Pense-bête VIP : Modèles basés sur les états

Afshine Amidi et Shervine Amidi

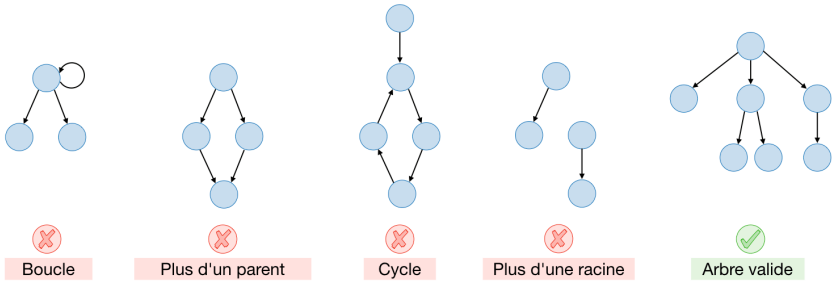
8 septembre 2019

Optimisation de parcours

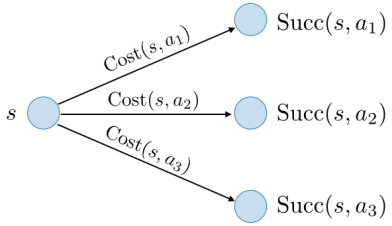
Dans cette section, nous supposons qu'en effectuant une action a à partir d'un état s, on arrive de manière déterministe à l'état Succ(s,a). Le but de cette étude est de déterminer une séquence d'actions (a_1,a_2,a_3,a_4,...) démarrant d'un état initial et aboutissant à un état final. Pour y parvenir, notre objectif est de minimiser le coût associés à ces actions à l'aide de modèles basés sur les états (*state-based model* en anglais).

Parcours d'arbre

Cette catégorie d'algorithmes explore tous les états et actions possibles. Même si leur consommation en mémoire est raisonnable et peut supporter des espaces d'états de taille très grande, ce type d'algorithmes est néanmoins susceptible d'engendrer des complexités en temps exponentielles dans le pire des cas.



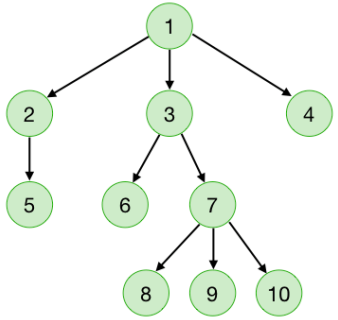
❒ **Problème de recherche** – Un problème de recherche est défini par :
- un état de départ s_{start}
- des actions Actions(s) pouvant être effectuées depuis l'état s
- le coût de l'action Cost(s,a) depuis l'état s pour effectuer l'action a
- le successeur Succ(s,a) de l'état s après avoir effectué l'action a
- la connaissance d'avoir atteint ou non un état final IsEnd(s)



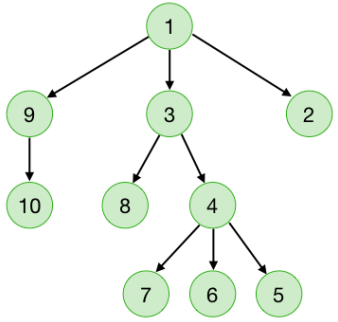
L'objectif est de trouver un chemin minimisant le coût total des actions utilisées.

❒ **Retour sur trace** – L'algorithme de retour sur trace (en anglais *backtracking search*) est un algorithme récursif explorant naïvement toutes les possibilités jusqu'à trouver le chemin de coût minimal. Ici, le coût des actions peut aussi bien être positif que négatif.

❒ **Parcours en largeur (BFS)** – L'algorithme de parcours en largeur (en anglais *breadth-first search* ou *BFS*) est un algorithme de parcours de graphe traversant chaque niveau de manière successive. On peut le coder de manière itérative à l'aide d'une queue stockant à chaque étape les prochains nœuds à visiter. Cet algorithme suppose que le coût de toutes les actions est égal à une constante $c \geqslant 0$.



❒ **Parcours en profondeur (DFS)** – L'algorithme de parcours en profondeur (en anglais *depth-first search* ou *DFS*) est un algorithme de parcours de graphe traversant chaque chemin qu'il emprunte aussi loin que possible. On peut le coder de manière récursive, ou itérative à l'aide d'une pile qui stocke à chaque étape les prochains nœuds à visiter. Cet algorithme suppose que le coût de toutes les actions est égal à 0.



