PROJET PROLOG

2024-2025

EQUIPE

LES PETITES LOUTRES

RÉALISÉ PAR

BENAHCENE Sophia
et
GUILBERT Maelle

Sommaire

INTRODUCTION

Le projet présenté consiste en un jeu à deux joueurs, dans lequel chaque joueur choisit un nombre entre 1 et 5 à chaque tour. En fonction des coups choisis, les joueurs se voient attribuer des points selon les règles spécifiques à chaque version du jeu. L'objectif est d'obtenir le score le plus élevé possible après plusieurs tours.

Bien qu'il parait simple en apparence, ce jeu présente plusieurs **défis stratégiques**. L'une des principales difficultés réside dans la nécessité de **prédire les coups** de l'adversaire. En effet, les points obtenus par le joueur A dépendent du coup qu'il joue mais **également du coup joué par son adversaire**. Le score de chacun est déterminé par le couple (CoupJoueurA, CoupJoueurB). Ainsi, chaque joueur doit non seulement **optimiser ses propres choix** pour maximiser son score, mais aussi **anticiper les actions** de l'autre. Par ailleurs, il est crucial de **ne pas devenir prévisible**, car un adversaire capable de repérer une répétition dans les choix d'un joueur pourra facilement en tirer parti.

Un autre défi majeur réside dans l'**adaptabilité du programme**. En fonction des actions de l'adversaire, le joueur doit **ajuster sa stratégie** au fil du jeu pour rester compétitif. Cette flexibilité est essentielle, car une stratégie rigide ne permettrait pas de réagir efficacement face à des changements dans la façon de jouer de l'autre joueur.
Dans ce contexte, l'objectif est de développer un programme capable de s'adapter de manière optimale à l'**environnement dynamique du jeu**, en tenant compte de ces différentes contraintes et en maximisant les chances de victoire.

Mais alors, quelle stratégie choisir ?

Dans ce rapport, nous vous présenterons nos pistes de réflexion concernant les stratégies possibles. Nous aborderons notamment deux approches principales auxquelles nous avons pensé : l'une centrée sur **l'équilibre de Nash**, visant à rendre les choix de l'adversaire indifférents, et l'autre exploitant les **méthodes de Q-learning**, permettant au programme d'apprendre et de s'adapter dynamiquement au comportement adverse.

1. Une première stratégie : L'EQUILIBRE DE NASH

Étant toutes deux en parcours économie, nous avons d'abord mobilisé nos connaissances en **théorie des jeux.** Nous avons considéré le jeu comme un **jeu simultané**, où les joueurs choisissent un coup entre 1 et 5 sans connaître celui de l'autre, et où les gains sont les points obtenus à chaque manche.

Nous avons construit la **matrice des gains** en associant à chaque couple de coups un couple de scores, puis cherché à déterminer un **équilibre de Nash**. Il s'agit d'une situation où aucun joueur n'a intérêt à changer de stratégie seul, car il n'améliorerait pas son gain. Nous avons donc exploré l'équilibre en **stratégie mixte**, qui répartit les choix selon des **probabilités optimales.**

Matrice des gains associée à la version 1 du jeu

JOUEUR B

JOUEUR A	1	2	3	4	5
1	(1,1)	(3,0)	(1,3)	(1,4)	(1,5)
2	(0,3)	(2,2)	(5,0)	(2,4)	(2,5)
3	(3,1)	(0,5)	(3,3)	(7,0)	(3,5)
4	(4,1)	(4,2)	(0,7)	(4,4)	(9,0)
5	(5,1)	(5,2)	(5,3)	(0,9)	(5,5)

Dans un premier temps, nous avons tenté de déterminer l'équilibre de Nash en stratégie mixte en suivant la méthode vue en cours de microéconomie. Nous avons posé les **équations d'indifférence** permettant de trouver les probabilités optimales. Cependant, ces calculs se sont révélés longs et sources potentielles d'erreurs.
Afin de gagner du temps et de confirmer nos résultats, nous avons ensuite fourni la matrice de gain à ChatGPT. Celui-ci nous a permis d'obtenir les probabilités optimales en stratégie mixte : **p=0** pour le coup 1, **p=0** pour le coup 2, **p=4/9** pour le coup 3, **p=2/9** pour le coup 4 et **p= ⅓** pour le coup 5

Implémentation en prolog

Maintenant que nous avions déterminé les probabilités associées à l'équilibre de Nash, il nous fallait programmer un agent capable de jouer chaque coup en respectant cette stratégie. Pour cela, nous avons implémenté plusieurs prédicats en Prolog, afin de générer un coup aléatoire selon les probabilités définies. Nous avions ainsi pour ambition de jouer l'équilibre de Nash lors du premier test du tournoi.

