite, densite_maximale_supportee, foule_pourcentage_depart, densite_sortie


def definir_densite_foule_depart(densite):
    """Définit à la même densité toutes les foules au lancement d'une simulation"""
    for case, type in dico_carte.items():
        if type[0] == "F":
            dico_carte[case] = ['F', densite]


def calcul_densite(voisins):
    """Calcul la densité totale des voisins autour d'une case"""
    somme_densite = 0  # Somme des densités des voisins
    for voisin in voisins:
        if voisin in dico_carte:
            somme_densite += dico_carte[voisin][1]
    return somme_densite/[illegible]


def fusion(gauche, droite):
    """Fonction complémentaire pour tri foule et fusion"""
    resultat = []
    index_gauche, index_droite = 0, 0
    while index_gauche < len(gauche) and index_droite < len(droite):
        if gauche[index_gauche][1] <= droite[index_droite][1]:
            resultat.append(gauche[index_gauche])
            index_gauche += 1
        else:
            resultat.append(droite[index_droite])
            index_droite += 1
    if gauche:
        resultat.extend(gauche[index_gauche:])
    if droite:
        resultat.extend(droite[index_droite:])
    return resultat
```

```python
def tri_foule(tableau):
    """Tri par fusion les éléments dans l'ordre croissant d'un tableau de tableau en fonction de son deuxième élément
    exemple : [[element1, 33], [element2, 28]] devient [[element2, 28], [element1, 33]]
    """
    if len(tableau) <= 1:
        return tableau
    milieu = len(tableau) // 2
    gauche = tableau[:milieu]
    droite = tableau[milieu:]
    gauche = tri_foule(gauche)
    droite = tri_foule(droite)
    return list(fusion(gauche, droite))


def distance_euclidienne(i, j, chemin):
    """Calcul la longueur d'un chemin en mètres (1 case = 1 mètre)"""
    chemin.reverse()  # Car le chemin commence au dernier indice

    if len(chemin) == 0:
        return 0
    longueur_chemin = 0
    case_precedente = (i, j)  # On initialise
    for k in range(0, len(chemin)):
        if chemin[k][0] != case_precedente[0] and chemin[k][1] != case_precedente[1]:  # Longueur d'une diagonale
            longueur_chemin += 1.41  # Racine carrée de 2
        elif chemin[k][0] != case_precedente[0] or chemin[k][1] != case_precedente[1]:  # Longueur d'un côté
            longueur_chemin += 1
        elif chemin[k][0] == case_precedente[0] and chemin[k][1] == case_precedente[1]:  # Elle se déplace sur elle même
            return 0
        else:
            print("Problème dans le calcul de la distance euclidienne")
        case_precedente = chemin[k]
    chemin.reverse()
    return longueur_chemin


def affichage(cases_changement, tout=False):
    """Permet de gérer l'affichage de l'application"""
    taille_largeur = int(size[0] / cellules_largeur)
    taille_hauteur = int(size[1] / cellules_hauteur)

    if tout is True:  # Activé quand on ouvre le programme, permet de charger toute la carte
        cases_changement = []
        for i in range(0, cellules_hauteur):
            for j in range(0, cellules_largeur):
                cases_changement.append((i, j))

    # Affichage des cellules sur l'application avec leur couleur respective
    for case in cases_changement:
        j, i = case[0], case[1]
        if (j, i) in dico_carte:  # Regarde si c'est une cellule vide ou si son état est connu
            if dico_carte[(j, i)][0] == 'S':
                couleur = couleur_sortie
            elif dico_carte[(j, i)][0] == 'O':
                couleur = couleur_obstacle
            elif dico_carte[(j, i)][0] == 'C':
                couleur = (255, 255, 0)
            elif dico_carte[(j, i)][0] == 'F':
                pourcentage_rouge = dico_carte[(j, i)][1]
                if affichage_chemin is True:  # Affiche en jaune le chemin de chaque case foule
                    afficher_chemin(j, i)
                if pourcentage_rouge < 0.5:  # Gestion de la couleur de la foule
                    couleur = (255, 200 * (1.5 - pourcentage_rouge), 200 * (0.5 - pourcentage_rouge))
                else:
                    couleur = (255 - 130 * (pourcentage_rouge - 0.5), 0, 0)