❒ **Approfondissement itératif** – L'astuce de l'approfondissement itératif (en anglais *iterative deepening*) est une modification de l'algorithme de DFS qui l'arrête après avoir atteint une certaine profondeur, garantissant l'optimalité de la solution trouvée quand toutes les actions ont un même coût constant $c \geqslant 0$.

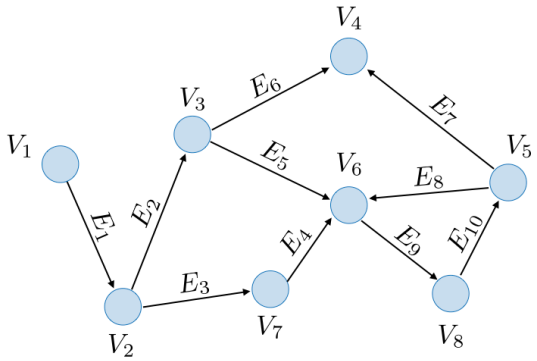
❒ **Récapitulatif des algorithmes de parcours d'arbre** – En notant b le nombre d'actions par état, d la profondeur de la solution et D la profondeur maximale, on a :

Algorithme	Coût des actions	Espace	Temps
Retour sur trace	peu importe	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
Parcours en largeur	$c \geqslant 0$	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$
Parcours en profondeur	0	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
DFS-approfondissement itératif	$c \geqslant 0$	$\mathcal{O}(d)$	$\mathcal{O}(b^d)$

Parcours de graphe

Cette catégorie d'algorithmes basés sur les états vise à trouver des chemins optimaux avec une complexité moins grande qu'exponentielle. Dans cette section, nous allons nous concentrer sur la programmation dynamique et la recherche à coût uniforme.

❒ **Graphe** – Un graphe se compose d'un ensemble de sommets V (aussi appelés nœuds) et d'arêtes E (appelés arcs lorsque le graphe est orienté).

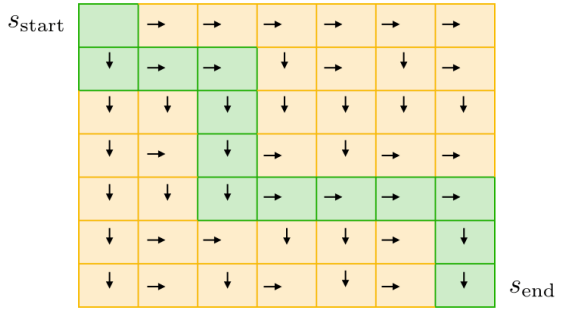


Remarque : un graphe est dit être acyclique lorsqu'il ne contient pas de cycle.

❒ **État** – Un état contient le résumé des actions passées suffisant pour choisir les actions futures de manière optimale.

❒ **Programmation dynamique** – La programmation dynamique (en anglais *dynamic programming* ou *DP*) est un algorithme de recherche de type retour sur trace qui utilise le principe de mémoïsation (i.e. les résultats intermédiaires sont enregistrés) et ayant pour but de trouver le chemin à coût minimal allant de l'état s à l'état final s_{end}. Cette procédure peut potentiellement engendrer des économies exponentielles si on la compare aux algorithmes de parcours de graphe traditionnels, et a la propriété de ne marcher que dans le cas de graphes acycliques. Pour un état s donné, le coût futur est calculé de la manière suivante :

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min\limits_{a \in \text{Actions}(s)} \big[\text{Cost}(s,a) + \text{FutureCost}(\text{Succ}(s,a))\big] & \text{otherwise} \end{cases}$$

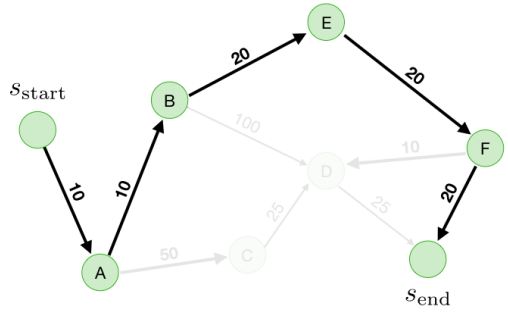


Remarque : la figure ci-dessus illustre une approche ascendante alors que la formule nous donne l'intuition d'une résolution avec une approche descendante.

❒ **Types d'états** – La table ci-dessous présente la terminologie relative aux états dans le contexte de la recherche à coût uniforme :

État	Explication
Exploré $\mathcal{E}$	États pour lesquels le chemin optimal a déjà été trouvé
Frontière $\mathcal{F}$	États rencontrés mais pour lesquels on se demande toujours comment s'y rendre avec un coût minimal
Inexploré $\mathcal{U}$	États non rencontrés jusqu'à présent

❒ **Recherche à coût uniforme** – La recherche à coût uniforme (*uniform cost search* ou *UCS* en anglais) est un algorithme de recherche qui a pour but de trouver le chemin le plus court entre les états s_{start} et s_{end}. Celui-ci explore les états s en les triant par coût croissant de PastCost(s) et repose sur le fait que toutes les actions ont un coût non négatif.