Les prédicats

```
% Prédicat pour choisir un élément aléatoire pondéré
choisir_aleatoire_pondere(lesPetitesLoutres, Probabilites, Elements, ElementChoisi) :-
    random(1, 101, NombreAleatoire),  % Nombre aléatoire entre 1 et 100
    choisir_par_probabilite(lesPetitesLoutres,Probabilites, Elements, NombreAleatoire, ElementChoisi).
```

Le prédicat choisir_aleatoire_pondere genere un nombre aléatoire entre 1 et 100. A partir de ce nombre, il choisi un élément en appelant le prédicat choisir_par_probabilité. Il prend plusieurs paramètres :

- Le nom de l'équipe
- La liste des probabilités de chaque coup
- La liste des coup possibles
- L'élément choisi (celui renvoyé)

```
% Règle pour choisir un élément en fonction des probabilités
choisir_par_probabilite(lesPetitesLoutres,[Prob|_], [Elem|_], Random, Elem) :-
    Random =< Prob, !.
```

```
choisir_par_probabilite(lesPetitesLoutres,[Prob|RestProbs], [_|RestElems], Random, ElementChoisi) :-
    Random1 is Random - Prob,
    choisir_par_probabilite(lesPetitesLoutres, RestProbs, RestElems, Random1, ElementChoisi).
```

Le predicat choisir_par_probabilite prend en paramètre la liste des probabilités, la liste des coups possibles, un nombre aléatoire et renvoie un élément i le nombre aléatoire est inférieur ou égal à la première probabilité de la liste, alors on renvoie le premier élément de la liste des coups possibles. Sinon, on soustrait cette première probabilité au nombre aléatoire et on recommence le processus avec les éléments restants des deux listes.
Ce mécanisme permet d'associer chaque intervalle de probabilité à un coup spécifique et de simuler un tirage pondéré.

```
joue(lesPetitesLoutres,_, Coup) :-choisir_aleatoire_pondere(lesPetitesLoutres, [44, 22, 33], [3, 4, 5], Coup).
```

Le prédicat joue appelle alors le prédicat choisir_aleatoire_pondere pour générer un coup selon les probabilités associées à chaque coup. La liste des probabilités utilisée est **[44, 22, 33]**, car 4/9 correspond approximativement à 44 %, 2/9 à 22 %, et 1/3 à 33 %. Ensuite, on lui fournit la liste des coups [3, 4, 5], puisque, selon l'équilibre de Nash, les coups 1 et 2 ne sont pas joués. On obtient ainsi un coup tiré au hasard en respectant cette distribution.

Critique de l'implémentation

Ce code est celui que nous avons soumis pour le test du tournoi. Cependant, nous ne nous étions pas vraiment investies ni suffisamment concentrées au moment de sa réalisation, ce qui nous a fait passer à côté d'un problème important : **la somme des probabilités [44, 22, 33] ne fait que 99 au lieu de 100.** Cela peut provoquer une erreur lorsque le nombre aléatoire généré vaut exactement 100, car aucun coup ne sera alors sélectionné. Ce décalage vient du fait que nous avons utilisé des **approximations de fractions** (4/9, 2/9, 1/3) sans ajustement précis.

Nous avons corrigé le code en générant cette fois-ci un nombre **flottant entre 0 et 1,** ce qui nous permet de conserver les fractions exactes pour les probabilités. Le reste du processus est resté similaire, mais cette modification a permis d'éviter les erreurs générées par l'approximation des fractions. Le programme fonctionne désormais de manière plus précise.

```
choisir_aleatoire_pondere(lesPetitesLoutres, Probabilites, Elements, ElementChoisi) :-
    random(R),  % R est un flottant entre 0.0 et 1.0
    choisir_par_probabilite(lesPetitesLoutres, Probabilites, Elements, R, ElementChoisi).
choisir_par_probabilite(lesPetitesLoutres, [P|_], [E|_], R, E) :-
    R =< P, !.

choisir_par_probabilite(lesPetitesLoutres, [P|Ps], [_|Es], R, E) :-
    R > P,
    R1 is R - P,
    choisir_par_probabilite(lesPetitesLoutres, Ps, Es, R1, E).

joue(lesPetitesLoutres, _, Coup) :-
    % Probabilités exactes : 4/9 pour 3, 2/9 pour 4, 1/3 pour 5
    choisir_aleatoire_pondere(lesPetitesLoutres, [4/9, 2/9, 1/3], [3, 4, 5], Coup).
```

Autres prédicats defensifs

En plus du prédicat joue qui utilise l'équilibre de Nash, nous avons créé deux autres prédicats joue destinés à **reconnaître des patterns** et à contrer la stratégie de l'adversaire. Pour ce faire, nous avons utilisé la liste de paramètres pour **identifier des motifs récurrents** dans les choix de l'adversaire. Ces prédicats avaient pour objectif d'améliorer l'**adaptabilité** du programme face aux comportements récurrents ou **prévisibles** du joueur adverse.

```
joue(lesPetitesLoutres, [[C,B],[B,A],[A,_]|_],Coup):- C\=1, Coup is C-1,!.
joue(lesPetitesLoutres, [[_,B],[B,A],[A,_]|_],5).
```

Ce prédicat permet de reconnaître les joueurs qui **imitent notre dernier coup**. Par exemple, si je joue le coup A, l'adversaire jouera également A au tour suivant, et si je joue B, il jouera B au tour suivant. Cela me permet **d'anticiper sa prochaine action**. Si je joue C, je sais qu'il choisira aussi C au prochain tour, ce qui me **pousse à jouer C-1** pour maximiser mes points. Cependant, si C = 1, je joue alors 5 pour optimiser mes gains. L'adversaire, adoptant une stratégie de type "tit for tat", **gagnera ainsi soit 0, soit 1** point en réponse à ce prédicat, selon le cas.

```
joue(lesPetitesLoutres,[[_,A],[_,A]|_],C):-C\=1,C is A-1.
```

Ce prédicat est principalement utilisé pour se défendre dans la deuxième version du jeu, qui intègre des **multiplicateurs**. Lorsqu'un adversaire joue **deux fois le même coup** (par exemple, A suivi de A), ce prédicat permet de **réagir en jouant A-1** au tour suivant. Cette stratégie vise à **limiter les gains de l'adversaire** en l'empêchant de bénéficier pleinement de ses choix répétitifs. En jouant A-1, on cherche à minimiser l'impact de son coup et à éviter qu'il ne marque trop de points.