            else:
                print("Problème dans la couleur d'une case ")
            pygame.draw.rect(screen, couleur,
                             pygame.Rect(i * taille_largeur, j * taille_hauteur, taille_largeur, taille_hauteur))
        elif (j, i) in dico_chemin:  # Si c'est un chemin jaune
            couleur = (255, 255, 0)
            pygame.draw.rect(screen, couleur,
                             pygame.Rect(i * taille_largeur, j * taille_hauteur, taille_largeur, taille_hauteur))
        else:
            # On appelle 2 fois pygame.draw.rect pour 'nettoyer' le fond du carré
            pygame.draw.rect(screen, (255, 255, 255),
                             pygame.Rect(i * taille_largeur, j * taille_hauteur, taille_largeur, taille_hauteur))
            pygame.draw.rect(screen, (0, 0, 0),
                             pygame.Rect(i * taille_largeur, j * taille_hauteur, taille_largeur, taille_hauteur), 1)

def check_sorties():
    """Actualise la variable globale "sorties" """
    sorties.clear()
    for key, item in dico_carte.items():
        if item[0] == 'S':
            sorties.append(key)


def determiner_dico_distance_euclidienne():
    dico_distance_euclidienne.clear()
    for sortie in sorties:
        dico_distance_euclidienne[(sortie[0], sortie[1])] = 0
        liste_attente = [(sortie[0], sortie[1])]
        deja_vu = [()]
        fin = False
        while fin is False:
            case = (liste_attente[0][0], liste_attente[0][1])
            deja_vu.append(case)

            voisins = check_entourage(case[0], case[1], [], distance_non_euclidienne=True)[1]
            for voisin in voisins:
                distance_sortie = 0
                # On regarde si c'est un voisin diagonal ou un voisin direct
                if case[0] != voisin[0] and case[1] != voisin[1]:  # Voisin diagonal
                    distance_sortie = dico_distance_euclidienne[case] + 1.41
                elif case[0] != voisin[0] or case[1] != voisin[1]:  # Voisin direct
                    distance_sortie = dico_distance_euclidienne[case] + 1
                else:
                    distance_sortie = 99999

                if voisin in dico_distance_euclidienne:
                    if dico_distance_euclidienne[voisin] > distance_sortie:
                        dico_distance_euclidienne[voisin] = distance_sortie
                else:
                    dico_distance_euclidienne[voisin] = distance_sortie

                if voisin not in deja_vu and voisin not in liste_attente:
                    liste_attente.append(voisin)
            liste_attente.pop(0)
            if len(liste_attente) == 0:
                fin = True

    # Pour les cases qui n'ont pas été parcourues, c'est qu'elles ne sont pas accessibles par une sortie
    for j in range(0, cellules_hauteur):
        for i in range(0, cellules_largeur):
            if (j, i) not in dico_distance_euclidienne:
                dico_distance_euclidienne[(j, i)] = 99999999


def determiner_dico_pcpe():
    """Détermine le plus court chemin en distance euclidienne vers la sortie pour chaque case en ne prenant pas en
    compte la foule comme obstacle """
    dico_pcpe.clear()
    if len(sorties) == 0:
        return True  # Il n'y a pas de sortie donc ce dictionnaire est vide
    for i in range(0, cellules_hauteur):
        for j in range(0, cellules_largeur):
            calcul_autorise = False
            if (i, j) in dico_carte:
                if dico_carte[(i, j)][0] == 'F':
                    calcul_autorise = True
            else:
                calcul_autorise = True

            if calcul_autorise is True:
                sortie_trouve, _ = check_entourage(i, j, sorties)  # Le check_entourage renvoie l'entourage dans
                # l'ordre du plus court vers la sortie en distance euclidienne
                chemin = [(i, j)]
                while sortie_trouve[0] is False:
                    sortie_trouve, prochain_chemin = check_entourage(chemin[len(chemin) - 1][0],
                                                                    chemin[len(chemin) - 1][1], sorties)
                    if sortie_trouve[0] is False:
                        while prochain_chemin[0] in chemin:
                            prochain_chemin.pop(0)
                        chemin.append(prochain_chemin[0])
                chemin.append(sortie_trouve[1])
                chemin.pop(0)
                chemin.reverse()
                dico_pcpe[(i, j)] = chemin
```

