

# RAPPORT

## → AGENT INTELLIGENT: ←

### LIEN GITHUB DU PROJET:

[https://github.com/Sw0ocHCorp/Projet\\_Aspirateur\\_Intelligent](https://github.com/Sw0ocHCorp/Projet_Aspirateur_Intelligent)

### LIEN GDOC DU RAPPORT

(L'animation de l'Algorithme A\* ne s'affiche pas sur le PDF):

<https://docs.google.com/document/d/1BSx-PST6VklIIOJjtffa2h-f4AO1JjBiM7SUS8ISZ0/edit?usp=sharing>

## TP#1

### Q1=

RÉALISER UN SIMULATEUR DE L'ENVIRONNEMENT POUR L'ASPIRATEUR AVEC MESURE DES PERFORMANCES. VOTRE CODE DEVRA ÊTRE MODULAIRE DE SORTE À CE QUE LES CAPTEURS, ACTUATEURS ET CARACTÉRISTIQUE DE L'ENVIRONNEMENT=

#### - Organisation du Projet=

J'ai choisi d'adopter une méthode exploitant pleinement les propriétés de la POO. Ainsi, j'ai créé différentes classes, représentant chaque concepts / objets. Ces classes sont les suivantes:

- |                   |  |
|-------------------|--|
| + Agent           | → Permet de représenter l'Agent interagissant avec un Environnement 1D   |
| + Agent2D         | → Permet de représenter l'Agent interagissant avec un Environnement 2D   |
| + Classe          | → Permet de représenter les Salles de l'Environnement  |
| + AlgorithmeAStar | → Permet d'embarquer l'Algorithme A* afin de calculer le plus courts chemin possible pour nettoyer l'Environnement |
| + Environnement   | → Permet de représenter une configuration de salles, constituant l'Environnement dans lequel se trouve l'Agent     |

#### - Modélisation de l'Environnement=

##### + Environnement 1D=

```
PS C:\Users\nc1sr\OneDrive\Bureau\Cours L3IA\Agent Intelligent> & C:/Users/nc1sr/AppData/Local/Programs/Python/Python39-64/Scripts/python.exe C:/Users/nc1sr/AppData/Local/Programs/Python/Python39-64/Scripts/python.exe
Entrée la dimension de l'environnement (1D ou 2D)? 1D
Entrée le mode de sélection d'actions de l'agent (Random ou Memorisation ou Simple)? simple
Entrée le nombre de salles de l'environnement (1D)? 4
--> EPOCH# 0
Etat Initial de l'environnement E Z X D*
2 Salles à nettoyer
```

Pour modéliser l'environnement 1D, je demande la configuration de ce dernier que souhaite l'utilisateur (Dimension de l'environnement / Stratégie de prise de décision de l'Agent / Nombre de Salles). Le nombre de classes sales est aléatoire, sur celui-ci. Les Classes sales sont écrites en Rouge, la présence de l'Agent dans la pièce est indiqué avec le caractère `\*` après le nom de la salle dans laquelle il est présent. Cet environnement m'a UNIQUEMENT servi de support pour créer l'environnement 2D

+ Environnement 2D=

```
PS C:\Users\nclsr\OneDrive\Bureau\Cours_L3IA\Agent_Intelligent> & 'C:\Users\nclsr\AppData\Local\Programs\Python\Python39\python.exe' 'C:\Users\nclsr\AppData\Local\Programs\Python\Python39\Scripts\launcher' '54732' '--' 'c:\Users\nclsr\OneDrive\Bureau\Cours_L3IA\Agent_Intelligent\TD1\Envir
Entrée la dimension de l'environnement (1D ou 2D)? 2D
Entrée le mode de sélection d'actions de l'agent (Random ou Memorisation ou Simple)? memorisation
Entrée le nombre de salles de l'environnement (2D)? 9
Entrée la largeur de l'environnement (2D)? 3
Entrée le nombre maximum d'obstacles que vous voulez dans l'environnement (2D)? 1
Etat Initial de l'environnement
B N L
V C P*
K G X
2 Salles à nettoyer
```

Pour modéliser l'environnement 2D, je demande la configuration de ce dernier que souhaite l'utilisateur (Dimension de l'environnement / Stratégie de prise de décision de l'Agent / Nombre total de classe / Nombre de classes sur une même ligne / Nombre maximal d'obstacles dans l'Environnement). Pour la configuration des classes sales et le nombre réel d'obstacles, leurs nombres réels sont aléatoires([0, *valeur choisie*] pour les obstacles).

```
PS C:\Users\nclsr\OneDrive\Bureau\Cours_L3IA\Agent_Intelligent> & 'C:\Users\nclsr\AppData\Local\Programs\Python\Python39\python.exe' 'C:\Users\nclsr\AppData\Local\Programs\Python\Python39\Scripts\launcher' '58510' '--' 'c:\Users\nclsr\OneDrive\Bureau\Cours_L3IA\Agent_Intelligent\TD1\Envir
Entrée la dimension de l'environnement (1D ou 2D)? 2D
Entrée le mode de sélection d'actions de l'agent (Random ou Memorisation ou Simple)? memorisation
Entrée le nombre de salles de l'environnement (2D)? 7
Entrée la largeur de l'environnement (2D)? 3
Entrée le nombre maximum d'obstacles que vous voulez dans l'environnement (2D)? 0
--> EPOCH# 0
Etat Initial de l'environnement
G O K*
Z R D
C N N
3 Salles à nettoyer
```

De la même manière que pour l'environnement 1D, les classes sales sont représentées en Rouge, les Obstacles sont représentées par la lettre N noire. La présence de l'Agent dans la pièce est indiqué avec le caractère `\*` après le nom de la salle dans laquelle il est présent.

