ENSAM – Casablanca **Année Universitaire 2025-2026**
Département Génie Informatique **Pr. Soukaina Bouhsissin**

Intelligence Artificielle

TP 1 : Agent Réflexe Simple (Robot Aspirateur)

Objectifs du TP

- Comprendre la boucle Perception → Décision → Action
- Implémenter un agent réflexe simple (sans mémoire) qui nettoie une grille 3×3
- Mesurer la performance d'un agent (% de cases propres, nombre d'actions)
- Analyser le comportement d'un agent autonome

Vous allez programmer un **robot aspirateur** qui doit nettoyer une grille 3×3 contenant des cases sales et propres.

Représentation de la grille:

- 0 = Case propre, 1 = Case sale, R = Position du robot

Exemple :

R 0 1
1 1 0
0 1 1

Le robot doit suivre des règles réflexes simples pour nettoyer efficacement.

PARTIE A : Environnement et Utilitaires

Copiez le code suivant dans votre notebook Python :

```
import random
def init_grille(n=3, p=0.5, seed=0):

    """
    Initialise une grille n×n avec p% de cases sales.
    Args:
        n: Taille de la grille
        p: Probabilité qu'une case soit sale (0.0 à 1.0)
        seed: Graine aléatoire pour la reproductibilité
    Returns:
        Liste de listes représentant la grille
    """
    rnd = random.Random(seed)
    return [[1 if rnd.random() < p else 0 for _ in range(n)] for _ in range(n)]
```

```
def tout_propre(G):
    """Vérifie si toutes les cases sont propres."""
    return all(c == 0 for row in G for c in row)

def in_bounds(i, j, n):
    """Vérifie si la position (i,j) est dans la grille n×n."""
    return 0 <= i < n and 0 <= j < n

def print_grille(G, pos=None, label=""):
    """
    Affiche la grille.
    Args:
        G: La grille à afficher
        pos: Position du robot (i, j) ou None
        label: Titre à afficher
    """
    if label:
        print(label)
    for i, row in enumerate(G):
        line = []
        for j, c in enumerate(row):
            if pos is not None and (i, j) == pos:
                line.append("R")
            else:
                line.append(str(c))
        print(" ".join(line))
    print()
```

1. Créez une grille 3×3 avec 60% de cases sales
2. Choisissez une position initiale (x, y) au hasard pour le robot
3. Affichez la grille avec print_grille

Code à compléter :

```
# TODO: Initialisez les paramètres

n =          # Taille de la grille

p =         # % de saleté
```

```
seed =        # Pour la reproductibilité

# TODO: Créez la grille

G = init_grille(n=n, p=p, seed=seed)

# TODO: Position initiale aléatoire du robot

x, y = ,

# TODO: Affichez la grille

print_grille( , , label="Grille initiale (R = robot) :")
```

Questions :

Q1.1 Quelle est la grille générée avec seed=1 ? Dessinez-la.

Q1.2 Combien de cases sales y a-t-il dans cette grille ?

Q1.3 Quelle est la position initiale du robot ?

Q1.4 Testez avec seed=2 et seed=3. Les grilles changent-elles ? Pourquoi utilise-t-on une graine (seed) ?

Q1.5 Que se passe-t-il si vous mettez p=1.0 ? Et p=0.0 ?

PARTIE B : Règles Réflexes

Principe de l'agent réflexe : L'agent suit des règles simples

RÈGLE 1 : SI la case courante est sale (=1)

ALORS ASPIRER (nettoyer la case)

RÈGLE 2 : SINON se déplacer selon un cycle :

DROITE → BAS → GAUCHE → HAUT → DROITE → ...

(en évitant les bords de la grille)

Code à compléter :

```
# Cycle de déplacement
cycle = ["DROITE", "BAS", "GAUCHE", "HAUT"]
def choisir_action(G, x, y, step):
   # Règle 1 : Aspirer si la case est sale
```

```
    if G[x][y] == 1:
        return ""
    # Règle 2 : Suivre le cycle de déplacement
    for t in range(4):
        a = cycle[(step + t) % 4]
        nx, ny = x, y
        if a == "HAUT":
        if a == "BAS":
        if a == "GAUCHE":
        if a == "DROITE":
        # Vérifier si la nouvelle position est valide
        if in_bounds(nx, ny, len(G)):
            return a
    return "RIEN"
```

Appliquer une Action

Complétez la fonction appliquer_action qui :

- Si l'action est "ASPIRER" → met la case courante à 0 (propre)
- Si l'action est un déplacement ("HAUT", "BAS", etc.) → déplace le robot (si possible)
- Retourne la nouvelle position (x, y) et l'action effectuée

Code à compléter :

```
def appliquer_action(G, x, y, action):
    # TODO: Si l'action est ASPIRER
    if action == "ASPIRER":
        # Nettoyez la case courante
        G[x][y] =
        return x, y, action

    # TODO: Si l'action est HAUT et que c'est possible
    if action == "HAUT" and in_bounds(x-1, y, len(G)):
        return

    # TODO: Complétez pour BAS
    if action == "BAS" and in_bounds(x+1, y, len(G)):
        return

    # TODO: Complétez pour GAUCHE
    if action == "GAUCHE" and in_bounds(x, y-1, len(G)):
```

```
    return

  # TODO: Complétez pour DROITE
  if action == "DROITE" and in_bounds(x, y+1, len(G)):
    return

  # Si aucune action n'est possible
  return
```

Questions :

Q2.1 Que fait exactement G[x][y] = 0 ?

Q2.2 Pourquoi vérifie-t-on in_bounds avant de se déplacer ?

Q2.3 Que se passe-t-il si le robot est dans un coin et essaie d'aller vers le bord ?

Q2.4 Testez manuellement : Si le robot est en (0, 0) et fait "HAUT", quelle est la nouvelle position ?

Q2.5 Tracez sur papier le parcours du robot pour les 5 premières actions avec la grille de Q1.

PARTIE C : Boucle d'Exécution et Performance

On va maintenant faire tourner le robot en boucle jusqu'à ce que :

- Soit **toute la grille soit propre**
- Soit on atteigne un **nombre maximum d'actions** (30 dans notre cas)

Code de simulation

```
# Paramètres de simulation
max_steps = 30
step = 0
actions = []
# Boucle d'exécution
while not tout_propre(G) and step < max_steps:
  # Le robot choisit une action
  a =
  # Le robot applique l'action
  x, y, a_eff =

  # Enregistrer l'action
  actions.append()
```

```
    step += 1
# Calcul de la performance
propres = sum()
taux = 100.0 *
# Affichage des résultats
print_grille(G, (x, y), label="Grille finale :")
print({
    "cases_propres": propres,
    "sur": n*n,
    "taux_%": round(taux, 1),
    "nb_actions": len(actions)
})
print("\nActions effectuées :", actions)
```