```python
def trier_entourage_distance_euclidienne(voisins, case_depart):
    """Prend la liste des voisins proches pour les trier selon leur distance à la sortie la plus proche"""
    liste_triee = []
    for voisin in voisins:
        if voisin[0] != case_depart[0] and voisin[1] != case_depart[1]:
            liste_triee.append([voisin, dico_distance_euclidienne[voisin] * 1.41])
        elif voisin[0] != case_depart[0] or voisin[1] != case_depart[1]:
            liste_triee.append([voisin, dico_distance_euclidienne[voisin] * 1])
        else:
            liste_triee.append([voisin, dico_distance_euclidienne[voisin]])
    liste_triee = tri_foule(liste_triee)
    return [voisin[0] for voisin in liste_triee]


def trier_entourage(voisins, case_depart):
    """Prend la liste des voisins proches pour les trier sur les voisins directement proches en priorité plutôt que
    ceux sur les coins"""
    liste_triee = []
    liste_attente = []
    for voisin in voisins:  # On ajoute en premier les voisins direct
        if voisin[0] == case_depart[0] or voisin[1] == case_depart[1]:
            liste_triee.append(voisin)
        else:
            liste_attente.append(voisin)
    for voisin in liste_attente:  # On ajoute à la fin les voisins dans les coins
        liste_triee.append(voisin)
    return liste_triee


def check_entourage(i, j, tableau_sortie, foule=False, distance_non_euclidienne=False):
    """Cette fonction renvoie la liste de l'entourage vide de la case demandée, ou si la sortie est à proximité"""
    voisins_vides = []
    is_sortie = [False, ()]
    for k in range(-1, 2):
        for l in range(-1, 2):
            coin_licite = True
            if l == 0 and k == 0:  # La case elle-même ne peut pas être son entourage
                print('', end='')  # rien
            elif (k != 0) and (l != 0):  # Si c'est un coin, on vérifie qu'on puisse y accéder
                if (i + k, j) in dico_carte:
                    if dico_carte[(i + k, j)][0] == 'O':
                        coin_licite = False  # Coin inaccessible
                if (i, j + l) in dico_carte:
                    if dico_carte[(i, j + l)][0] == 'O':
                        coin_licite = False  # Coin inaccessible
            if (i + k) < 0 or (j + l) < 0:  # Hors de la carte
                coin_licite = False
            elif (i + k) >= cellules_hauteur or (j + l) >= cellules_largeur:  # Hors de la carte
                coin_licite = False
            if coin_licite is True:
                if (i + k, j + l) in tableau_sortie:
                    is_sortie = [True, (i + k, j + l)]
                    return is_sortie, trier_entourage(voisins_vides, (i, j))
                if (i + k, j + l) not in dico_carte:
                    voisins_vides.append((i + k, j + l))
                elif dico_carte[(i + k, j + l)][
                    0] == 'F' and foule is False:  # On ne compte pas la foule comme un obstacle
                    voisins_vides.append((i + k, j + l))
    if distance_non_euclidienne is False:
        return is_sortie, trier_entourage_distance_euclidienne(voisins_vides, (i, j))  # très instable
    return is_sortie, trier_entourage(voisins_vides, (i, j))


def pcbs(i, j, tab_sortie, foule_en_obstacle=False):  # (Plus Court Chemin vers Sortie)
    """Calcul pour chaque foule le plus court chemin en utilisant un parcours en largeur"""
    file_attente = [(i, j)]
    deja_vu = []
    peres = []  # ((fils), (pere))
    sortie = False
    coord_sortie = ()
    # Pour ne pas recalculer inutilement plusieurs fois le même chemin, on regarde si le chemin a déjà été calculer
    # pour une case, puis on vérifie qu'il marche toujours.