Ces prédicats sont utiles pour contrer des stratégies basiques et prévisibles, comme celles où l'adversaire répète nos coups ou joue toujours la même chose. Ils permettent de prendre l'avantage en l'anticipant. Mais cela peut aussi **se retourner contre nous** : un adversaire malin peut deviner qu'on cherche à le contrer, et utiliser cette anticipation contre nous en nous piégeant. En effet, l'adversaire **peut anticiper qu'on va l'anticiper** et dans ce cas c'est **nous qui devenons prévisible**.

Nous n'étions pas totalement satisfaites de notre code : il nous paraissait assez simple, peu original, et vulnérable à certains pièges tendus par des adversaires plus stratèges. En discutant avec d'autres groupes, nous avons aussi constaté que beaucoup avaient fait le choix de jouer l'équilibre de Nash, ce qui rendait notre approche peu innovante. Il nous a donc semblé nécessaire de réfléchir à une stratégie plus intéressante, plus adaptée au contexte du tournoi, et surtout plus élaborée.

2 Une seconde stratégie : LE Q-LEARNING

Pour nous, le meilleur programme était celui capable de **s'adapter** à son adversaire au fil du jeu. Un programme efficace ne devait pas simplement appliquer une **stratégie figée**, mais plutôt **apprendre** des comportements adverses pour **ajuster ses choix au fur et à mesure**. C'est dans cette optique que nous nous sommes intéressées aux méthodes d'apprentissage, notamment celles utilisées en **intelligence artificielle**. C'est ainsi que nous avons découvert le **Q-learning**, une méthode d'apprentissage par **renforcement** qui permet à un agent d'apprendre progressivement quelle action adopter dans chaque situation afin de maximiser ses gains.

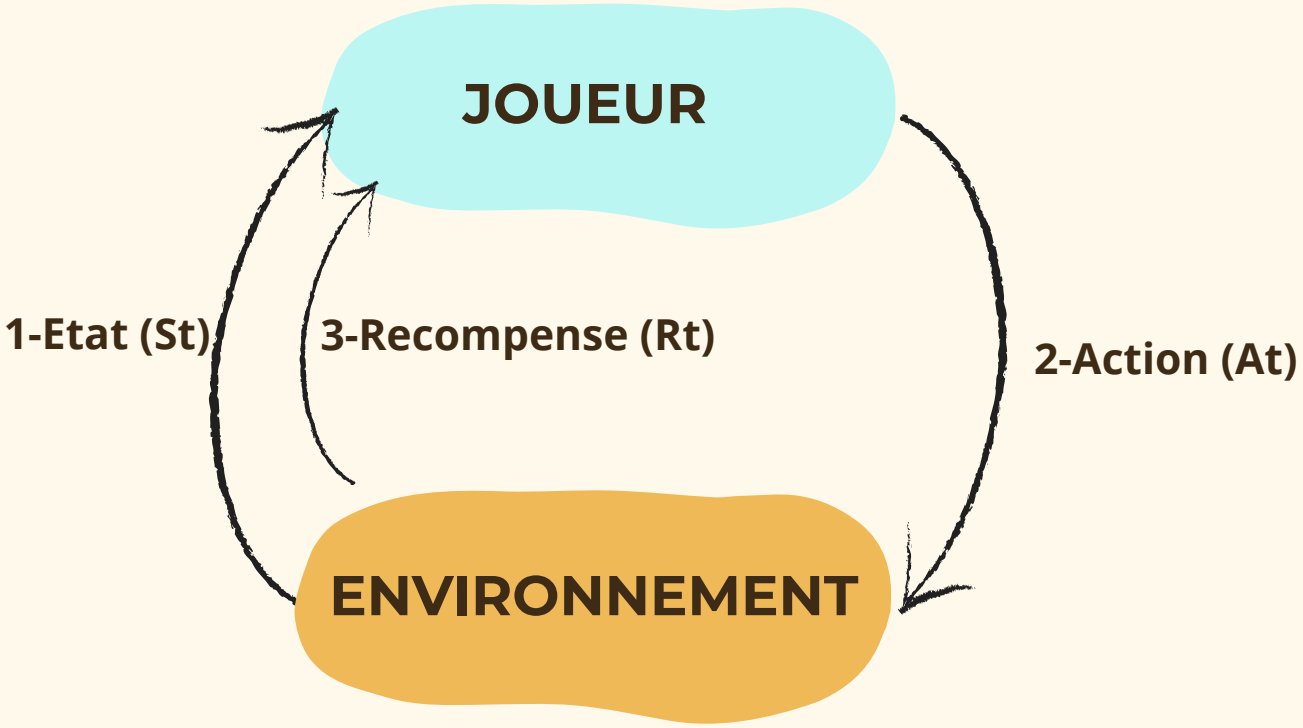
Principe du Q-Learning

Le Q-learning, c'est une méthode qui permet à un programme d'apprendre tout seul à bien jouer, juste en **essayant et en observant ce qu'il se passe**. L'idée, c'est qu'à chaque tour, le programme se retrouve dans une certaine situation (qu'on appelle un **état**), et il doit choisir une **action** (ici, quel coup jouer).