Remarque 1 : UCS fonctionne de la même manière que l'algorithme de Dijkstra.
Remarque 2 : cet algorithme ne marche pas sur une configuration contenant des actions à coût négatif. Quelqu'un pourrait penser à ajouter une constante positive à tous les coûts, mais cela ne résoudrait rien puisque le problème résultant serait différent.

❒ **Théorème de correction** – Lorsqu'un état s passe de la frontière $\mathcal{F}$ à l'ensemble exploré $\mathcal{E}$, sa priorité est égale à PastCost(s), représentant le chemin de coût minimal allant de s_{start} à s.

❒ **Récapitulatif des algorithmes de parcours de graphe** – En notant N le nombre total d'états dont n sont explorés avant l'état final s_{end}, on a :

Algorithme	Acyclicité	Coûts	Temps/Espace
Programmation dynamique	oui	peu importe	$\mathcal{O}(N)$
Recherche à coût uniforme	non	$c \geqslant 0$	$\mathcal{O}(n\log(n))$

Remarque : ce décompte de la complexité suppose que le nombre d'actions possibles à partir de chaque état est constant.

Apprentissage des coûts

Supposons que nous ne sommes pas donnés les valeurs de Cost(s,a). Nous souhaitons estimer ces quantités à partir d'un ensemble d'apprentissage de chemins à coût minimaux d'actions $(a_1, a_2, ..., a_k)$.

❒ **Perceptron structuré** – L'algorithme du perceptron structuré vise à apprendre de manière itérative les coûts des paires état-action. À chaque étape, il :

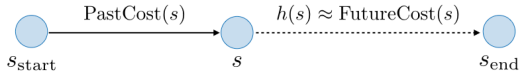
— fait décroître le coût estimé de chaque état-action du vrai chemin minimisant y donné par la base d'apprentissage,

— fait croître le coût estimé de chaque état-action du chemin y' prédit comme étant minimisant par les paramètres appris par l'algorithme.

Remarque : plusieurs versions de cette algorithme existent, l'une d'elles réduisant ce problème à l'apprentissage du coût de chaque action a et l'autre paramétrisant chaque Cost(s,a) à un vecteur de paramètres pouvant être appris.

Algorithme A*

❒ **Fonction heuristique** – Une heuristique est une fonction h opérant sur les états s, où chaque $h(s)$ vise à estimer FutureCost(s), le coût du chemin optimal allant de s à s_{end}.



❒ **Algorithme** – A^* est un algorithme de recherche visant à trouver le chemin le plus court entre un état s et un état final s_{end}. Il le fait en explorant les états s triés par ordre croissant de PastCost$(s) + h(s)$. Cela revient à utiliser l'algorithme UCS où chaque arête est associée au coût Cost$'(s,a)$ donné par :

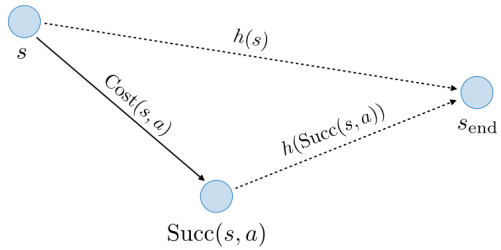
$$\boxed{\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s)}$$

Remarque : cet algorithme peut être vu comme une version biaisée de UCS explorant les états estimés comme étant plus proches de l'état final.

❒ **Consistance** – Une heuristique h est dite consistante si elle satisfait les deux propriétés suivantes :

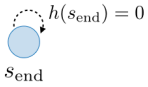
— Pour tous états s et actions a,

$$\boxed{h(s) \leqslant \text{Cost}(s,a) + h(\text{Succ}(s,a))}$$



— L'état final vérifie la propriété :

$$\boxed{h(s_{\text{end}}) = 0}$$



❒ **Correction** – Si h est consistante, alors A^* renvoie le chemin de coût minimal.