    chemin_libre = True
    if (i, j) in dico_memoisation_chemin and foule_en_obstacle is False and tab_sortie == sorties:
        for case in dico_memoisation_chemin[(i, j)]:
            if case in dico_carte:
                if dico_carte[(case[0], case[1])][0] != 'F' and dico_carte[(case[0], case[1])][0] != 'S':
                    del dico_memoisation_chemin[(i, j)]
                    chemin_libre = False
        if chemin_libre == True:
            return dico_memoisation_chemin[(i, j)]

    # Début de l'algorithme de parcours en largeur
    while sortie is False:

        if len(file_attente) == 0:  # Dans le cas où on a déjà parcouru toute la carte
            if foule_en_obstacle is True:
                return deja_vu, False
            print("Problème dans le calcul du chemin de la foule, sortie introuvable")
            return [(i, j)]
        else:
            deja_vu.append(file_attente[0])

        is_sortie, voisins = check_entourage(file_attente[0][0], file_attente[0][1], tab_sortie,
                                             foule=foule_en_obstacle)
        if is_sortie[0] is True:  # Est-ce qu'on a trouvé la sortie ?
            sortie = True
            coord_sortie = is_sortie[1]
            peres.append([is_sortie[1], file_attente[0]])
        # Sinon on continue de chercher
        for voisin in voisins:
            if voisin not in deja_vu:
                deja_vu.append(voisin)
                file_attente.append(voisin)
                peres.append([voisin, file_attente[0]])
        file_attente.pop(0)
    chemin_sortie = remonter_peres(peres, coord_sortie, (i, j))
    if foule_en_obstacle is False and len(tab_sortie) == 0:
        for i in range(0, len(chemin_sortie)):
            dico_memoisation_chemin[(chemin_sortie[i][0], chemin_sortie[i][1])] = chemin_sortie[i:]
    if foule_en_obstacle is True:
        return chemin_sortie, True
    return chemin_sortie


def remonter_peres(peres, coord_sortie, coord_entree):
    """Fonction supplémentaire à la fonction de recherche du plus court chemin, permet de remonter le dictionnaire des
    peres pour afficher la sortie"""
    fin = False  # tant qu'on n'a pas tout remonté
    chemin = [coord_sortie]
    chemin_actuel = coord_sortie
    while fin is False:
        for lien in peres:
            if lien[0] == chemin_actuel:
                if lien[1] == coord_entree:
                    fin = True
                else:
                    chemin_actuel = lien[1]
                    chemin.append(lien[1])
    return chemin


def valeur_absolu(x):
    if x >= 0:
        return x
    return -x
```

```python
def str_to_tuple(ligne, type):
    """Permet de convertir un tuple sous forme de string en reel tuple"""
    mot = ligne[1:len(ligne) - 1]  # on enlève les ()
    if type == "int":
        return tuple(
            map(float, mot.split(', ')))  # On sépare les deux nombres et on les convertit en int, puis en tuple
    return [mot[1], float(mot.split(' ')[1])]


def ajout_foule(i, j):
    """Cette fonction s'occupe de répartir la foule sur le point sélectionné lors du mode édition"""
    liste_foule_ajoutee = []  # Tableau contenant toutes les cases de foule générées pour la fonction affichage
    for k in range(-portee_ajout_foule + 1, portee_ajout_foule):
        for l in range(-portee_ajout_foule + 1, portee_ajout_foule):
            if (i + k, j + l) not in dico_carte:
                if randrange(0, valeur_absolu(k) + 1) == 0:
                    dico_carte[(i + k, j + l)] = ["F", foule_pourcentage_depart]
                    liste_foule_ajoutee.append([i + k, j + l])
    affichage(liste_foule_ajoutee)


def enregistrer_carte(dico):
    """Enregistre l'état de toutes les cellules de la carte dans un fichier temporaire"""
    fichier = open("cartes/Slot modification.txt", 'w')
    fichier.write(f"Size_window={width};{height}")
    fichier.write(f"\nSize_tiles={cellules_largeur};{cellules_hauteur}")
    for key, item in dico.items():
        fichier.write(f"\n{key};{(item[0], round(item[1],3))}")
    fichier.close()