Une fois qu'il a fait son choix, il voit ce que ça lui **rapporte** (des points, ou pas), et il garde ça en mémoire. Au fur et à mesure, il apprend quelles actions sont les plus intéressantes dans chaque situation pour **gagner un maximum de points à long terme.**

Il fait ça en **remplissant une sorte de tableau** (qu'on appelle une table Q), où il note, pour chaque situation, quelle action semble être la meilleure. Et ce tableau, il le **met à jour en permanence**, en fonction des résultats de ses choix.

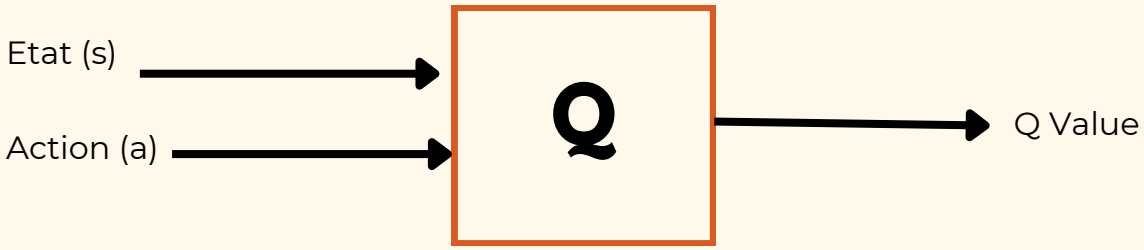
On peut synthétiser le fonctionnement du Q Learning par le schéma ci-dessous



On a rapidement constaté que le cadre du jeu se prêtait très bien à une approche par Q-learning. D'abord, les actions possibles sont claires : ce sont simplement les coups qu'on peut jouer (1 à 5). Ensuite, on a choisi de représenter l'état comme **les trois derniers coups de l'adversaire**. L'idée, c'est que s'il suit un certain **pattern**, notre programme pourra l'identifier et apprendre à y répondre efficacement. Enfin, les récompenses sont naturellement définies : ce sont les points qu'on gagne à chaque manche. Bref, tous les éléments nécessaires pour mettre en place un apprentissage par renforcement étaient déjà présents dans le jeu.

La fonction Q

La fonction Q est une fonction qui permet de **déterminer le niveau d'efficacité** dune action prise dans un Etat particulier. Elle prend en entrée l'état(s) et l'action (a) et renvoie une valeur appelée Q value.



Ainsi, la Q value représente **à quel point il a été bénéfique** de jouer l'action a dans l'état s. Un fois calculée, la Q value est stockée dans une table : la **Q table**. Dans cette table, chaque Q value est associée à un couple état, action. Ce tableau peut en fait être considérer comme **la mémoire** de notre programme avec tous ce qu'il a appris dedans. C'est les valeurs de ce tableau que le programme va utiliser pour choisir le coup optimal. Ainsi, tout le principe du Q Learning consiste a **actualiser** et **ajuster** la Q table, afin d'**apprendre de plus en plus**, et de maximiser sa récompense. Voici un aperçu de la Q table dans le cadre du jeu :

Q(s,a)	s1	s2	s3
Jouer 1	Q(s1,1)	Q(s2,1)	Q(s3,1)
Jouer 2	Q(s1,2)	Q(s2,2)	Q(s3,2)
Jouer 3	Q(s1,3)	Q(s2,3)	Q(s3,3)
Jouer 4	Q(s1,4)	Q(s2,4)	Q(s3,4)
Jouer 5	Q(s1,5)	Q(s2,5)	Q(s3,5)

s1,s2,s3 correspondent à une **liste contenant les 3 derniers coups de l'adversaire**. Par exemple, si il a jouer 2 puis 3 puis 4 alors s1=[2,3,4].

Comment calculer la Q value ?

La formule pour calculer la Q value est la suivante :

$$Q(s,a) <- Q(s,a) + \alpha[R(s,a) + \gamma \max_a Q'(s',a') - Q(s,a)]$$

Le terme alpha correspond au **taux d'apprentissage**. Il est compris entre 0 et 1. Plus il est élevé, plus l'algorithme apprend rapidement, mais il risque aussi d'être **moins stable**. Un alpha plus bas rend l'apprentissage plus lent, mais **plus régulier**. Le R représente la **récompense immédiate** obtenue après avoir effectué une action a dans un état s.
Le gamma est le **taux d'actualisation.** Il sert à équilibrer l'importance entre les récompenses immédiates et celles du futur. Plus gamma est élevé, plus l'agent prend en compte les récompenses à long terme. Enfin, s' désigne l'état **futur**, c'est-à-dire l'état dans lequel l'agent se retrouve après avoir effectué son action.