**ATTENTION=** Pour réaliser les Test de ces TDs on va utiliser directement les fonctions présentes dans les fichiers TestAgent.py et Test\_Agent.py afin de simplifier la configuration de l'environnement

- **Méthode d'évaluation de la performance=**

+ Algorithme A\* (Environnement 2D)=

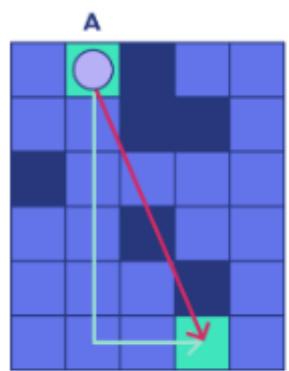


- On utilise cette méthode dans le cas de la Stratégie de prise de décision Random ou État Interne(mémorisation). Cet algorithme permet de trouver le plus court chemin entre 2 points dans un plan en minimisant la somme

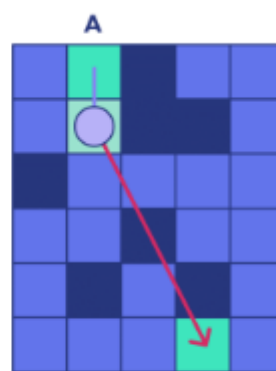
$$F = G + H \rightarrow \text{Avec } G = \text{distance entre le point et la position précédente}$$

$$\rightarrow \text{Avec } H = \text{distance entre le point et la destination}$$

Pour ce faire, on part de notre point de départ, on calcul la valeur  $F$  des points adjacents de notre position actuelle et on considère que, pour le point adjacent avec la valeur de  $F$  la plus faible, notre position est le point précédent. On fait les mêmes étapes pour les points adjacents jusqu'à arriver à la destination. Enfin, on retrouve le chemin optimal facilement en récupérant les points précédents, de manière récursive, de la destination au point de départ.



Manhattan distance = 8  
Pythagore distance =  $45^{1/2}$



Distance parcourue = 1  
Pythagore distance =  $34^{1/2}$

L'intérêt de cet Algorithme est qu'il permet de contourner des obstacles, de manière optimisée.

- **Organisation du Projet=**

+ Dossier TD1=

Dans ce dossier, nous avons un dossier "img\_rapport\_TD1" dans lequel se trouvent des images permettant d'illustrer les explications du rapport du TD1.

Le fichier, Tests\_Agent permet de réaliser tous les tests à réaliser pour répondre aux questions du TD1

**ATTENTION= Pour les tests, on va se concentrer sur l'environnement 2D car L'Environnement 1D est bien trop simpliste. Ce dernier m'a servi de support pour construire l'Environnement 2D, il n'est donc pas destiné à être utilisé**

- + Dossier TD2=

Le fichier TestAgent, permet de réaliser tous les tests à réaliser pour répondre aux questions du TD2

## Q2=

**RÉALISER UN AGENT REFLEXE SIMPLE POUR CE SIMULATEUR. EXECUTER-LE DANS LE SIMULATEUR POUR TOUTES LES CONFIGURATIONS INITIALES POSSIBLES DE LA SALISSURE ET POSITION DE L'ASPIRATEUR. NOTER LA NOTE DE PERFORMANCE POUR CHAQUE CONFIGURATION ET SA MOYENNE GENERALE=**

- **Test de l'Agent avec réflexes simples=**

- ENVIRONNEMENT 2D=

Pour tester ce dernier dans différentes configurations, on va fixer un cadre, de manière à normaliser tous les tests:

- + 9 pièces / On ne prend pas un environnement trop important pour lancer les tests afin d'avoir des tests rapides
- + 3 pièces par lignes
- + Pas d'obstacles
- + Nombre de pièces sales variables
- + Agent avec réflexes simple= Stratégie précise définie à l'avance
- + Agent avec détecteur de positions & bord de l'environnement
- + Position Initiale de l'Agent variable
- **TEST#1.1** → Agent en (0, 0) → Pièces sales [(0, 1), (2, 0), (2, 2)]  
 → 9 pièces au total, 3 par lignes  
 L'Agent nettoie tout l'environnement en 12 actions alors que le nombre minimum d'actions optimisées sont de 9.  
 L'agent obtient donc une note de 8.5/10  

$$\text{car } 10 - (12 - 9) * 0.5 = 8.5$$
- **TEST#1.2** → Agent en (0, 2) → Pièces sales [(0, 1), (2, 0), (2, 2)]  
 → 9 pièces au total, 3 par lignes  
 L'Agent nettoie tout l'environnement en 11 actions alors que le nombre minimum d'actions optimisées sont de 9.  
 L'agent obtient donc une note de 9/10  

$$\text{car } 10 - (11 - 9) * 0.5 = 9$$
- **TEST#1.3** → Agent en (1, 1) → Pièces sales [(0, 1), (0, 2), (2, 1), (2, 2)]  
 → 9 pièces au total, 3 par lignes  
 L'Agent nettoie tout l'environnement en 12 actions alors que le nombre minimum d'actions optimisées sont de 9.  
 L'agent obtient donc une note de 8.5/10
- **TEST#1.4** → Agent en (0, 2) → Pièces sales [(2, 0)]

→ 9 pièces au total, 3 par lignes

L'Agent nettoie tout l'environnement en 9 actions alors que le nombre minimum d'actions optimisées sont de 5.

L'agent obtient donc une note de 8/10

$$\text{car } 10 - (9 - 5) * 0.5 = 8$$

### Q3=

**CONSIDÉREZ UNE VERSION MODIFIÉE DE L'ENVIRONNEMENT OU L'AGENT EST PÉNALISÉ D'UN POINT POUR CHAQUE MOUVEMENT. EST-CE QU'UN AGENT SIMPLE PEUT-ÊTRE PARFAITEMENT RATIONNEL POUR CET ENVIRONNEMENT=**

- **Agent Rationnel=**

Un Agent Rationnel est une Agent prenant toujours les actions les plus optimisées dans chacuns des états de l'Environnement par lequel il "passe".

Selon moi, l'Agent Réflexes Simple n'est, pas un Agent Rationnel, en effet, il ne possède pas de processus de prise de décisions "intelligente". C'est le programmeur qui donne à l'agent, sa stratégie, trajectoire. Elle est donc statique(ne change pas) ainsi, on ne peut pas s'adapter aux différentes positions de salissure des pièces de l'environnement.