def ouvrir_carte():
    """Charge le dictionnaire en charge des cellules depuis un fichier slot.txt"""
    fichier = open(f"{slot}", 'r')
    for ligne in fichier:
        ligne = ligne.strip('\n')
        # On vérifie la dimension de la carte/cellules
        if ligne[5] == "w":
            ligne = ligne.split("=")[1].split(";")
            if int(ligne[0]) != width or int(ligne[1]) != height:
                print("Mauvaise dimension de carte")
        elif ligne[5] == "t":
            ligne = ligne.split("=")[1].split(";")
            if int(ligne[0]) != cellules_largeur or int(ligne[1]) != cellules_hauteur:
                print("Mauvaise dimension des cellules")
        ######
        else:  # C'est les bonnes dimensions, on charge la carte
            ligne = ligne.split(";")
            dico_carte[str_to_tuple(ligne[0], 'int')] = str_to_tuple(ligne[1], str)
    fichier.close()


def modifier_carreau(x, y):
    """Cette fonction sert à éditer les cellules de la carte pendant le mode édition"""
    j = (cellules_largeur * x) // width  # Colonne de la carte
    i = (cellules_hauteur * y) // height  # Ligne de la carte
    dico_chemin.clear()
    dico_memoisation_chemin.clear()
    if bool_edition['obstacle'] is True:
        dico_carte[(i, j)] = ['O', 0]
    elif bool_edition['sortie'] is True:
        dico_carte[(i, j)] = ['S', 0]
    elif bool_edition['foule'] is True:
        ajout_foule(i, j)
    else:
        if (i, j) in dico_carte:
            dico_carte.pop((i, j))
        else:
            print("Erreur, il n'y a rien à faire sur cette case")
    affichage([(i, j)])


def edition(lettre):
    """Cette fonction edition permet de gérer l'activation du mode edition et modes d'ajouts comme les obstacles/sorties
    sans conflits entre eux"""
    # 112:p, 115:s, 111:o
    numero_lettre = {
        111: 'obstacle',  # lettre o
        112: 'edition',  # lettre p
        115: 'sortie',  # lettre s
        102: 'foule',  # lettre f
        13: 'demarrer_sims'  # lettre entrer
    }
    if lettre == 112:  # Cette partie permet d'activer ou de désactiver le mode edition
        if bool_edition['edition'] is False:
            bool_edition['edition'] = True
            print("Mode edition activé")
        else:
            for key in bool_edition.keys():  # Si on désactive le mode edition, on désactive tous les modes d'ajouts
                bool_edition[key] = False
            check_sortie()
            determiner_dico_distance_euclidienne()
            determiner_dico_pcbs()
            print("Mode edition désactivé")
    else:  # Cette partie permet de vérifier si un autre mode d'ajout n'est pas déjà activé pour éviter un conflit
        if bool_edition['edition'] is True:
            autorisation = True
            for key in bool_edition.keys():
                if key != 'edition' and key != numero_lettre[lettre] and bool_edition[key] is True:
                    print(f"Désactivez le mode ajout {key} avant")
                    autorisation = False
            if autorisation is True:  # Aucun autre mode n'est activé, on peut donc l'activer ou le désactiver
                if bool_edition[numero_lettre[lettre]] is True:
                    bool_edition[numero_lettre[lettre]] = False
                    print(f"Mode ajout {numero_lettre[lettre]} désactivé")
                else:
                    bool_edition[numero_lettre[lettre]] = True
                    print(f"Mode ajout {numero_lettre[lettre]} activé")
        else:  # Un autre mode est activé, on prévient l'utilisateur
            print("Activez le mode edition avant!")
```

```python
def deplacement_foule(i, j, osi):
    """Deplace les cases foule"""
    est_sortie, entourage = check_entourage(i, j, sorties)

    # On calcule le taux de densité qui peut se déplacer
    densite_depart = coef_deplacement_densite[int(calcul_densite(entourage))]