❒ **Admissibilité** – Une heuristique h est dite admissible si l'on a :

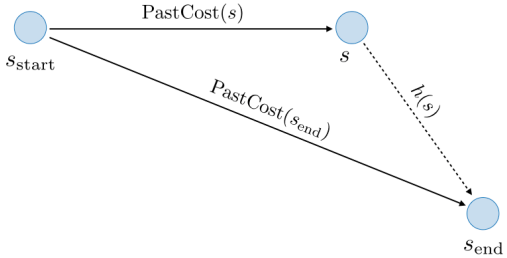
$$\boxed{h(s) \leqslant \text{FutureCost}(s)}$$

❒ **Théorème** – Soit $h(s)$ une heuristique. On a :

$$\boxed{h(s) \text{ consistante} \implies h(s) \text{ admissible}}$$

❒ **Efficacité** – A^* explore les états s satisfaisant l'équation :

$$\boxed{\text{PastCost}(s) \leqslant \text{PastCost}(s_{\text{end}}) - h(s)}$$



Remarque : avoir $h(s)$ élevé est préférable puisque cette équation montre que le nombre d'états s à explorer est alors réduit.

Relaxation

C'est un type de procédure permettant de produire des heuristiques consistantes. L'idée est de trouver une fonction de coût facile à exprimer en enlevant des contraintes au problème, et ensuite l'utiliser en tant qu'heuristique.

❒ **Relaxation d'un problème de recherche** – La relaxation d'un problème de recherche P aux coûts Cost est noté P_{rel} avec coûts Cost_{rel}, et vérifie la relation :

$$\boxed{\text{Cost}_{\text{rel}}(s,a) \leqslant \text{Cost}(s,a)}$$

❒ **Relaxation d'une heuristique** – Étant donné la relaxation d'un problème de recherche P_{rel}, on définit l'heuristique relaxée $h(s) = \text{FutureCost}_{\text{rel}}(s)$ comme étant le chemin de coût minimal allant de s à un état final dans le graphe de fonction de coût $\text{Cost}_{\text{rel}}(s,a)$.

❒ **Consistance de la relaxation d'heuristiques** – Soit P_{rel} une relaxation d'un problème de recherche. Par théorème, on a :

$$\boxed{h(s) = \text{FutureCost}_{\text{rel}}(s) \implies h(s) \text{ consistante}}$$

❒ **Compromis lors du choix d'heuristique** – Le choix d'heuristique se repose sur un compromis entre :

- Complexité de calcul : $h(s) = \text{FutureCost}_{\text{rel}}(s)$ doit être facile à calculer. De manière préférable, cette fonction peut s'exprimer de manière explicite et elle permet de diviser le problème en sous-parties indépendantes.
- Approximation adéquate : l'heuristique $h(s)$ devrait être assez proche de $\text{FutureCost}(s)$ ce qui veut dire qu'il ne faudrait pas enlever trop de contraintes.

❒ **Heuristique max** – Soient $h_1(s)$, $h_2(s)$ deux heuristiques. On a la propriété suivante :

$$\boxed{h_1(s),\, h_2(s) \text{ consistante} \implies h(s) = \max\{h_1(s),\, h_2(s)\} \text{ consistante}}$$

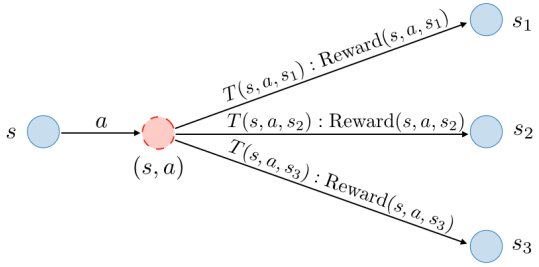
Processus de décision markovien

Dans cette section, on suppose qu'effectuer l'action a à partir de l'état s peut mener de manière probabiliste à plusieurs états s'_1,s'_2,... Dans le but de trouver ce qu'il faudrait faire entre un état initial et un état final, on souhaite trouver une stratégie maximisant la quantité des récompenses en utilisant un outil adapté à l'imprévisibilité et l'incertitude : les processus de décision markoviens.

Notations

❒ **Définition** – 'objectif d'un processus de décision markovien (en anglais *Markov decision process* ou *MDP*) est de maximiser la quantité de récompenses. Un tel problème est défini par :

- un état de départ s_{start}
- l'ensemble des actions $\text{Actions}(s)$ pouvant être effectuées à partir de l'état s
- la probabilité de transition $T(s,a,s')$ de l'état s vers l'état s' après avoir pris l'action a
- la récompense $\text{Reward}(s,a,s')$ pour être passé de l'état s à l'état s' après avoir pris l'action a
- la connaissance d'avoir atteint ou non un état final $\text{IsEnd}(s)$
- un facteur de dévaluation $0 \leqslant \gamma \leqslant 1$