- **Test de Performances des différents mode d'Agent (Etat Interne / Décisions Randoms) comparé à l'Agent à Réactions Simples=**

Dans le test que j'ai lancé, je me suis basé sur le test

- **TEST#1.3** → Agent en (1, 1) → Pièces sales [(0, 1), (0, 2), (2, 1), (2, 2)]  
→ 9 pièces au total, 3 par lignes

On obtient, pour l'Agent à Réactions Simple, une note de 8.5/10

```
> Action n° 12 = |.^|
M O S*
D A T
D P X
Note de Performance= 8.5 / 10
Note moyenne= 8.5 / 10
Note max= 8.5 / 10
-----
-----
```

On obtient, pour l'Agent à prise de décision random une note de 6/10

```
> Action n° 17 = |.^|
Y M C
H P R
A F* K
Note de Performance= 6.0 / 10
Note moyenne= 6.0 / 10
Note max= 6.0 / 10
```

On obtient, pour l'Agent à état interne, une note de 7.5/10

```

> Action n° 14 = |.^|
K A I
R S M
V W F*
Note de Performance= 7.5 / 10
Note moyenne= 7.5 / 10
Note max= 7.5 / 10

```

On a, des notes plus faibles pour les autres méthodes de prises de décisions, car on utilise une prise de décision random, même pour la méthode états internes. Cependant, cette dernière méthode peut être utilisée dans un processus d'Apprentissage par Renforcement, permettant d'améliorer drastiquement la performance de l'Agent

## Q4=

### RÉALISEZ UN AGENT RÉFLEXE AVEC ETAT. EST CE QU'IL PEUT ETRE PARFAITEMENT RATIONNEL POUR L'ENVIRONNEMENT DE LA QUESTION#3=

#### - **Agent avec États Internes=**

Un Agent avec Etat Interne est un Agent stockant dans une mémoire l'ensemble / l'historique des états qu'il à visité, ainsi que les actions possibles pour chacun d'entre eux. Les méthodes de Prises de Décisions de ce type d'Agent peuvent être diverses et variées.

#### - **Stratégie Aléatoire=**

L'action effectuée par l'Agent, dans un état de l'environnement est prise de manière aléatoire

#### - **Stratégie Pseudo Aléatoire=**

L'action effectuée par l'Agent, dans un état de l'environnement est prise de manière aléatoire. Cependant, l'action effectuée lors de la dernière visite de cet état ne peut pas être choisie.

#### - **Stratégie Par Apprentissage=**

L'Agent prend l'action permettant de se trouver dans le meilleur état adjacent. Implique le calcul de Valeurs d'Etats & d'Actions via l'utilisation des équations de Bellman(Apprentissage par Renforcement).

Un Agent Réflexe avec États internes peut être parfaitement Rationnel, pour l'Environnement. Cependant, cela dépend de la "stratégie" de prise de Décision qui est implémentée. En effet, si on utilise une Stratégie Aléatoire ou même Pseudo-Aléatoire, l'Agent ne peut pas être qualifié de Rationnel car il y a une trop faible probabilité qu'il choisisse la meilleure action pour chacun des états dans lesquels il se trouve. Comme indiqué précédemment, la méthode de prise de décision qui selon moi, permettrait de produire un Agent Rationnel, serai l'Apprentissage par Renforcement. Cela permettrait de mapper l'environnement et qu'il sache, dans n'importe quel état de l'environnement, l'action à prendre pour atteindre un état plus intéressant.

```

--> EPOCH# 0
Etat Initial de l'environnement
K A I
R S* M
V W F
4 Salles à nettoyer
Nombre d'actions Minimum pour tout Nettoyer= 9
> Action n° 1 = v
K A I
R S M
V W* F
> Action n° 2 = |.^|
K A I
R S M
V W* F
> Action n° 3 = ^
K A I
R S* M
V W F
> Action n° 4 = ^
K A* I
R S M
V W F
> Action n° 5 = |.^|
K A* I
R S M
V W F
> Action n° 6 = <-
K* A I
R S M
V W F
> Action n° 7 = v
K A I
R* S M
V W F
> Action n° 8 = ^
K* A I
R S M
V W F
> Action n° 9 = ->
K A* I
R S M
V W F
> Action n° 10 = ->
K A I*
R S M
V W F
> Action n° 11 = |.^|
K A I*
R S M
V W F
> Action n° 12 = v
K A I
R S M*
V W F
> Action n° 13 = v
K A I
R S M
V W F*
> Action n° 14 = |.^|
K A I
R S M
V W F*
Note de Performance= 7.5 / 10
Note moyenne= 7.5 / 10
Note max= 7.5 / 10

```

### **Q5=**

#### **CONSIDÉREZ UNE VERSION ALTERNATIVE DE L'ENVIRONNEMENT DANS LEQUEL L'AGENT NE CONNAÎT PAS LA TOPOGRAPHIE DE L'ENVIRONNEMENT AINSI QUE LA SALISSURE DES PIÈCES=**

Lorsque j'ai développé l'Agent ainsi que ses interactions avec l'Environnement, j'ai déjà pris en compte cela, dans le sens où, mon Agent possède un "capteur d'obstacle / mur", ainsi, pour chaque pièce de l'environnement où il se trouve, il est capable de savoir si les pièces adjacentes existent ou s'il est au bord de l'environnement. J'ai suivi, les indications du sujet "initial": *La topographie de l'environnement est connue a priori par l'agent, mais la distribution de la salissure et la position initiale de l'agent ne le sont pas.*

### **Q6=**

#### **EST CE QU'UN AGENT RÉFLEXE SIMPLE PEUT ÊTRE PARFAITEMENT RATIONNEL POUR L'ENVIRONNEMENT DE LA QUESTION#5=**

L'Agent avec Réflexes Simples ne peut pas être Rationnel car, tout comme pour la Q#4, il ne s'adapte pas en fonction de ses observations sur l'environnement, il exécute simplement, la stratégie définie par le développeur(stratégie statique). De plus, le caractère Randomisé va induire des potentielles mauvaises prises de décisions. De plus, il y a tellement d'actions possibles que peut prendre l'Agent durant sa tâche, que la probabilité de prendre toujours la bonne action dans chacuns des états par lesquels il passe est très faible. La performance d'un tel Agent est donc très variable. On ne peut donc pas dire que ce type d'Agent puisse être Rationnel.