    # Si on peut déplacer plus de densité que la case en contient
    if densite_depart > dico_carte[i, j][1]:
        densite_depart = dico_carte[i, j][1]
    densite_deplacable = densite_depart

    if est_sortie[0] == True:  # Si la sortie est à proximité, on va directement se diriger à l'intérieur
        if densite_sortie_imposee is True:
            pourcentage_sortie = densite_sortie
        else:
            pourcentage_sortie = densite_deplacable

        if dico_carte[(i, j)][1] <= pourcentage_sortie:
            del dico_carte[(i, j)]
        else:
            dico_carte[(i, j)][1] -= pourcentage_sortie
        affichage([(i, j)])
    else: # Si la sortie n'est pas à proximité
        # On détermine tous les voisins qui sont plus proches de la sortie que la case actuelle, puis on répartit se
        # les paramètres du programme la densité sur ces voisins.

        voisins = []
        # Les voisins ici, sont les voisins qui sont plus proches de la sortie que la case elle-même
        for voisin in entourage:
            if dico_distance_euclidienne[voisin] < dico_distance_euclidienne[(i, j)]:
                voisins.append([voisin, dico_distance_euclidienne[voisin]])

        # On calcule la densité totale libre des voisins
        densite_totale_libre_voisins = 0
        for voisin in voisins:
            if voisin[0] in dico_carte:
                densite_totale_libre_voisins += (1 - dico_carte[voisin[0]][1])
            else:
                densite_totale_libre_voisins += 1

        # Si la densité qui peut se déplacer est plus grande que la densité totale libre, des voisins, on déplace la
        if densite_deplacable >= densite_totale_libre_voisins:
            for voisin in voisins:
                dico_carte[voisin[0]] = ['F', 1]
            dico_carte[(i,j)][1] -= densite_totale_libre_voisins
            if dico_carte[(i,j)][1] < 0:
                print('Erreur de densité lors du déplacement d'une case')
        else: # Si on peut déplacer toute la densité dans les voisins, on le fait progressivement
            dico_carte[(i,j)][1] -= densite_deplacable
            voisins = tri_foule(voisins) # On trie les voisins pour remplir ceux plus proche de la sortie d'abord
            densite_toleree_locale = densite_maximale_supportee # On crée une densité tolérée locale, car dans certa.
            while densite_deplacable > 0:
                round(densite_deplacable, 4)
                for voisin in voisins:
                    if densite_deplacable > 0: # Sécurité si jamais on déplace trop dans un même tour de boucle whi.
                        # On récupère la densité du voisin
                        if voisin[0] in dico_carte:
                            densite_voisin = dico_carte[voisin[0]][1]
                        else:
                            densite_voisin = 0

                        if densite_voisin < densite_toleree_locale: # Si on peut déplacer une densité dans ce voisin
                            densiter_a_deplacer = densite_toleree_locale - densite_voisin

                            # Si l'espace disponible est plus grand que l'espace requis
                            if densite_deplacable <= densiter_a_deplacer:
                                densiter_a_deplacer = densite_deplacable
                            densite_deplacable -= densiter_a_deplacer
                            # ensuite on actualise les données des cases
                            if voisin[0] in dico_carte:
                                dico_carte[voisin[0]][1] -= densiter_a_deplacer
                            else:
                                dico_carte[voisin[0]] = ['F', densiter_a_deplacer]
                densite_toleree_locale += 0.01

        if dico_carte[(i, j)][1] == 0:
            del dico_carte[i, j]
        elif dico_carte[(i, j)][1] < 0:
            print("erreur densites !")