❒ **Probabilités de transition** – La probabilité de transition $T(s,a,s')$ représente la probabilité de transitionner vers l'état s' après avoir effectué l'action a en étant dans l'état s. Chaque $s' \mapsto T(s,a,s')$ est une loi de probabilité :

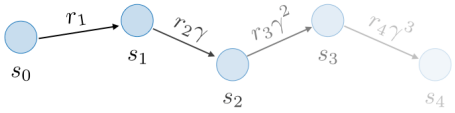
$$\forall s,a, \quad \boxed{\sum_{s' \in \text{ States}} T(s,a,s') = 1}$$

❒ **Politique** – Une politique π est une fonction liant chaque état s à une action a, i.e.

$$\boxed{\pi : s \mapsto a}$$

❒ **Utilité** – L'utilité d'un chemin $(s_0, ..., s_k)$ est la somme des récompenses dévaluées récoltées sur ce chemin. En d'autres termes,

$$\boxed{u(s_0,...,s_k) = \sum_{i=1}^{k} r_i \gamma^{i-1}}$$



Remarque : la figure ci-dessus illustre le cas $k = 4$.

❒ **Q-value** – La fonction de valeur des états-actions (*Q-value* en anglais) d'une politique π évaluée à l'état s avec l'action a, aussi notée $Q_\pi(s,a)$, est l'espérance de l'utilité partant de l'état s avec l'action a et adoptant ensuite la politique π. Cette fonction est définie par :

$$\boxed{Q_\pi(s,a) = \sum_{s' \in \text{ States}} T(s,a,s') \left[\text{Reward}(s,a,s') + \gamma V_\pi(s')\right]}$$

❒ **Fonction de valeur des états d'une politique** – La fonction de valeur des états d'une politique π évaluée à l'état s, aussi notée $V_\pi(s)$, est l'espérance de l'utilité partant de l'état s et adoptant ensuite la politique π. Cette fonction est définie par :

$$\boxed{V_\pi(s) = Q_\pi(s,\pi(s))}$$

Remarque : $V_\pi(s)$ vaut 0 si s est un état final.

Applications

❒ **Évaluation d'une politique** – Étant donnée une politique π, on peut utiliser l'algorithme itératif d'évaluation de politiques (en anglais *policy evaluation*) pour estimer V_π :

— Initialisation : pour tous les états s, on a

$$\boxed{V_\pi^{(0)}(s) \longleftarrow 0}$$

— Itération : pour t allant de 1 à T_{PE}, on a

$$\forall s, \quad \boxed{V_\pi^{(t)}(s) \longleftarrow Q_\pi^{(t-1)}(s,\pi(s))}$$

avec

$$\boxed{Q_\pi^{(t-1)}(s,\pi(s)) = \sum_{s' \in \text{ States}} T(s,\pi(s),s')\Big[\text{Reward}(s,\pi(s),s') + \gamma V_\pi^{(t-1)}(s')\Big]}$$

Remarque : en notant S le nombre d'états, A le nombre d'actions par états, S' le nombre de successeurs et T le nombre d'itérations, la complexité en temps est alors de $\mathcal{O}(T_{PE}SS')$.

❒ **Q-value optimale** – La Q-value optimale $Q_{\text{opt}}(s,a)$ d'un état s avec l'action a est définie comme étant la Q-value maximale atteinte avec n'importe quelle politique. Elle est calculée avec la formule :

$$\boxed{Q_{\text{opt}}(s,a) = \sum_{s' \in \text{ States}} T(s,a,s')\Big[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')\Big]}$$

❒ **Valeur optimale** – La valeur optimale $V_{\text{opt}}(s)$ d'un état s est définie comme étant la valeur maximum atteinte par n'importe quelle politique. Elle est calculée avec la formule :

$$\boxed{V_{\text{opt}}(s) = \max_{a \in \text{ Actions}(s)} Q_{\text{opt}}(s,a)}$$

❒ **Politique optimale** – La politique optimale π_{opt} est définie comme étant la politique liée aux valeurs optimales. Elle est définie par :

$$\forall s, \quad \boxed{\pi_{\text{opt}}(s) = \operatorname*{argmax}_{a \in \text{ Actions}(s)} Q_{\text{opt}}(s,a)}$$

❒ **Itération sur la valeur** – L'algorithme d'itération sur la valeur (en anglais *value iteration*) vise à trouver la valeur optimale V_{opt} ainsi que la politique optimale π_{opt} en deux temps :

— Initialisation : pour tout état s, on a

$$\boxed{V_{\text{opt}}^{(0)}(s) \longleftarrow 0}$$

— Itération : pour t allant de 1 à T_{VI}, on a

$$\forall s, \quad \boxed{V_{\text{opt}}^{(t)}(s) \longleftarrow \max_{a \in \text{ Actions}(s)} Q_{\text{opt}}^{(t-1)}(s,a)}$$

avec

$$\boxed{Q_{\text{opt}}^{(t-1)}(s,a) = \sum_{s' \in \text{ States}} T(s,a,s')\Big[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')\Big]}$$

Remarque : si $\gamma < 1$ ou si le graphe associé au processus de décision markovien est acyclique, alors l'algorithme d'itération sur la valeur est garanti de converger vers la bonne solution.