### **Q7=**

#### **EST CE QU'UN AGENT RÉFLEXE SIMPLE AVEC UNE PRISE DE DÉCISION RANDOMISÉE PEUT FAIRE MIEUX QU'UN AGENT RÉFLEXE SIMPLE=**

Un Agent Randomisé peut cependant faire mieux qu'un Agent avec Réflexe Simple en configurant l'environnement pour piéger l'Agent avec Réflexe Simple. Par exemple, on pourrait le configurer comme suit



```

--> EPOCH# 0
Etat Initial de l'environnement
P A D*
B Y T
C E S
1 Salles à nettoyer
Nombre d'actions Minimum pour tout Nettoyer= 2
> Action n° 1 = <-
P A* D
B Y T
C E S
> Action n° 2 = <-
P* A D
B Y T
C E S
> Action n° 3 = v
P A D
B* Y T
C E S
> Action n° 4 = ->
P A D
B Y* T
C E S
> Action n° 5 = ->
P A D
B Y T*
C E S
> Action n° 6 = |.^|
P A D
B Y T*
C E S
Note de Performance= 8.0 / 10
Note moyenne= 8.0 / 10
Note max= 8.0 / 10
-----
--> EPOCH# 0
Etat Initial de l'environnement
Z W K*
G B Y
Q H F
1 Salles à nettoyer
Nombre d'actions Minimum pour tout Nettoyer= 2
> Action n° 1 = v
Z W K
G B Y*
Q H F
> Action n° 2 = |.^|
Z W K
G B Y*
Q H F
Note de Performance= 10 / 10
Note moyenne= 10.0 / 10
Note max= 10.0 / 10

```

Si on positionne une seule pièce sale “à côté” du chemin de l’Agent avec Réflexes Simples, on va l’obliger à faire un détour pour la nettoyer. Cependant, l’Agent Randomisé pourra, avec de la chance, choisir de se déplacer directement vers la salle en question et, la nettoyer plus rapidement. De plus, dans cette configuration, l’Agent Randomisé peut uniquement choisir l’action GAUCHE ou BAS, ce qui augmente nos chances(1/2) de prendre la bonne action car on est dans un coin de l’Environnement

## **Q8=**

### **EXISTE-T-IL DES CONFIGURATIONS D'ENVIRONNEMENTS DANS LESQUELS L'AGENT RANDOMISÉ EST "PIRE" QUE L'AGENT AVEC RÉFLEXES SIMPLES=**

Il existe de nombreuses configurations d'environnements dans lequel l'Agent Randomisé est pire que l'Agent avec Réflexes Simples:

- *Environnement avec beaucoup de pièces:*

En effet, plus on a de pièces dans l'environnement, plus l'Agent devra effectuer d'actions pour nettoyer toutes les pièces. Ainsi, l'Agent randomisé aura beaucoup plus de chance de prendre des mauvaises action (retour en arrière / blocage entre 2 pièces), tandis que l'Agent avec Réflexes Simples suivra une Stratégie précise qui, logiquement, essaie de minimiser le nombre d'actions pour atteindre tous les états(en passant par chacun d'eux)

- *Environnement avec toutes les pièces sales:*

En effet, l'Agent avec Réflexes simple va passer UNE SEULE FOIS par chacun des états de l'environnement et les nettoyer. Pour ce qui est de l'Agent Randomisé, il suffit qu'il prenne UNE SEULE mauvaise action(retour sur une salle précédemment nettoyée pour que ce dernier soit moins performant. C'est la faiblesse caractéristique de la prise de décision randomisée.

Il existe beaucoup d'autres configurations induisant des performances inférieures pour l'Agent randomisé, car les performances de ce dernier sont très aléatoires comparé à l'Agent avec Réflexes Simples.

Je n'ai pas choisi une des 2 configurations explicités précédemment pour améliorer la lisibilité de la capture d'écran. Voici la configuration choisi:

```

--> EPOCH# 0
Etat Initial de l'environnement
M W Q*
B S N
V A C
1 Salles à nettoyer
Nombre d'actions Minimum pour tout Nettoyer= 4
> Action n° 1 = <-
M W* Q
B S N
V A C
> Action n° 2 = <-
M* W Q
B S N
V A C
> Action n° 3 = v
M W Q
B* S N
V A C
> Action n° 4 = |.^|
M W Q
B* S N
V A C
Note de Performance= 10 / 10
Note moyenne= 10.0 / 10
Note max= 10.0 / 10
-----
--> EPOCH# 0
Etat Initial de l'environnement
O Y C*
F F B
Q U D
1 Salles à nettoyer
Nombre d'actions Minimum pour tout Nettoyer= 4
> Action n° 1 = v
O Y C
F F B*
Q U D
> Action n° 2 = v
O Y C
F F B
Q U D*
> Action n° 3 = ^
O Y C
F F B*
Q U D
> Action n° 4 = v
O Y C
F F B
Q U D*
> Action n° 5 = <-
O Y C
F F B
Q U* D
> Action n° 6 = <-
O Y C
F F B
Q* U D
> Action n° 7 = ->
O Y C
F F B
Q U* D
> Action n° 8 = <-
O Y C
F F B
Q* U D
> Action n° 9 = ^
O Y C
F* F B
Q U D
> Action n° 10 = |.^|
O Y C
F* F B
Q U D
Note de Performance= 7.0 / 10
Note moyenne= 7.0 / 10
Note max= 7.0 / 10

```

Dans cette configuration, j'aide fortement l'Agent avec Réflexes Simples car je met la pièce sale sur son chemin directe, ainsi, il prendra forcément le plus petit nombre d'actions, cependant, ce n'est très probablement pas le cas de l'Agent Randomisé du fait de ses potentielles mauvaises prises d'actions