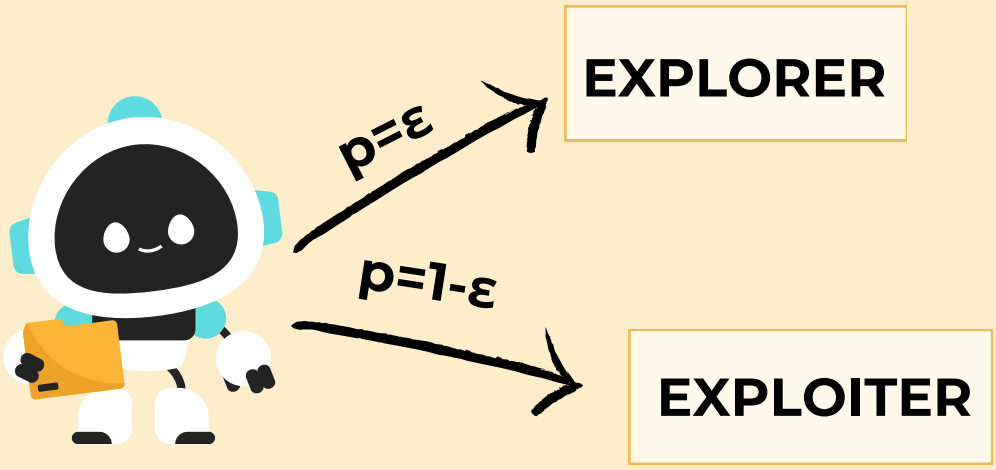
La stratégie d'exploration

Au début du jeu, la Q-table est **vide**, et les premières valeurs enregistrées ne **sont pas encore fiables ou bien ajustées**. Il est donc essentiel d'alterner entre deux types de comportements : l'**exploration** et l'**exploitation**.

L'exploration, c'est le fait de **tester de nouvelles actions**, même si elles ne semblent pas les meilleures à première vue. Cela permet de découvrir de nouvelles stratégies ou de meilleures options qu'on n'aurait pas essayées autrement. Cela permet de remplir la Q table.

L'exploitation, à l'inverse, consiste **à rejouer les actions qui ont déjà donné de bons résultats** dans le passé, en se basant sur les Q-values apprises.

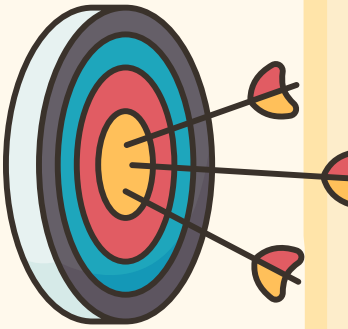
En combinant ces deux approches, le programme peut affiner ses Q-values de manière efficace. Il apprend progressivement quelles actions rapportent vraiment, tout en gardant une certaine curiosité pour ne pas passer à côté d'une stratégie plus performante. Ce principe est appelée **ε-greedy**



Plus le programme joue, plus il accumule d'informations sur les actions les plus efficaces dans différentes situations. Il a donc **de moins en moins besoin d'explorer** au hasard, et **de plus en plus intérêt à exploiter** ce qu'il a appris pour maximiser ses gains.
C'est pour cette raison que ε (epsilon) **diminue progressivement au fil du jeu** : au début, l'exploration est utile pour découvrir différentes situations, mais plus l'apprentissage avance, plus il devient pertinent de se concentrer sur les actions qui ont donné les meilleurs résultats jusque-là.

Implémentation en prolog

Une fois le principe du Q Learning bien compris, le **principal défi** a été de l'implémenter dans Prolog. En cours, on avait principalement travaillé sur des faits et des **règles statiques**, sans vraiment aborder la **notion d'apprentissage** ou de **mise à jour en temps réel.** Or, ici, il fallait justement **introduire du dynamisme**, faire évoluer les Q-values au fil des parties, en fonction des choix et des résultats. C'était donc un défi bien plus **complexe** que l'Equilibre de Nash mais en faisant quelques recherches nous avons trouver des solutions pour réaliser le Q Learning en Prolog.



```
:- dynamic q_value/3, visite/2, epsilon/1.
```

Lors de nos recherches sur Prolog, nous avons découvert les **prédicats dynamiques**, qui permettent de modifier des faits pendant l'exécution. Cette fonctionnalité était parfaite pour implémenter le Q-learning. Nous avons donc déclaré trois prédicats dynamiques essentiels :

- **q_value/3** : pour **stocker les Q-values** associées à chaque couple (état, action) et les actualiser.
- **visite/2 :** pour **mémoriser les couples (état, action)** joués précédemment.
- **epsilon/1 :** pour suivre la valeur de epsilon, utilisée dans la **stratégie d'exploration ε-greedy.**

Ces prédicats nous ont permis de rendre notre programme dynamique et d'ajuster l'apprentissage au fil du jeu.

Dans cette partie du code, nous définissons les **hyperparamètres** du Q-learning. Ce sont des constantes qui influencent le comportement de l'algorithme d'apprentissage.

```
alpha(0.1).
gamma(0.9).
epsilon(1.0).
epsilon_min(0.05).
epsilon_reduc(0.995).
```

- **alpha(0.1)** est le **taux d'apprentissage.** Il détermine à quel point les nouvelles informations viennent corriger les anciennes. Une valeur de 0.1 signifie qu'on met à jour lentement les Q-values.
- **gamma(0.9)** est le **facteur de réduction** (ou de "discount"). Il permet de pondérer l'importance des récompenses futures. Ici, on donne une grande importance aux coups suivants.