Cas des transitions et récompenses inconnues

On suppose maintenant que les probabilités de transition et les récompenses sont inconnues.

❒ **Monte-Carlo basé sur modèle** – La méthode de Monte-Carlo basée sur modèle (en anglais *model-based Monte Carlo*) vise à estimer $T(s,a,s')$ et $\text{Reward}(s,a,s')$ en utilisant des simulations de Monte-Carlo avec :

$$\boxed{\widehat{T}(s,a,s') = \frac{\#\text{ de fois où } (s,a,s') \text{ se produit}}{\#\text{ de fois où } (s,a) \text{ se produit}}}$$

and

$$\boxed{\widehat{\text{Reward}}(s,a,s') = r \text{ dans } (s,a,r,s')}$$

Ces estimations sont ensuite utilisées pour trouver les Q-values, ainsi que Q_π et Q_{opt}.

Remarque : la méthode de Monte-Carlo basée sur modèle est dite "hors politique" (en anglais "off-policy") car l'estimation produite ne dépend pas de la politique utilisée.

❒ **Monte-Carlo sans modèle** – La méthode de Monte-Carlo sans modèle (en anglais *model-free Monte Carlo*) vise à directement estimer Q_π de la manière suivante :

$$\boxed{\widehat{Q}_\pi(s,a) = \text{moyenne de } u_t \text{ où } s_{t-1} = s, a_t = a}$$

où u_t désigne l'utilité à partir de l'étape t d'un épisode donné.

Remarque : la méthode de Monte-Carlo sans modèle est dite "sur politique" (en anglais "on-policy") car l'estimation produite dépend de la politique π utilisée pour générer les données.

❒ **Formulation équivalente** – En introduisant la constante $\eta = \frac{1}{1+(\#\text{mises à jour } (s,a))}$ et pour chaque triplet (s,a,u) de la base d'apprentissage, la formule de récurrence de la méthode de Monte-Carlo sans modèle s'écrit à l'aide de la combinaison convexe :

$$\boxed{\widehat{Q}_\pi(s,a) \leftarrow (1-\eta)\widehat{Q}_\pi(s,a) + \eta u}$$

ainsi qu'une formulation mettant en valeur une sorte de gradient :

$$\boxed{\widehat{Q}_\pi(s,a) \leftarrow \widehat{Q}_\pi(s,a) - \eta(\widehat{Q}_\pi(s,a) - u)}$$

❒ **SARSA** – État-action-récompense-état-action (en anglais *state-action-reward-state-action* ou *SARSA*) est une méthode de bootstrap qui estime Q_π en utilisant à la fois des données réelles et estimées dans sa formule de mise à jour. Pour chaque (s,a,r,s',a'), on a :

$$\boxed{\widehat{Q}_\pi(s,a) \longleftarrow (1-\eta)\widehat{Q}_\pi(s,a) + \eta\Big[r + \gamma\widehat{Q}_\pi(s',a')\Big]}$$

Remarque : l'estimation donnée par SARSA est mise à jour à la volée contrairement à celle donnée par la méthode de Monte-Carlo sans modèle où la mise à jour est uniquement effectuée à la fin de l'épisode.

❒ **Q-learning** – Le Q-apprentissage (en anglais *Q-learning*) est un algorithme hors politique (en anglais *off-policy*) donnant une estimation de Q_{opt}. Pour chaque (s,a,r,s',a'), on a :

$$\widehat{Q}_{\text{opt}}(s,a) \leftarrow (1-\eta)\widehat{Q}_{\text{opt}}(s,a) + \eta\left[r + \gamma \max_{a' \in \text{ Actions}(s')} \widehat{Q}_{\text{opt}}(s',a')\right]$$

❒ **Epsilon-glouton** – La politique epsilon-gloutonne (en anglais *epsilon-greedy*) est un algorithme essayant de trouver un compromis entre l'exploration avec probabilité ϵ et l'exploitation avec probabilité $1-\epsilon$. Pour un état s, la politique π_{act} est calculée par :

$$\pi_{\text{act}}(s) = \begin{cases} \underset{a \in \text{ Actions}}{\text{argmax}} \ \widehat{Q}_{\text{opt}}(s,a) & \text{avec proba } 1-\epsilon \\ \text{random from Actions}(s) & \text{avec proba } \epsilon \end{cases}$$

Jeux

Dans les jeux (e.g. échecs, backgammon, Go), d'autres agents sont présents et doivent être pris en compte au moment d'élaborer une politique.