## Q9=

### L'AGENT AVEC ETAT INTERNE PEUT IL FAIRE MIEUX QUE L'AGENT AVEC RÉFLEXES SIMPLES=

On va réutiliser la même configuration que la Q#7 car on va désavantager l'Agent avec Réflexes Simples

```
--> EPOCH# 0
Etat Initial de l'environnement
E X C*
P D L
S W E
1 Salles à nettoyer
Nombre d'actions Minimum pour tout Nettoyer= 2
> Action n° 1 = <-
E X* C
P D L
S W E
> Action n° 2 = <-
E* X C
P D L
S W E
> Action n° 3 = v
E X C
P* D L
S W E
> Action n° 4 = ->
E X C
P D* L
S W E
> Action n° 5 = ->
E X C
P D L*
S W E
> Action n° 6 = |.^|
E X C
P D L*
S W E
Note de Performance= 8.0 / 10
Note moyenne= 8.0 / 10
Note max= 8.0 / 10
-----
--> EPOCH# 0
Etat Initial de l'environnement
U L D*
P M T
I N S
1 Salles à nettoyer
Nombre d'actions Minimum pour tout Nettoyer= 2
> Action n° 1 = <-
U L* D
P M T
I N S
> Action n° 2 = ->
U L D*
P M T
I N S
> Action n° 3 = v
U L D
P M T*
I N S
> Action n° 4 = |.^|
U L D
P M T*
I N S
Note de Performance= 9.0 / 10
Note moyenne= 9.0 / 10
Note max= 9.0 / 10
```

Comme indiqué précédemment, l'Agent avec Réflexes Simples est désavantagé. On profite également de la particularité de mon Agent avec Etat Interne. Ce dernier, mémorise les actions qui peuvent être prises dans les différents états de l'environnement qu'il visite, de plus il mémorise la dernière action prise pour chaque état, lorsque ce dernier retournera dans cet état, il enlèvera temporairement cette action de son pool d'actions possible. De cette manière on peut contrôler un minimum, la prise d'action aléatoire de cet agent (on réduit les possibilités de blocage entre 2 états)

1ère Visite de la pièce R

0 = GAUCHE  
1 = DROITE  
2 = HAUT  
3 = BAS

2ème Visite de la pièce R

```
table_interne: {'R': (array([
> special variables
> function variables
> 'R': (array([3.]), 0.0)
> 'C': (array([0., 3.]), 1.0)
len(): 2

table_interne: {'R': (array([
> special variables
> function variables
> 'R': (array([0.]), 3.0)
> 'C': (array([0., 3.]), 1.0)
len(): 2
```

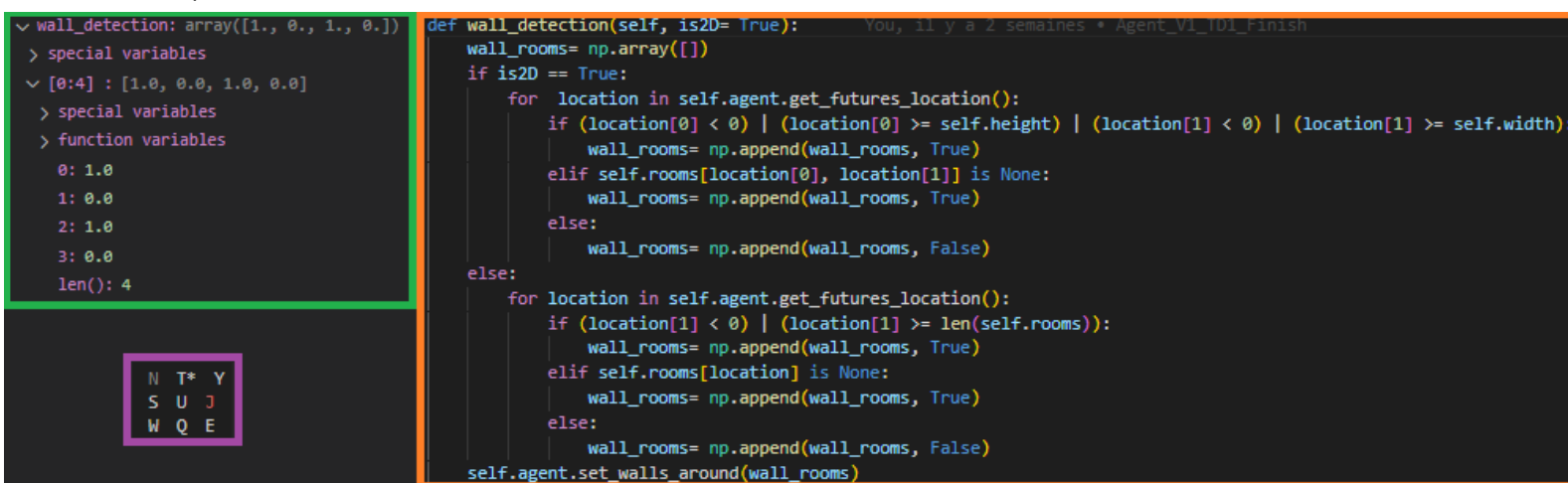
Voici la table des états interne. Pour chaque pièce visitée, l'Agent crée ou

update son dictionnaire des états de l'environnement. La `array` représente les actions pouvant être prises par l'Agent, la valeur après l'`array` représente la dernière actions sélectionnée, qui ne peut donc pas être prise. Lorsque l'Agent va retourner sur une pièce déjà visitée, il va update les données en fonction de l'action qu'il à pris

## Q10=

### REEMPLACER LE CAPTEUR DE POSITION PAR UN CAPTEUR D'OBSTACLE=

Dans le code, ce changement est déjà implémenté car l'agent détecte la présence de mur autour de lui,



The screenshot shows a Jupyter Notebook with three main components:

- Left Panel (Special/Function Variables):**

```

wall_detection: array([1., 0., 1., 0.])
> special variables
[0:4] : [1.0, 0.0, 1.0, 0.0]
> special variables
> function variables
0: 1.0
1: 0.0
2: 1.0
3: 0.0
len(): 4

```
- Bottom Left Panel (Environment Grid):**

```

N T* Y
S U J
W Q E

```
- Right Panel (Code):**

```

def wall_detection(self, is2D= True):
    wall_rooms= np.array([])
    if is2D == True:
        for location in self.agent.get_futures_location():
            if (location[0] < 0) | (location[0] >= self.height) | (location[1] < 0) | (location[1] >= self.width):
                wall_rooms= np.append(wall_rooms, True)
            elif self.rooms[location[0], location[1]] is None:
                wall_rooms= np.append(wall_rooms, True)
            else:
                wall_rooms= np.append(wall_rooms, False)
    else:
        for location in self.agent.get_futures_location():
            if (location[1] < 0) | (location[1] >= len(self.rooms)):
                wall_rooms= np.append(wall_rooms, True)
            elif self.rooms[location] is None:
                wall_rooms= np.append(wall_rooms, True)
            else:
                wall_rooms= np.append(wall_rooms, False)
    self.agent.set_walls_around(wall_rooms)

```

Vous pouvez voir, dans le cadre Vert, la modélisation du capteur de mur / Obstacles. La fonction permettant de créer cette liste de Booléens dans le cadre Orange et enfin la position de l'agent dans l'environnement permettant d'obtenir cette liste de Booléens, dans le cadre Violet.