- **epsilon(1.0)** représente la **probabilité initiale d'exploration** : au début, le programme choisira souvent des actions aléatoires pour explorer l'environnement.
- **epsilon_min(0.05)** est la **valeur minimale** que peut prendre epsilon, ce qui garantit qu'on **garde toujours un peu d'exploration**.
- **epsilon_reduc(**0.995) est le facteur par lequel epsilon sera multiplié à chaque tour, ce qui permet de **réduire progressivement l'exploration** au fil du temps.

Pourquoi le choix de ces valeurs ?

Nous avons défini plusieurs hyperparamètres essentiels pour le bon fonctionnement du Q-learning. alpha (fixé à 0.1) contrôle la **vitesse d'apprentissage** : plus il est faible, plus l'agent apprend lentement, ce qui permet de **lisser les erreurs** et de **stabiliser les valeurs**. gamma, fixé à 0.9, permet de prendre en compte non seulement le gain immédiat, mais aussi les gains futurs, ce qui est important dans un jeu ou l'anticipation peut rapporter beaucoup de points.
Pour gérer l'équilibre entre exploration et exploitation, nous utilisons la stratégie ε-greedy présentée précédemment. On commence avec epsilon = 1.0 pour favoriser l'exploration totale au début, ce qui permet à l'agent **d'acquérir de nouvelles connaissances.** Ensuite, epsilon décroît à chaque tour avec un facteur de epsilon_reduc = 0.995 : plus l'agent joue, plus il privilégie les actions les plus prometteuses selon ses Q-values. Enfin, on impose un minimum (epsilon_min = 0.05) afin de conserver **une petite part d'exploration** tout au long de la partie, et **éviter de rester bloqué sur une stratégie imparfaite**

```
actions([1,2,3,4,5]).
val_defaut(0).
```

Ce bout de code définit les coups possibles avec **actions/1**, ce qui est essentiel pour parcourir les actions lors du choix ou de la mise à jour. La **val_defaut(0)** est utilisée au **début du jeu**, quand on n'a **pas encore assez d'historique pour former un état complet**. Cela évite des erreurs et initialise proprement l'apprentissage.

```
get_q(Etat,Action,Q):- q_value(Etat,Action,Q),!.
get_q(Etat,Action,0.0):- assertz(q_value(Etat,Action,0.0)).
```

Le prédicat **get_q** sert à **récupérer la Q-value associée à un couple état-action**. Si la Q-value existe déjà pour ce couple, elle est simplement retournée. Sinon, on l'initialise à 0.0, car au départ, l'agent ne connaît pas encore les valeurs des actions et considère qu'elles sont toutes égales. Cela reflète l'**absence de préférence et de connaissance** pour cette action.

```
choix_action(Etat, Action) :-
    epsilon(E),
    epsilon_min(Em),
    epsilon_reduc(Ed),
    actions(AList),
    random(R),
    choisir_action(E, R, AList, Etat, Action),
    maj_epsilon(E, Em, Ed).
```

Le prédicat **choix_action/2** est le **point d'entrée de la sélection d'un coup** par le programme. Il utilise la stratégie dite ε-greedy, qui équilibre l'exploration et l'exploitation.
Dans un premier temps, **on récupère les paramètres nécessaires** :

- epsilon
- sa valeur minimale (epsilon_min),
- son taux de réduction à chaque tour (epsilon_reduc),
- ainsi que la liste des actions possibles (actions).

Ensuite, on génère un **nombre aléatoire R entre 0 et 1.** Ce nombre R va nous permettre de déterminer si on explore ou si on exploite. Ensuite on appelle le prédicat **choisir_action/5**, qui détermine le coup à jouer en fonction de R et des Q-values déjà apprises.
Enfin, maj_epsilon **permet de réduire et d'acutaliser la valeur d'epsilon.**

```
choisir_action(E, R, AList,_, Action) :-
    R < E,
    random_member(Action, AList), !.
```

Ce prédicat **choisir_action/5** correspond au cas où **l'agent choisit d'explorer.**
La condition R < E signifie que l'on est dans la phase d'exploration, c'est-à-dire que le nombre aléatoire tiré R (entre 0 et 1) est inférieur à la probabilité d'exploration epsilon.
Dans ce cas, l'agent ignore temporairement ce qu'il a appris et choisit une **action au hasard** dans la liste des actions possibles AList, grâce à **random_member/2**. On met ensuite un cut car le coup sera choisi à l'issu de ce prédicat (si on est dans la phase d'exploration alors on n'est pas dans la phase d'exploitation, d'où le cut).

```
choisir_action(_, _, AList, Etat, Action) :-
    findall(Q-A, (member(A, AList), get_q(Etat, A, Q)), Paires),
    keysort(Paires, L),
    reverse(L, [_-Action | _]).
```

Ce prédicat est utilisé lorsque l'agent n'est pas en phase d'exploration, donc lorsqu'il choisit une action **en exploitant ce qu'il a appris jusque-là** . Avec le prédicat **findall/3**, on parcourt toutes les actions possibles (AList) et on récupère, pour chacune d'elles, la Q-value associée à l'état courant (Etat). On construit alors une **liste Paires constituée de couples Q-A, où Q est la valeur et A l'action**.
Avec **keysort/2** on trie cette liste de couples par ordre croissant selon la Q-value. Comme on veut l'action avec **la plus grande Q-value**, on inverse la liste triée pour que la meilleure action soit en tête. On extrait ensuite cette action, qui sera celle choisie par le programme.

```
maj_epsilon(E, Em, Ed) :-
    E > Em,
    E1 is max(Em, E * Ed),
    retractall(epsilon(_)),
    assertz(epsilon(E1)), !.

maj_epsilon(_, _, _).
```