❒ **Arbre de jeu** – Un arbre de jeu est un arbre détaillant toutes les issues possibles d'un jeu. En particulier, chaque nœud représente un point de décision pour un joueur et chaque chemin liant la racine à une des feuilles traduit une possible instance du jeu.

❒ **Jeu à somme nulle à deux joueurs** – C'est un type jeu où chaque état est entièrement observé et où les joueurs jouent de manière successive. On le définit par :

- un état de départ s_{start}
- de possibles actions $\text{Actions}(s)$ partant de l'état s
- du successeur $\text{Succ}(s,a)$ de l'état s après avoir effectué l'action a
- la connaissance d'avoir atteint ou non un état final $\text{IsEnd}(s)$
- l'utilité de l'agent $\text{Utility}(s)$ à l'état final s
- le joueur $\text{Player}(s)$ qui contrôle l'état s

Remarque : nous assumerons que l'utilité de l'agent a le signe opposé de celui de son adversaire.

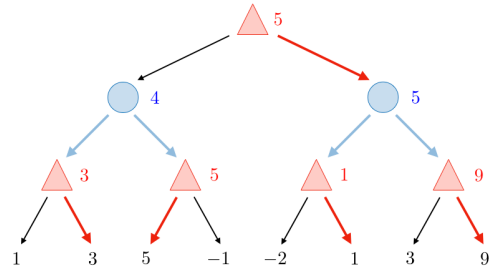
❒ **Types de politiques** – Il y a deux types de politiques :

- Les politiques déterministes, notées $\pi_p(s)$, qui représentent pour tout s l'action que le joueur p prend dans l'état s.
- Les politiques stochastiques, notées $\pi_p(s,a) \in [0,1]$, qui sont décrites pour tout s et a par la probabilité que le joueur p prenne l'action a dans l'état s.

❒ **Expectimax** – Pour un état donné s, la valeur d'expectimax $V_{\text{exptmax}}(s)$ est l'utilité maximum sur l'ensemble des politiques utilisées par l'agent lorsque celui-ci joue avec un adversaire de politique connue π_{opp}. Cette valeur est calculée de la manière suivante :

$$V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max\limits_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum\limits_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

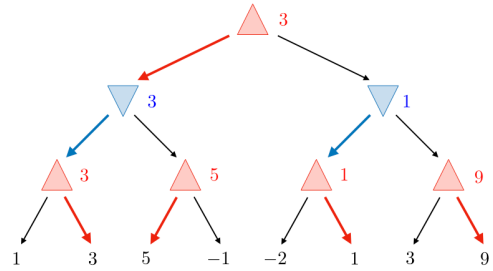
Remarque : expectimax est l'analogue de l'algorithme d'itération sur la valeur pour les MDPs.



❒ **Minimax** – Le but des politiques minimax est de trouver une politique optimale contre un adversaire que l'on assume effectuer toutes les pires actions, i.e. toutes celles qui minimisent l'utilité de l'agent. La valeur correspondante est calculée par :

$$V_{\text{minimax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max\limits_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min\limits_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

Remarque : on peut déduire π_{max} et π_{min} à partir de la valeur minimax $V_{minimax}$.



❒ **Propriétés de minimax** – En notant V la fonction de valeur, il y a 3 propriétés sur minimax qu'il faut avoir à l'esprit :

- *Propriété 1* : si l'agent changeait sa politique en un quelconque π_{agent}, alors il ne s'en sortirait pas mieux.

$$\forall \pi_{\text{agent}}, \quad V(\pi_{\max}, \pi_{\min}) \geqslant V(\pi_{\text{agent}}, \pi_{\min})$$

- *Propriété 2* : si son adversaire change sa politique de $\pi_{\min}$ à π_{opp}, alors il ne s'en sortira pas mieux.

$$\forall \pi_{\text{opp}}, \quad V(\pi_{\max}, \pi_{\min}) \leqslant V(\pi_{\max}, \pi_{\text{opp}})$$

- *Propriété 3* : si l'on sait que son adversaire ne joue pas les pires actions possibles, alors la politique minimax peut ne pas être optimale pour l'agent.