La liste de Booléen permet de savoir s'il y a (*True* → 1.0) un obstacle / mur ou non (*False* → 0.0). Pour ce faire on récupère les coordonnées des pièces adjacentes à la pièce dans laquelle se trouve l'Agent, puis on regarde si elle existe. Si on a une pièce *None* (Matérialisée par un N noir), c'est un obstacle, si on sort des dimension de l'environnement, on est au bord de l'environnement.

Dans le cas de cette position dans l'environnement, on a la liste suivante

[1.0, 0.0, 1.0, 0.0] → Un obstacle / Mur à Gauche → Rien à Droite  
 → Un obstacle / Mur en Haut → Rien en Bas

## TP#2

### Q1=

ON VA MAINTENANT CRÉER UN NOUVEL AGENT DOTE D'UN MODELE, C'EST-A-DIRE D'UNE DESCRIPTION DE LA FAÇON DONT L'ETAT SUIVANT DÉPEND DE L'ÉTAT COURANT ET DE L'ACTION EFFECTUÉE PAR L'AGENT=

#### - **Choix de la Représentation du Modèle Q1-A=**

J'ai choisi d'adopter une représentation souple, pour stocker les données des états de l'Environnement et actions de l'Agent. Ainsi, j'ai choisi d'utiliser des Dictionnaires. Pour représenter un Etat de l'Environnement et les actions possibles.

```
model: {(0, 0): {0: -1, 1: (...), 2: -1, 3: (...)}, (0, 1): {0: (0, 0), 1: (0, 2), 2: -1, 3: (1, 1)}, (0, 2): {0: -1, 1: -1, 2: -1, 3: (1, 2)}, (1, 0): {0: -1, 1: (1, 1), 2: (0, 0), 3: (2, 0)}, (1, 1): {0: (1, 0), 1: (1, 2), 2: (0, 1), 3: (2, 1)}, (1, 2): {0: (1, 1), 1: -1, 2: (0, 2), 3: (2, 2)}, (2, 0): {0: -1, 1: (2, 1), 2: (1, 0), 3: -1}, (2, 1): {0: (2, 0), 1: (2, 2), 2: (1, 1), 3: -1}, (2, 2): {0: (2, 1), 1: -1, 2: (1, 2), 3: -1}}
len(): 9

Création du Modèle Interne de l'Agent / Table de Transition de l'ensemble des états de l'environnement
def build_model(self):
    for i in range(self.height_env):
        for j in range(self.width_env):
            actual_position= (i, j)
            action= 0
            #Futures positions possibles | 0: Gauche | 1: Droite | 2: Haut | 3: Bas
            next_positions= [(i, j-1), (i, j+1), (i-1, j), (i+1, j)]
            future_locations= dict()
            for location in next_positions:
                if actual_position == (self.init_y_position, self.init_x_position):
                    if self.wall_detection[action] == True:
                        future_locations[action]= -1
                    else:
                        future_locations[action]= location
                elif (location[0] < 0) | (location[0] >= self.height_env) | (location[1] < 0) | (location[1] >= self.width_env):
                    future_locations[action]= -1
                else:
                    future_locations[action]= location
            action +=1
            self.model.update({actual_position: future_locations})
```

Dans le cadre Vert se trouve la modélisation du modèle tandis que dans le cadre orange se trouve la fonction permettant de le créer / contruire

$\{(0,0): \{0: -1, 1: (0,1), 2: -1, 3: (1,0)\}\}$

La première clé correspond à la position de la pièce

Les autres clés correspondent aux position des pièces adjacents

GAUCHE →0, DROITE→1, HAUT→2, BAS→3

La valeur - 1 intervient quand on a pas de pièces adjacente dans la direction donnée(bord de l'Environnement / obstacles).

#### - **Update de L'Environnement Q1-B=**

```
def update_model(self, room): #--> FONCTION ACTUALISER-ETAT
    next_pos_dict= self.model.get(room.position)
    next_position= [(room.position[0], room.position[1]-1), (room.position[0], room.position[1]+1), (room.position[0]-1, room.position[1]), (room.position[0]+1, room.position[1])]
    action= 0
    for wall in self.wall_detection:
        assert next_pos_dict is not None
        if wall == True and next_pos_dict.get(action) != -1:
            next_pos_dict.update({action: -1})
            self.model.update({next_position[action]: -1})
        action += 1
    self.model.update({room.position: next_pos_dict})
```

Grâce à cette fonction l'Agent est capable d'actualiser les états de l'environnement. Dans les faits, elle est très peu utilisée car, on crée une fois la table entière, l'environnement ne se modifie pas(apparition d'obstacles, etc...) donc on a pas vraiment d'updates à faire

- **Choix de la Représentation des Règles Conditions-Actions Q1-C / Codez la Fonction TROUVER\_REGLE Q1-D / Ecrivez les règles qui déterminent le comportement de l'agent=**

