



PRÉSENTÉ PAR:

NOUR EL HADDAD & MATTEO DENIS

APPRENTISSAGE PAR RENFORCEMENT

REINFORCEMENT LEARNING, Q-LEARNING

SOMMAIRE

1. Introduction
2. Description de l'algorithme utilisé
3. Explication du code
4. Résultats et interprétation
5. Conclusion

1. INTRODUCTION

Dans certaines applications, la sortie du système est une séquence d'actions. Dans un tel cas, une action ou un mouvement unique n'est pas si important en soi.

Le but de ce projet est d'implémenter un ou plusieurs algorithmes dans un ou plusieurs environnements.

L'apprentissage par renforcement est utilisé lorsque nous avons une idée claire de ce que nous voulons, mais pas exactement de la façon dont nous pouvons l'atteindre.



1. INTRODUCTION

Exemple : Prenons le cas où vous apprenez un nouveau tour à votre chien :

- Vous ne pouvez pas lui dire ce qu'il doit faire, mais vous pouvez le récompenser ou le punir.
- Ici, nous formons un ordinateur comme si nous formions un chien

1. INTRODUCTION



Dans le cas où l'agent agit sur son environnement, il reçoit une évaluation de son action (renforcement)

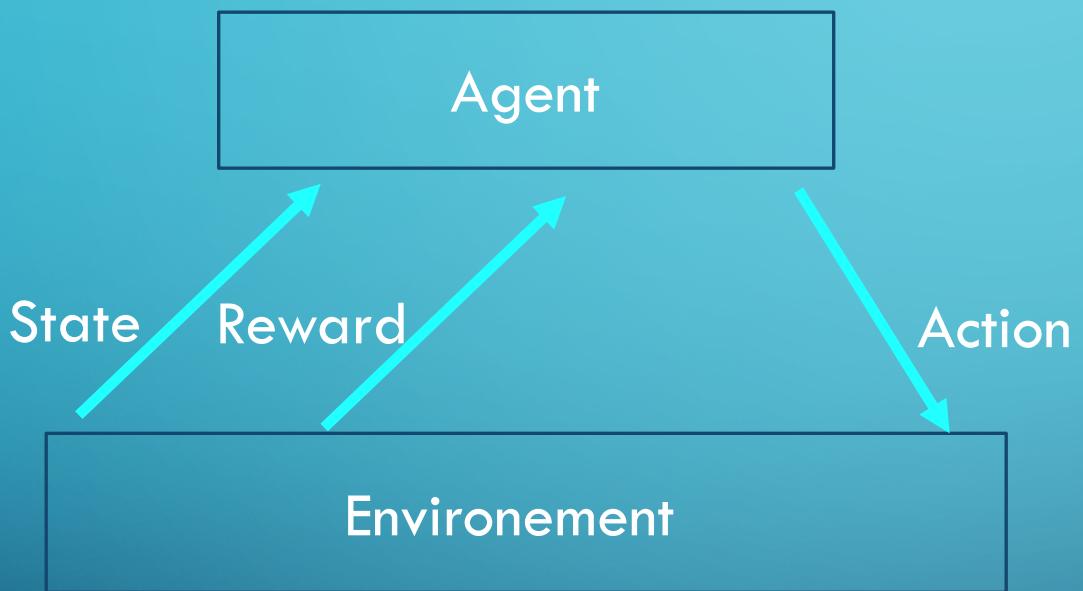


Il faut noter qu'il ne sait pas quelle est l'action correcte pour atteindre son objectif.



But: Apprendre comment se comporter avec succès pour atteindre un objectif tout en interagissant avec un environnement externe (apprendre par l'expérience !)

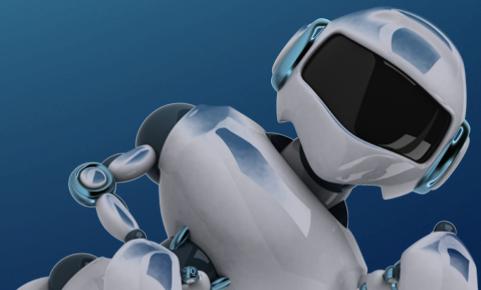
2. DESCRIPTION DE L'ALGORITHME UTILISÉ



- S : ensemble d'états (states)
- A : ensemble d'actions
- $R(s,a)$: la récompense (reward) attendu de l'action a dans l'état s

$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3$

$r_0 \quad r_1 \quad r_2$



2. DESCRIPTION DE L'ALGORITHME UTILISÉ

EXPLICATION DE LA MATRICE Q

- Prédit : récompense future
- Évalue : qualité des état-action
- Sélectionne : actions

$$Q_{\pi}(s, a) = \mathbb{E}_{\pi}[\sum_k \gamma^k r_{t+k+1} | s_t = s, a_t = a]$$

Exemple: Notre agent se trouve dans un **état s** donné, il doit savoir s'il doit aller vers le **haut**, le **bas**, la **gauche** ou la **droite**.

Nous pouvons représenter la fonction **Q-Value** par une matrice de dimension $|S| \times |A|$.