$$\forall \pi, \quad V(\pi_{\max}, \pi) \leqslant V(\pi_{\text{exptmax}}, \pi)$$

À la fin, on a la relation suivante :

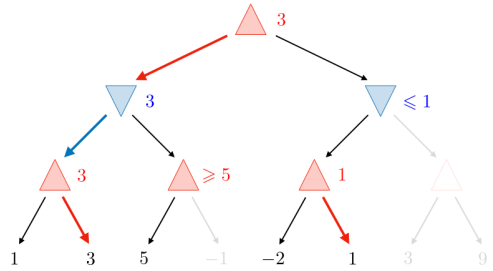
$$V(\pi_{\text{exptmax}}, \pi_{\min}) \leqslant V(\pi_{\max}, \pi_{\min}) \leqslant V(\pi_{\max}, \pi) \leqslant V(\pi_{\text{exptmax}}, \pi)$$

Accélération de minimax

❒ **Fonction d'évaluation** – Une fonction d'évaluation estime de manière approximative la valeur $V_{\text{minimax}}(s)$ selon les paramètres du problème. Elle est notée $\text{Eval}(s)$.

Remarque : l'analogue de cette fonction utilisé dans les problèmes de recherche est FutureCost(s).

❒ **Élagage alpha-bêta** – L'élagage alpha-bêta (en anglais *alpha-beta pruning*) est une méthode exacte d'optimisation employée sur l'algorithme de minimax et a pour but d'éviter l'exploration de parties inutiles de l'arbre de jeu. Pour ce faire, chaque joueur garde en mémoire la meilleure valeur qu'il puisse espérer (appelée α chez le joueur maximisant et β chez le joueur minimisant). À une étape donnée, la condition $\beta < \alpha$ signifie que le chemin optimal ne peut pas passer par la branche actuelle puisque le joueur qui précédait avait une meilleure option à sa disposition.



❒ **TD learning** – L'apprentissage par différence de temps (en anglais *temporal difference learning* ou *TD learning*) est une méthode utilisée lorsque l'on ne connait pas les transitions/récompenses. La valeur et alors basée sur la politique d'exploration. Pour pouvoir l'utiliser, on a besoin de connaître les règles du jeu $\text{Succ}(s,a)$. Pour chaque (s,a,r,s'), la mise à jour des coefficients est faite de la manière suivante :

$$\boxed{w \longleftarrow w - \eta\Big[V(s,w) - (r + \gamma V(s',w))\Big]\nabla_w V(s,w)}$$

Jeux simultanés

Ce cas est opposé aux jeux joués tour à tour. Il n'y a pas d'ordre prédéterminé sur le mouvement du joueur.

❒ **Jeu simultané à un mouvement** – Soient deux joueurs A et B, munis de possibles actions. On note $V(a,b)$ l'utilité de A si A choisit l'action a et B l'action b. V est appelée la matrice de profit (en anglais *payoff matrix*).

❒ **Stratégies** – Il y a principalement deux types de stratégies :

— Une stratégie pure est une seule action :

$$\boxed{a \in \text{Actions}}$$

— Une stratégie mixte est une loi de probabilité sur les actions :

$$\forall a \in \text{Actions}, \quad \boxed{0 \leqslant \pi(a) \leqslant 1}$$

❒ **Évaluation de jeu** – La valeur d'un jeu $V(\pi_A,\pi_B)$ quand le joueur A suit π_A et le joueur B suit π_B est telle que :

$$\boxed{V(\pi_A,\pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a,b)}$$

❒ **Théorème minimax** – Soient π_A et π_B des stratégies mixtes. Pour chaque jeu à somme nulle à deux joueurs ayant un nombre fini d'actions, on a :

$$\boxed{\max_{\pi_A}\min_{\pi_B} V(\pi_A,\pi_B) = \min_{\pi_B}\max_{\pi_A} V(\pi_A,\pi_B)}$$

Jeux à somme non nulle

❒ **Matrice de profit** – On définit $V_p(\pi_A,\pi_B)$ l'utilité du joueur p.

❒ **Équilibre de Nash** – Un équilibre de Nash est défini par (π_A^*,π_B^*) tel qu'aucun joueur n'a d'intérêt de changer sa stratégie. On a :

$$\boxed{\forall \pi_A, V_A(\pi_A^*,\pi_B^*) \geqslant V_A(\pi_A,\pi_B^*)} \quad \text{et} \quad \boxed{\forall \pi_B, V_B(\pi_A^*,\pi_B^*) \geqslant V_B(\pi_A^*,\pi_B)}$$

Remarque : dans un jeu à nombre de joueurs et d'actions finis, il existe au moins un équilibre de Nash.