J'ai choisi d'adopter une représentation simple / claire. Ainsi, je vais utiliser un Arbre de Décisions.

```
def take_action_from_tree(self):    #--> FONCTION TROUVER-REGLE
    # self.wall_detection= [left, right, top, bottom] correspond au detecteur de murs autour de l'Agent
    action= None
    if self.wall_detection[0] == True:    #--> S'il y a un mur à gauche    You, il y a 25 minu
        if self.wall_detection[1] == True:    #--> Sinon S'il y a un mur à droite
            if self.wall_detection[2] == True:    #--> Sinon S'il y a un mur en haut
                if self.wall_detection[3] == True:    #--> Sinon S'il y a un mur en bas    | Entouré de murs
                    action= -1
                else:
                    action= BAS    #Mur Gauche / Droite / Haut
            else:
                if self.wall_detection[3] == True:    #Mur Gauche / Droite / Bas
                    action= HAUT
                else:
                    action= BAS    #Mur Gauche / Droite
        else:
            if self.wall_detection[2] == True:
                if self.wall_detection[3] == True:    #Mur Gauche / Haut / Bas
                    action= DROITE
                else:
                    action= DROITE    #Mur Gauche / Haut
            else:
                if self.wall_detection[3] == True:    #Mur Gauche / Bas
                    action= HAUT
                else:
                    #Mur uniquement à gauche
                    if random.random() < 0.1:
                        action= DROITE
                    else:
                        action= HAUT

    elif self.wall_detection[1] == True:    #--> Sinon S'il y a un mur à droite
        if self.wall_detection[0] == True:    #--> Sinon S'il y a un mur à gauche
            if self.wall_detection[2] == True:    #--> Sinon S'il y a un mur en haut
                if self.wall_detection[3] == True:    #--> Sinon S'il y a un mur en bas    | Entouré de murs
                    action= -1
                else:
                    action= BAS    #Mur Droite / Gauche / Haut
            else:
                if self.wall_detection[3] == True:    #Mur Droite / Gauche / Bas
                    action= HAUT
                else:
                    action= HAUT    #Mur Droite / Gauche
        else:
            if self.wall_detection[2] == True:
                if self.wall_detection[3] == True:    #Mur Droite / Haut / Bas
                    action= -1
                else:
                    action= BAS    #Mur Droite / Haut
            else:
                if self.wall_detection[3] == True:    #Mur Droite / Bas
                    action= GAUCHE
                else:
                    #Mur uniquement à Droite
                    if random.random() < 0.1:
                        action= GAUCHE
                    else:
                        action= BAS
```

La capture d'écran ne montre que la moitié de la fonction. La stratégie de l'Agent sera de se déplacer sur les bords de l'environnement et de briser cette "routine" en se déplaçant au centre de l'environnement. Lorsque l'Agent sera au centre de l'environnement(ou plutôt, pas sur les bords), il prendra des actions aléatoires. Voici une capture du fonctionnement de cet agent

```

Etat Initial de l'environnement
J* S L
M I Q
G P B
1 Salles à nettoyer
Nombre d'actions Minimum pour tout Nettoyer= 3
> Action n° 1 = ->
J S* L
M I Q
G P B
> Action n° 2 = ->
J S L*
M I Q
G P B
> Action n° 3 = v
J S L
M I Q*
G P B
> Action n° 4 = v
J S L
M I Q
G P B*
> Action n° 5 = <-
J S L
M I Q
G P* B
> Action n° 6 = <-
J S L
M I Q
G* P B
> Action n° 7 = ^
J S L
M* I Q
G P B
> Action n° 8 = ^
J* S L
M I Q
G P B
> Action n° 9 = ->
J S* L
M I Q
G P B
> Action n° 10 = v
J S L
M I* Q
G P B
> Action n° 11 = |.^|
J S L
M I* Q
G P B
Note de Performance= 6.0 / 10
Note moyenne= 6.0 / 10
Note max= 6.0 / 10

```

On voit très bien que l'Agent possède "se déplacer sur les bords de l'environnement" comme routine. S'il n'est pas dans un coins de l'environnement, il possède aussi une probabilité de 10% de briser cette routine et de se déplacer en direction du centre de l'environnement(il sort des bordures). Cet arbre n'est clairement pas optimisé car, la prise d'actions de l'Agent, hors des bordures de l'environnement, est aléatoire. Je n'ai pas réussi à trouver des règles pertinentes dans ce cas pour lever la prise d'actions aléatoire. De plus, selon moi le comportement Condition-Action doit permettre à l'Agent de prendre une action "le plus rapidement possible". C'est-à-dire qu'il ne doit pas y avoir des méthodes de prise de décisions dynamique. Ainsi, cette méthode permet de créer un Agent très limité. Si on apporte plus de complexité à cette méthode, on va préférer utiliser la méthode Réflexes Simples qui, pour ma part consiste en une arbre de décision un peu plus évolué.



## Q2=

### MODIFIEZ MAINTENANT LE COMPORTEMENT DE L'ENVIRONNEMENT POUR LE RENDRE NON DÉTERMINISTE=

- **A chaque fois que l'aspirateur veut aspirer, 20% de chance que l'action échoue Q2-A=**

Pour "activer" cette feature, il faut changer la ligne suivante:

L76 Agent2D.py =

```
if random.random() <= 1: ⇒ if random.random() <= 0.8:
```

- **A chaque fois que l'aspirateur veut aspirer, 20% de chance que l'action échoue Q2-A=**

Pour "activer" cette feature, il faut changer la ligne suivante:

L101 Agent2D.py =

```
if random.random() > 1: ⇒ if random.random() <= 0.1:
```

- **A chaque instants, chaque cases sales à 5% de chance de se salir Q2-C=**

Pour "activer" cette feature, il faut décommenter les lignes suivantes:

L238 - 241 Environnement.py =

```
"""if (nrow != agent_y_pos) & (ncol != agent_x_pos):  
    if random.random() < 0.05:  
        modified_env= True  
        room.mess_room() """
```

↓

↓

```
if (nrow != agent_y_pos) & (ncol != agent_x_pos):  
    if random.random() < 0.05:  
        modified_env= True  
        room.mess_room()
```

- **ATTENTION=**

Ma méthode de notation ne fonctionne pas correctement sur l'environnement à partir du moment que l'Agent à nettoyé sa première pièce. En effet, pour actualiser le nombre minimum d'actions à prendre, j'utilise la fonction suivante(voir code dans le fichier AlgorithmmeAStar.py).