Le prédicat **maj_epsilon/3** sert à **mettre à jour** la valeur d'epsilon dans le cadre de la stratégie ε-greedy. Tout d'abord, on vérifie si la valeur actuelle d'epsilon (E) est **supérieure à la valeur minimale** (Em). Si c'est le cas, on calcule une nouvelle valeur pour epsilon en la **réduisant par un facteur de Ed** (le facteur de réduction), mais en garantissant que **cette valeur ne soit pas inférieure à Em** grâce à la fonction **max/2**. Ensuite, on met à jour dynamiquement la valeur d'epsilon dans le programme en **supprimant l'ancienne valeur** avec **retractall/1** et en **ajoutant la nouvelle** avec **assertz/1**. Enfin, un cut (!) est utilisé pour garantir que ce prédicat s'arrête ici et n'essaie pas d'autres clauses.

Si la valeur d'epsilon est **déjà inférieure ou égale à la valeur minimale**, la deuxième clause s'applique, qui ne fait rien et permet simplement de **ne pas modifier l'epsilon**, garantissant ainsi que l'agent ne cesse jamais complètement d'explorer.

```
ajuster_q(S,A,R,S2):-
  get_q(S,A,Q0), actions(AList),
  findall(Qn,(member(A2,AList),get_q(S2,A2,Qn)),Qs), max_list(Qs,MaxQ),
  alpha(Al), gamma(G), TD is R + G*MaxQ - Q0, Qnew is Q0 + Al*TD,
  retractall(q_value(S,A,_)), assertz(q_value(S,A,Qnew)).
```

Le prédicat **ajuster_q/4 met à jour la Q-value** associée à une action A dans un état S après qu'une récompense R ait été reçue et que l'agent soit passé à l'état suivant S2.
Il commence par **récupérer la Q-value actuelle** pour l'état S et l'action A. Ensuite, il explore toutes les actions possibles dans l'état suivant S2 et trouve la **Q-value maximale** parmi elles, ce qui représente la meilleure option à long terme.
En utilisant les paramètres alpha (taux d'apprentissage) et gamma (facteur de réduction), la **différence temporelle** (TD) est calculée, mesurant l'écart entre la Q-value actuelle et la récompense future attendue. Cette différence permet d'ajuster la Q-value de l'action (S, A) pour **mieux refléter l'estimation de la récompense globale.**
Enfin, la Q-value mise à jour est enregistrée, remplaçant l'ancienne valeur, afin d'améliorer les décisions futures de l'agent. Ce processus permet à l'agent d'apprendre et d'ajuster ses choix en fonction de ses expériences.

```
etat_depuis_historique(Historique, Etat) :-
  val_defaut(D),
  findall(Coup, (nth1(I, Historique, [_, Coup]), I =< 3), CoupsRecents),
  length(CoupsRecents, L),
  ( L < 3 -> Manquant is 3 - L, length(Z, Manquant), maplist(=(D), Z), append(CoupsRecents, Z, CoupsFinal)
  ; CoupsFinal = CoupsRecents ),
  Etat = CoupsFinal.
```

Le prédicat **etat_depuis_historique/2** a pour objectif de transformer l'historique des coups en un état utilisable pour le Q-learning. Cet état est représenté par **une liste de trois entiers, correspondant aux trois derniers coups de l'adversaire.** L'implémentation suit les étapes suivant. D'abord on récupère une la valeur par défaut (0), qui sera utilisée si l'historique n'est pas assez long (au début du jeu notamment).Ensuite, avec **findall/3**, on parcourt les **trois premières paires de l'historique** (chaque paire est de la forme [_, CoupAdverse]), et on extrait les coups adverses. (nth1 nous permet de parcourir les 3 premières grâce à l'indice I). Cela donne une liste DerniersCoups contenant au maximum trois éléments.Puis, on vérifie combien de coups ont été extraits. Si on en a moins de trois (L < 3), on calcule combien de valeurs il manque (Manquant is 3 - L), puis on crée une liste de cette longueur contenant uniquement la valeur par défaut (grâce à maplist(=(D), Liste)), et on la concatène avec les coups récupérés pour obtenir une liste de taille trois. Sinon, on garde simplement les trois coups extraits. On obtient ainsi l'Etat qu'on retourne dans le second paramètre.

```
joue(lesPetitesLoutres,Hist,Action):-
   etat_depuis_historique(Hist,S),
   ( retract(visite(PS,PA))
  ->   PS=[Prev|_],
       score(PA,Prev,Hist,Rew,_),
       ajuster_q(PS,PA,Rew,S)
  ;    true ),
     choix_action(S,Action),
     assertz(visite(S,Action)).
```