3. EXPLICATION DU CODE

ORGANISATION GÉNÉRALE DU CODE

Main :

Initialisation du maze et des variables

Allocation et initialisation de Q

Q-Learning :

Pour chaque épisode :

Initialisation de l'état s

Tant qu'on n'a pas trouvé la sortie :

Méthode greedy $\rightarrow a$

Applique l'action a $\rightarrow s'$

$$Q [s] [a] = (1 - \alpha) Q [s] [a] + \alpha (récompense + \gamma \max_a Q)$$

$s \leftarrow s'$

Visiter le labyrinthe selon Q

Afficher le chemin

Libérer Q

3. EXPLICATION DU CODE

ORGANISATION GÉNÉRALE DU CODE : MÉTHODE SARSA

Main :

Initialisation du maze et des variables

Allocation et initialisation de Q

Q-Learning :

Pour chaque épisode :

Initialisation de l'état s
Méthode greedy \rightarrow a

Tant qu'on n'a pas trouvé la sortie :

Applique l'action a \rightarrow s'

Méthode greedy \rightarrow a'

$Q[s][a] = (1 - \alpha) Q[s][a] + \alpha (\text{récompense} + \gamma Q[s'][a'])$

s \leftarrow s' ; a \leftarrow a'

Visiter le labyrinthe selon Q

Afficher le chemin

Libérer Q

3. EXPLICATION DU CODE:

FONCTIONS DU CODE

```
int action_max_Q_rand(float** Q, int s){  
  
    int a = rand()%4;  
    int action_max=a;  
    float Q_max = Q[s][a];  
  
    for (int i=0; i<number_actions; i++){  
        if (Q[s][i]>Q_max){  
            Q_max=Q[s][i];  
            action_max=i;  
        }  
    }  
  
    return action_max;  
}
```

action_max_Q_rand:

renvoie l'action maximale, et si toutes les actions sont à égalité, renvoie une action au hasard (utile avec la méthode greedy)

```
int e_greedy(float eps, float** Q, int s) {      //renvoie une action en fct de Q  
  
    action direction;  
  
    if(rand()%101 < eps*100) {                  // au hasard  
        direction=rand()%number_actions;  
    }  
  
    direction=action_max_Q_rand(Q,s);  
  
    return direction;   //sinon on choisit l'action qui maximise Q  
}
```

e_greedy:

applique la méthode greedy avec le taux de hasard epsilon

```

void Q_render(float** Q){

    for (int i=0; i<rows; i++){
        for (int j=0; j<cols; j++){
            if (action_max_Q(Q,i*cols+j)==0){
                printf("H ");
            } else if (action_max_Q(Q,i*cols+j)==1){
                printf("B ");
            } else if (action_max_Q(Q,i*cols+j)==2){
                printf("G ");
            } else if (action_max_Q(Q,i*cols+j)==3){
                printf("D ");
            } else {
                printf("   ");
            }
        }
        printf("\n");
    }
}

```

D	D	D	D	D	D	D	D	D	D	B
H										B
H										B
H										B
H										B
H	G	G								B
										B
B										H
B	H									H
B	H	G	G							H
B										H
D	D	D	D	D	D	D	D	D	H	

Q_render:

Permet d'afficher une représentation de la matrice Q. Pour chaque état est donné l'action maximale. On peut donc superposer l'output au labyrinthe

Lorsque l'état n'a pas d'action maximale, il est symbolisé par un espace

```

float** alloc_Q_1(int size){
    float** Q=calloc(size,sizeof(int*));
    for (int i=0; i<size; i++){
        Q[i]=calloc(number_actions,sizeof(int));
        if(Q[i]==NULL){
            printf("Erreur d'allocation de Q[i]");
            exit (-1);
        }
    }
    if(Q==NULL){
        printf("Erreur d'allocation de Q");
        exit (-1);
    }

    return Q;
}

```

alloc_Q_1:

Alloue l'espace pour la matrice Q,
puis l'initialise à 0

```

void visit(float** Q){

    init_visited();

    int current_row=start_row;
    int current_col=start_col;

    while (visited[current_row][current_col]!=wall || visited[current_row][current_col]!=goal){

        if (action_max_Q(Q,current_row*cols+current_col)==0){
            current_row-=1;
        } else if (action_max_Q(Q,current_row*cols+current_col)==1){
            current_row+=1;
        } else if (action_max_Q(Q,current_row*cols+current_col)==2){
            current_col-=1;
        } else {
            current_col+=1;
        }

        if (visited[current_row][current_col]==crumb){
            break;
        }

        if (visited[current_row][current_col]!=goal && visited[current_row][current_col]!=wall){

            visited[current_row][current_col]=crumb;
        }
    }
}

```

visit:

permet de faire tourner le labyrinthe, selon les résultats obtenus par la matrice Q

4. RÉSULTATS ET INTERPRETATION:

DÉMONSTRATION DU PROGRAMME

Labrinthe donné initialement dans le fichier maze.txt et son environnement dans mazeEnv.c

Matrice Q initialement vide

On rentre en **input** les paramètres suivant que l'on veut

4. RÉSULTATS ET INTERPRETATION:

DÉMARCHE DE DÉVELOPPEMENT ET GESTION DES ERREURS



Matrice initialisée
à 0 partout =>
Blocage

SOLUTIONS

**Grand epsilon
(=0.9)**



alloc_Q_2



action_max_rand



4. RÉSULTATS ET INTERPRETATION:

DÉMARCHE DE DÉVELOPPEMENT ET GESTION DES ERREURS



Problème des murs

SOLUTIONS

Changer environnement :
Mur = terminal



Changer environnement :
Mur = recompense négative



Empecher de choisir certaines actions



5. CONCLUSION

- Ce projet nous a permis de comprendre concrètement la méthode d'apprentissage par renforcement et de faire nos premiers pas dans l'IA.
- Malgré le fait que on l'a beaucoup apprécié, ce projet nous a paru dur. On n'a pu implémenter qu'un seul algorithme dans un seul environnement mais on aurait aimé faire plus.

MERCI POUR VOTRE ATTENTION