    # Cette partie du code regroupe tous les voisins dans un tableau, afin d'actualiser l'affichage de ceux-ci
    for x in range(0, len(voisins)):
        voisins[x] = voisins[x][0]
    voisins.append((i, j))
    affichage(voisins)


def afficher_chemin(i, j):
    """Permet d'afficher le chemin en jaune qu'un 'carré' foule va poursuivre"""
    chemin_sortie = bfs((i, j))
    for case in chemin_sortie:
        dico_chemin[(case[0], case[1])] = 'J'
    affichage(dico_chemin)

# On importe les paramètres du programme
coef_deplacement_densite, densite_maximale_supportee, foule_pourcentage_depart, densite_sortie = importer_parametres
ouvrir_carte()  # On charge le dictionnaire qui contient les informations des cases de la carte
check_sorties()  # On charge les sorties
determiner_dico_distance_euclidienne()  # On charge le dictionnaire des distances euclidienne
determiner_dico_pcbs()  # On charge le dictionnaire qui contient tous les parcours en largeur
affichage_console()  # Affichage du menu
affichage([(0, 0)], tout=True)  # On affiche toute la carte pour la charger

while 1:
    for event in pygame.event.get():
        # print(event)
        if event.type == pygame.MOUSEBUTTONUP:
            x, y = pygame.mouse.get_pos()
            if bool_edition['edition'] is True:
                modifier_carreau(x, y)
        if event.type == pygame.KEYDOWN:
            # print(event)
            if event.key == 27:
                sys.exit()
            if event.key == 112 or event.key == 115 or event.key == 111 or event.key == 102:
                edition(event.key)
            if event.key == 110:
                print("Sauvegarde!")
                enregistrer_carte(dico_carte)
            if event.key == 13:  # entrer
                if demarrer_simulation is False:
                    if bool_edition['edition'] is False:
                        # verification_carte()
                        demarrer_simulation = True
                        definir_densite_foule_depart(foule_pourcentage_depart)
                        print("Simulation démarrée!")
                        nombre_tours_simulation = 0
                    else:
                        print("Désactivez le mode édition avant de démarrer la simulation!")
                else:
                    demarrer_simulation = False
                    print("Simulation stoppée!")
            if event.key == 99:  # c
                print('Fonctionnalité désactivée')
        if event.type == pygame.QUIT:
            enregistrer_carte(dico_carte)
            sys.exit()

    # Déplacement de la foule
    horloge = time.perf_counter()
    if horloge >= temps_ancien_deplacement + seconde_par_deplacement and (
            demarrer_simulation is True):  # Pour afficher un déplacement par pas de temps
        temps_ancien_deplacement = horloge
        dico_chemin.clear()
        liste_foule_a_deplacer = []  # Liste chacun des 'carrés' foule à faire bouger, boucle imbriquée pour obteni
        # cordonnées de toutes les cases foule
        total = 0
        for i in range(0, cellules_hauteur):
            for j in range(0, cellules_largeur):
                if (i, j) in dico_carte:
                    if dico_carte[(i, j)][0] == 'F':
                        # On arrondi la densité au centième près, et si l'arrondi est 0, on supprime cette foule

                        dico_carte[(i, j)][1] = round(dico_carte[(i, j)][1], 5)
                        if dico_carte[(i, j)][1] == 0:
                            del dico_carte[(i, j)]
                            affichage([(i, j)])
                        else:
                            liste_foule_a_deplacer.append([i, j])
        if len(liste_foule_a_deplacer) == 0:
            print(f"Nombre de tours : {nombre_tours_simulation}")
            print("Simulation terminée!")
            demarrer_simulation = False
            #calcul_statistiques(True)
        #else:
            #calcul_statistiques()
        nombre_tours_simulation += 1

        # On calcule la distance de chaque case foule avec la sortie, puis on le range par ordre croissant
        # On peut se permettre de faire un parcours en largeur de chaque foule juste pour connaître la distance
        # avec la sortie, car ce calcul a déjà été fait, il est mémoisé.
        for k in range(0, len(liste_foule_a_deplacer)):
            case = liste_foule_a_deplacer[k]
            liste_foule_a_deplacer[k] = [(case[0], case[1]),
                                         distance_euclidienne(case[0], case[1], dico_pcbs[(case[0], case[1])])]
        # On trie selon les cases foule les plus proches de la sortie
        liste_foule_a_deplacer = tri_foule(liste_foule_a_deplacer)
        # Puis on les déplace
        for case in liste_foule_a_deplacer:
            deplacement_foule(case[0][0], case[0][1], case[1])

    pygame.display.flip()
```