- **def update\_opti\_cpt(self, rooms, opti\_cpt):**

Elle permet de récupérer les pièces sales présentes dans l'environnement ainsi que la position de l'Agent. Elle calcule le nombre minimum d'actions à prendre pour parcourir et nettoyer les pièces sales à partir de sa position et calcul aussi la distance minimum entre la position de l'Agent et sa pièce de référence(position de l'Agent quand une pièce s'est précédemment salit). Lorsque l'agent n'a nettoyé aucunes pièces, cette méthode fonctionne, quand il a nettoyé les "premiers" checkpoints(pièces sales avant l'apparition de nouvelles), la fonction ajoute le nombre d'actions optimisés pour update lerer compteur global. Cependant, lorsqu'on se trouve entre ces 2 cas, je n'ai pas réussi à mettre en place une méthode claire / pertinente pour y répondre

### Q3=

#### MODIFIER LES REGLES QUI DETERMINENT LE COMPORTEMENT DE L'AGENT POUR GERER DE MANIERE RATIONNEL CETTE NOUVELLE VERSION DE L'ENVIRONNEMENT=

Selon moi, il n'existe pas de modifications qui puisse améliorer significativement les performances de l'Agent car les probabilité d'échouer interviennent après la prise de décision. Ainsi,  
A COMPLETER

### Q4=

#### FAITES TOMBER L'HYPOTHÈSE QUE L'AGENT CONNAÎT LA TOPOGRAPHIE DE L'ENVIRONNEMENT=

Pour "activer" cette feature, il faut commenter les lignes suivantes:

L221 - 235 Agent2D.py=

et décommenter les lignes suivantes:

L236 - 246 Agent2D.py=

```
#==> Version Connaissance Innée de l'Environnement
    if self.isModelInit == False:
        self.build_model()
        self.isModelInit= True
    else:
        self.update_model(room)      #--> APPEL FONCTION
ACTUALISER-ETAT
    actions_possibles= self.model[room.position]
    #prev_action= self.table_interne[room_name][1]
    action= random.randint(0,3)
    while actions_possibles[action] == -1:
        action += 1
        if action == 4:
            action= 0
    return action
"""

#==> Version Découverte de l'Environnement
self.update_model(room)      #--> APPEL FONCTION ACTUALISER-ETAT
actions_possibles= self.model[room.position]
#prev_action= self.table_interne[room_name][1]
action= random.randint(0,3)
while actions_possibles[action] == -1:
    action += 1
    if action == 4:
        action= 0
return action
```

```
"""
```

↓

↓

```
"""
    #==> Version Connaissance Innée de l'Environnement
    if self.isModelInit == False:
        self.build_model()
        self.isModelInit= True
    else:
        self.update_model(room)      #--> APPEL FONCTION
ACTUALISER-ETAT
        actions_possibles= self.model[room.position]
        #prev_action= self.table_interne[room_name][1]
        action= random.randint(0,3)
        while actions_possibles[action] == -1:
            action += 1
            if action == 4:
                action= 0
        return action
    """

    #==> Version Découverte de l'Environnement
    self.update_model(room)      #--> APPEL FONCTION ACTUALISER-ETAT
    actions_possibles= self.model[room.position]
    #prev_action= self.table_interne[room_name][1]
    action= random.randint(0,3)
    while actions_possibles[action] == -1:
        action += 1
        if action == 4:
            action= 0
    return action
```

De plus, il faut commenter les lignes suivantes:

L275 - 282 Agent.py=

et décommenter les lignes suivantes:

L283 - 297 Agent.py=

```
next_pos_dict= self.model.get(room.position)
    next_positions= [(room.position[0], room.position[1]-1),
    (room.position[0], room.position[1]+1), (room.position[0]-1,
room.position[1]), (room.position[0]+1, room.position[1])]
    action= 0

    #==> Version Connaissance Innée de l'Environnement
    for wall in self.wall_detection:
        assert next_pos_dict is not None
        if wall == True and next_pos_dict.get(action) != -1:
```

```

        next_pos_dict.update({action: -1})
        self.model.update({next_positions[action]: -1})
        action += 1
    self.model.update({room.position: next_pos_dict})
    """
    #==> Version Découverte de l'Environnement
    actual_position= room.position
    future_locations= dict()
    for location in next_positions:
        if actual_position == (self.init_y_position,
self.init_x_position):
            if self.wall_detection[action] == True:
                future_locations[action]= -1
            else:
                future_locations[action]= location
            elif (location[0] < 0) | (location[0] >= self.height_env) |
(location[1] < 0) | (location[1] >= self.width_env):
                future_locations[action]= -1
            else:
                future_locations[action]= location
            action +=1
    self.model.update({actual_position: future_locations})
    """

```

↓

↓

```

next_pos_dict= self.model.get(room.position)
    next_positions= [(room.position[0], room.position[1]-1),
(room.position[0], room.position[1]+1), (room.position[0]-1,
room.position[1]), (room.position[0]+1, room.position[1])]
    action= 0
    """
    #==> Version Connaissance Innée de l'Environnement
    for wall in self.wall_detection:
        assert next_pos_dict is not None
        if wall == True and next_pos_dict.get(action) != -1:
            next_pos_dict.update({action: -1})
            self.model.update({next_positions[action]: -1})
        action += 1
    self.model.update({room.position: next_pos_dict})
    """
    #==> Version Découverte de l'Environnement
    actual_position= room.position
    future_locations= dict()
    for location in next_positions:

```

```

        if actual_position == (self.init_y_position,
self.init_x_position):
            if self.wall_detection[action] == True:
                future_locations[action]= -1
            else:
                future_locations[action]= location
            elif (location[0] < 0) | (location[0] >= self.height_env) |
(location[1] < 0) | (location[1] >= self.width_env):
                future_locations[action]= -1
            else:
                future_locations[action]= location
            action +=1
        self.model.update({actual_position: future_locations})

```

Faire tomber l'hypothèse de la connaissance innée de la topographie de l'environnement, par l'agent, va induire une transformation d'un Agent avec Modèle vers un Agent avec État Interne. En effet, l'Agent avec Modèle va répertorier, dans son modèle, tous les états de l'environnement, le tout, avant de prendre la moindre actions. Tandis que l'Agent avec Etat Interne va UNIQUEMENT répertorier les états de l'Environnement par lesquels il va passer. En bref, il va créer un "modèle" incomplet. Ainsi, si l'Agent ne connaît pas la topologie de l'environnement à l'avance, il créera un modèle en fonction des états par lesquels il passe, comme un Agent avec Etat Interne.

Ces similarités se retrouves dans la méthode de stockage des données. Pour ce qui est du modèle, il répertorie la nature des position adjacentes(– 1= Mur / (X,Y)= pièce) tandis que la table d'état interne répertorie, pour un état donné, les actions que l'agent peut prendre, ainsi que la dernière action prise. La seule chose qui change est la méthode de représentation des états.