Ce prédicat est le cœur du fonctionnement de notre programme utilisant le Q-learning. À chaque tour de jeu, il commence par **transformer l'historique des coups (Hist) en un état** grâce au prédicat **etat_depuis_historique.** Ensuite, le programme regarde s'il a enregistré une visite précédente (c'est le cas **sauf au premier coup**), c'est-à-dire un couple (état précédent, action jouée) stocké à la fin du tour précédent. Si c'est le cas, cela signifie qu'il a suffisamment d'information pour **mettre à jour la Q-value** associée à cette décision. Pour cela, il extrait le **dernier coup adverse** (Prev) depuis l'état précédent, puis appelle le prédicat **score** pour obtenir le **gain** qu'il a reçu à la suite de son action. Ce gain, appelé "reward", est ensuite utilisé pour ajuster la Q-value avec **ajuster_q.**

Enfin, une fois cette mise à jour effectuée (ou ignorée si aucune visite précédente n'était disponible), l'agent **choisit sa prochaine action** grâce au prédicat **choix_action**, qui applique la stratégie ε-greedy : il explore ou exploite selon la valeur actuelle de epsilon. L'action choisie est renvoyée comme réponse, et le couple (état actuel, action jouée) est **enregistré via assertz pour pouvoir être utilisé lors du tour suivant.**

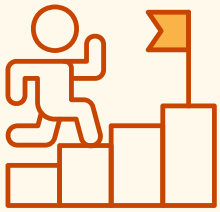
Les avantages du Q Learning

Ce code présente plusieurs avantages. D'abord, il permet à notre programme d'**apprendre progressivement** en s'adaptant au comportement de l'adversaire. Grâce au Q-learning, il ne suit pas une stratégie fixe mais **ajuste ses décisions** selon les résultats passés, en identifiant les actions les plus efficaces dans chaque situation. Par exemple, les réflexes défensifs que nous avions codés manuellement dans notre première version seront ici **appris automatiquement s'ils s'avèrent efficaces**.

L'utilisation de la stratégie epsilon-greedy permet aussi d'alterner entre exploration et exploitation, surtout en début de partie, pour **éviter de rester bloqué sur une stratégie sous-optimale.** Cela rend notre programme **moins prévisible** et limite les risques de tomber dans des pièges adverses. Enfin, l'usage de prédicats dynamiques permet une **gestion souple de la mémoire** et de l'apprentissage dans Prolog.

Les inconvenients du Q Learning

Notre approche basée sur le Q-learning, bien qu'**efficace sur le long terme**, présente plusieurs limites importantes. Tout d'abord, elle **nécessite du temps pour devenir performante** : au début de la partie, le programme n'a encore rien appris, donc ses décisions sont souvent basées sur du hasard. Cela signifie que sur un nombre de tours réduit, il sera généralement **moins performant qu'un programme utilisant une stratégie fixe.** En revanche, plus la partie est longue, plus il apprend, et plus il devient redoutable.



Ce type d'apprentissage a aussi un **coût en mémoire**, puisqu'il faut stocker une Q-value pour chaque couple (état, action) rencontré. Si les états sont nombreux, cela peut rapidement devenir volumineux. Il faut aussi souligner que le **choix des paramètres** (comme le taux d'apprentissage ou le taux d'exploration) a un impact considérable sur la qualité de l'apprentissage : de **mauvais réglages peuvent empêcher le programme de progresser**.

Enfin, le comportement du Q-learning peut devenir **instable** si l'adversaire change soudainement de stratégie. Comme le programme s'appuie sur ce qu'il a observé jusque-là, il lui faudra **un certain temps pour s'adapter à ce changement**, ce qui peut être exploité par les adversaires

Tout cela montre bien que notre choix d'utiliser le Q-learning comporte aussi des inconvénients, qui peuvent **fragiliser nos chances de remporter le tournoi**. Sur le court terme, avec peu de tours de jeu, il aurait peut-être été plus stratégique de **conserver une méthode fixe comme l'équilibre de Nash**. Cependant, nous avons trouvé bien plus enrichissant de nous tourner vers un algorithme d'apprentissage. Ce choix nous a permis d'**explorer des concepts nouveaux**, de **mieux comprendre le fonctionnement de l'adaptation**, et surtout, de **coder quelque chose de bien plus stimulant**. C'est ce qui a motivé notre décision d'en faire notre stratégie finale.

CONCLUSION

En conclusion, nous avons d'abord envisagé une stratégie simple fondée sur l'**équilibre de Nash**. Mais en testant notre code et en voyant que beaucoup de groupes avaient fait le même choix, nous avons voulu aller plus loin avec une approche plus originale et adaptative. C'est ainsi que nous avons découvert le **Q-learning,**.

Comprendre le principe du Q-learning a été une première étape passionnante, mais le plus grand défi a été de réussir à l'i**mplémenter dans Prolog**, un langage que nous avions surtout utilisé pour déclarer des faits statiques. Il a fallu repenser notre manière de programmer, apprendre à gérer des **prédicats dynamiques** et concevoir un système capable d'apprendre et de s'adapter au fil des parties.

Aujourd'hui, **on ne sait pas si notre programme gagnera le tournoi.** Peut-être qu'il ne sera pas le plus performant au vu de ses inconvénients. Mais au fond, ce n'est pas ça le plus important. Ce projet nous a permis de **découvrir de nouveaux concepts**, de nous **dépasser techniquement**, et d'**apprendre énormément sur la logique, l'adaptation et la résolution de problèmes**. Alors même si notre code ne fait pas encore ses preuves, nous sommes fières d'avoir réussi à le concevoir et nous considérons cela comme notre accomplissement du projet.